

---

# Query Rewriting Optimisation Techniques for Ontology-Based Data Access



José Mora López

Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

A thesis submitted for the degree of

*Doctor of Computer Science*

Supervisors

Oscar Corcho

2015

I would like to dedicate this thesis to my family.

---

## Acknowledgements

In the first place I must acknowledge the labor, the insight and the superhuman responsiveness of my supervisor, Oscar Corcho, whose guidance has been key in this thesis. I would like to manifest my gratitude towards Asunción Gómez Pérez, who gave me the opportunity to work at the Ontology Engineering Group, where I carried out most of my doctoral thesis.

Similarly, I would like to thank José Luis Ambite, Boris Motik and Riccardo Rosati for the respective research visits to the Information Sciences Institute (Los Angeles), the Information Systems Group (Oxford), and the Artificial Intelligence and Knowledge Representation group (Rome). They all three were very enlightening and fulfilling experiences. Perhaps the time for these research visits was short, but they were intense and remarkable times.

I cannot forget (nor reasonably list here) about all the people that I met during this particular journey. At least I can explicitly thank, in no specific order, to: Adrián, José Ángel, Ángel, Raúl, Boris, Jean Paul, Freddy, Elena, Rafa, Mari Carmen, Miguel, Alejandro, Carlos, Dani, Esther, Víctor, Francisco, María, Nandana, Jorge, Óscar, Luis, Andrés, Idafen, Filip, Ulf, Ernesto, Ana, Despoina, Valerio, Fabio, Markus, et al. really each and every one of them. In some cases we had the chance to collaborate in some project, in some others we simply discussed about work, life and everything in the lab and out of it. Definitively everything helped. Special thanks to Miguel Ángel and Raúl for keeping everything running, in particular my computer, during all these years.

También quiero agradecer a mi familia todo el apoyo y ayuda que me ha prestado en este tiempo, esto no habría sido posible sin ellos. Cuando alguien hace una inversión considerable en tiempo y esfuerzo, los más cercanos participan en ella de forma indirecta. A esa inversión le corresponde un retorno positivo y así procuraré que sea. Por el momento, gracias.

The work presented in this thesis has been mostly funded by an PIF grant (Personal Investigador en Formación) from UPM (Universidad Politécnica de Madrid) (RR01/2008) and additionally by the Spanish national project myBigData (TIN2010-17060).

As usual, actions speak louder than words, and I would say that they do so with greater honesty. I hope that I have been able to correspond to the acknowledgments that here I describe, and to the ones that I have not been able to put into words, as much as I hope that I will be able to repair any point in which I may have not yet been able to reciprocate.

# Abstract

Ontology-Based Data Access (OBDA) allows accessing different kinds of data sources (traditionally databases) using a more abstract model provided by an ontology. Query rewriting uses such ontology to rewrite a query into a rewritten query that can be evaluated on the data source. The rewritten queries retrieve the answers that are entailed by the combination of the data explicitly stored in the data source, the original query and the ontology. However, producing and evaluating the rewritten queries are both costly processes that become generally more complex as the expressiveness and size of the ontology and queries increase. In this thesis we explore several optimisations that can be performed both in the rewriting process and in the rewritten queries to improve the applicability of OBDA in real contexts.

Our main technical contribution is a query rewriting system that implements the optimisations presented in this thesis. These optimisations are the core contributions of the thesis and can be grouped into three different groups:

- optimisations that can be applied when considering the predicates in the ontology that are actually mapped to the data sources.
- engineering optimisations that can be applied by handling the process of query rewriting in a way that permits to reduce the computational load of the query generation process.
- optimisations that can be applied when considering additional meta-information about the characteristics of the ABox.

In this thesis we provide formal proofs for the correctness of the proposed optimisations, and an empirical evaluation about the impact of the optimisations. As an additional contribution, part of this empirical approach, we propose a benchmark for the evaluation of query rewriting systems. We also provide some guidelines for the creation and expansion of this kind of benchmarks.



## Resumen

Ontology-Based Data Access (OBDA) permite el acceso a diferentes tipos de fuentes de datos (tradicionalmente bases de datos) usando un modelo más abstracto proporcionado por una ontología. La reescritura de consultas (query rewriting) usa una ontología para reescribir una consulta en una consulta reescrita que puede ser evaluada en la fuente de datos. Las consultas reescritas recuperan las respuestas que están implicadas por la combinación de los datos explícitamente almacenados en la fuente de datos, la consulta original y la ontología. Al trabajar sólo sobre las queries, la reescritura de consultas permite OBDA sobre cualquier fuente de datos que puede ser consultada, independientemente de las posibilidades para modificarla. Sin embargo, producir y evaluar las consultas reescritas son procesos costosos que suelen volverse más complejos conforme la expresividad y tamaño de la ontología y las consultas aumentan. En esta tesis exploramos distintas optimizaciones que pueden ser realizadas tanto en el proceso de reescritura como en las consultas reescritas para mejorar la aplicabilidad de OBDA en contextos realistas.

Nuestra contribución técnica principal es un sistema de reescritura de consultas que implementa las optimizaciones presentadas en esta tesis. Estas optimizaciones son las contribuciones principales de la tesis y se pueden agrupar en tres grupos diferentes:

- optimizaciones que se pueden aplicar al considerar los predicados en la ontología que no están realmente mapeados con las fuentes de datos.
- optimizaciones en ingeniería que se pueden aplicar al manejar el proceso de reescritura de consultas en una forma que permite

reducir la carga computacional del proceso de generación de consultas reescritas.

- optimizaciones que se pueden aplicar al considerar metainformación adicional acerca de las características de la ABox.

En esta tesis proporcionamos demostraciones formales acerca de la corrección y completitud de las optimizaciones propuestas, y una evaluación empírica acerca del impacto de estas optimizaciones. Como contribución adicional, parte de este enfoque empírico, proponemos un banco de pruebas (benchmark) para la evaluación de los sistemas de reescritura de consultas. Adicionalmente, proporcionamos algunas directrices para la creación y expansión de esta clase de bancos de pruebas.

# Contents

<b>Contents</b>	<b>ix</b>
<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xv</b>
<b>Nomenclature</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background and state of the art</b>	<b>7</b>
2.1 Preliminaries . . . . .	7
2.1.1 Subsumption . . . . .	11
2.1.2 Resolution . . . . .	11
2.1.3 Cycles and recursion . . . . .	13
2.2 Query rewriting in query answering . . . . .	15
2.3 OBDA-related logics . . . . .	19
2.3.1 $\mathcal{ELHIQ}$ . . . . .	21
2.4 OBDA implementations . . . . .	22
2.4.1 Previous efforts in benchmarking query rewriting systems .	27
2.4.2 A general algorithm for query rewriting . . . . .	29
2.4.3 The REQUIEM Algorithm . . . . .	32
2.5 OBDA mappings in query rewriting . . . . .	33
2.5.1 Mapping characterisation . . . . .	36
2.6 EBoxes and query rewriting . . . . .	38
2.7 Limitations in the state of the art . . . . .	41

## CONTENTS

---

<b>3</b>	<b>Objectives</b>	<b>45</b>
3.1	Problem statement . . . . .	45
3.2	Hypotheses . . . . .	48
3.2.1	Assumptions . . . . .	48
3.2.2	Limitations . . . . .	49
3.3	Contributions . . . . .	50
<b>4</b>	<b>Query rewriting optimisations in the presence of RDB2RDF mappings</b>	<b>51</b>
4.1	Preliminaries . . . . .	54
4.2	The OBDA algorithm for kyrie . . . . .	56
4.2.1	Intuitive description . . . . .	56
4.2.2	A running example . . . . .	57
4.2.3	Pseudocode of the algorithm . . . . .	59
4.3	Optimisations . . . . .	64
4.3.1	Mapping-based reachability . . . . .	64
4.3.2	Resolution with mappings . . . . .	65
4.4	Proofs . . . . .	66
4.5	Conclusions . . . . .	68
<b>5</b>	<b>Engineering optimisations for query rewriting</b>	<b>71</b>
5.1	The OBDA algorithm for kyrie2 . . . . .	74
5.1.1	Intuitive Description . . . . .	74
5.1.2	A running example . . . . .	75
5.1.3	Pseudocode of the algorithms . . . . .	92
5.2	Optimisations . . . . .	96
5.2.1	Ontology preprocessing . . . . .	96
5.2.2	Query subsumption check . . . . .	99
5.2.3	Prioritising some inferences . . . . .	102
5.2.4	Constraining searches . . . . .	105
5.3	Proofs . . . . .	106
5.4	Conclusions . . . . .	108

<b>6</b>	<b>Optimisations in the presence of an EBox</b>	<b>111</b>
6.1	Preliminaries . . . . .	114
6.2	The OBDA algorithm for <i>kyrie3</i> . . . . .	117
6.2.1	Intuitive description . . . . .	117
6.2.2	A running example . . . . .	117
6.2.3	Pseudocode of the algorithms . . . . .	131
6.3	Optimisations . . . . .	135
6.3.1	Deletion of strongly connected components . . . . .	136
6.3.2	Removal of predicates with empty extension . . . . .	139
6.3.3	Deletion of extensionally subsumed clauses . . . . .	139
6.3.4	Extensional condensation of clauses . . . . .	141
6.4	Proofs . . . . .	141
6.5	Conclusions . . . . .	146
<b>7</b>	<b>Evaluation</b>	<b>149</b>
7.1	Dimensions in query rewriting evaluation . . . . .	149
7.1.1	Determine the ontology language expressiveness . . . . .	149
7.1.2	Characterise the impact of reduced expressiveness in each approach . . . . .	150
7.1.3	Determine the complexity of rewritten queries . . . . .	151
7.1.4	Determine the impact of the complexity of original queries . . . . .	152
7.1.5	Usage of additional information for the query rewriting pro- cess . . . . .	153
7.2	Assets used for evaluations . . . . .	153
7.3	Evaluation for the optimisations in the presence of mappings . . . . .	156
7.4	Evaluation for the engineering optimisations . . . . .	158
7.5	Evaluation for the optimisations in the presence of an EBox . . . . .	163
<b>8</b>	<b>Conclusions</b>	<b>167</b>
8.1	Conclusions on Mapping Optimisations . . . . .	167
8.2	Conclusions on Engineering Optimisations . . . . .	171
8.3	Conclusions on EBox Optimisations . . . . .	172
8.4	Conclusions from the evaluation . . . . .	173

## CONTENTS

---

8.5	General Conclusions . . . . .	175
8.6	Future work . . . . .	177
	<b>References</b>	<b>181</b>

# List of Figures

1.1	OBDA process with the usual stages for query rewriting. . . . .	4
2.1	Stages in the general algorithm . . . . .	29
2.2	Reasons for the lack of mappings for some terms. . . . .	36
4.1	OBDA process with mapping-based query rewriting. . . . .	53
4.2	Stages in the kyrie algorithm . . . . .	58
5.1	OBDA process with query rewriting and a preprocessing stage. . .	73
5.2	Stages in the kyrie2 algorithm . . . . .	75
6.1	OBDA process with query rewriting in the presence of an EBox .	112
6.2	Stages in the kyrie3 algorithm . . . . .	118
7.1	Average rewriting times relating size and cover. All queries and ontologies considered. “Reverse” is zero in all cases. . . . .	165
7.2	Average rewriting sizes relating size and cover. All queries and ontologies considered. “Reverse” is zero in all cases. . . . .	166
8.1	Input dimensions that may affect benchmark representativeness. .	173
8.2	Output dimensions that may affect the representativeness of the evaluations done with benchmarks. . . . .	174

## LIST OF FIGURES

---



# List of Tables

2.1	Overview of the main logics in the state of the art. . . . .	19
2.2	Correspondence between $\mathcal{ELKJO}$ axioms and FOL clauses . . . .	22
2.3	Main systems for query rewriting in the state of the art . . . . .	26
4.1	Mapping coverage for the test cases in FAO . . . . .	54
7.1	Results of the evaluation for the kyrie algorithm . . . . .	157
7.2	Results of the execution to obtain datalog (time in ms) . . . . .	160
7.3	Results of the execution to obtain UCQ (time in ms) . . . . .	160
7.4	Results of ontology preprocessing in kyrie2 . . . . .	161
7.5	Results for ontology V (original size 222 clauses) with EBoxes I, II, III and IV. . . . .	164

## LIST OF TABLES

---

# Chapter 1

## Introduction

Ontology Based Data Access (OBDA) is an information integration paradigm that consists on superimposing a conceptual layer as a view to an underlying information system. This conceptual layer abstracts away from how that information is maintained in the data layer and provides inference capabilities. In OBDA, the conceptual layer is represented using ontologies, while the data layer is commonly supported by relational databases [Calvanese et al., 2007b].

OBDA approaches present the advantage that users can pose queries that benefit from the expressiveness and inference provided by ontologies, while still relying on underlying systems that may be better suited to store large amounts of data (e.g. databases), to query real time data from sensors, to provide data by means of APIs, among other examples.

OBDA has been mainly applied to the combination of description logic TBoxes with relational databases, which are the predominant type of data sources. Therefore, OBDA stands for ontology-based data access and ontology-based database access in an interchangeable way. However, it is not uncommon to find other works focused on providing OBDA support for other types of data sources, such as data streams [Calbimonte et al., 2010], spreadsheets [Priyatna et al., 2013], REST APIs, etc. In all these cases, mappings are commonly used to specify the relationships between the ontologies and the schema used by the underlying data sources. These mappings are normally called database-to-ontology mappings, in general, or RDB2RDF mappings, when the resulting instances are transformed (in a materialized or virtual manner) to RDF. In the latter case, R2RML [Das

## 1. INTRODUCTION

---

et al., 2012] is a W3C recommendation that allows defining such type of mappings. These mappings are used in the *query translation* stage of the query answering process, which is detailed later in this section.

OBDA has been traditionally used in information integration scenarios [Paton et al., 2000; Wache et al., 2001], where the use of a global schema for information integration purposes favoured the adoption of ontologies in the role of this global schema. By using an ontology as a global schema, information integration is empowered, due to the reusable and linked nature associated with ontologies. Using ontologies for the global schema provides also additional query capabilities, allowing inferences on the query to provide richer answers, similarly to how deductive object-oriented databases work (as already pointed out in [Calvanese et al., 2007b]). In short we can say that in an OBDA scenario we use an ontology to rewrite a query into a rewritten query that can be used directly over the data source to obtain the results expanded with the expressiveness from the ontology.

However, expressiveness comes at a cost in terms of computational complexity, and thus OBDA has been traditionally an expensive feature which deemed only interesting when the challenges to tackle were on par with the cost of the solution. A greater expressiveness means that the resulting query will normally differ more from the original query. This will usually have two consequences. First, more operations may need to be carried out in the rewriting process. This means that the computational complexity of the rewriting is increased. Second, a more complex query may be produced as the result of query rewriting. This means that the underlying data management system will receive a more complex query that will be harder to process.

Early approaches in OBDA focused on enabling access to data sources with the use of ontologies, drawing inspiration from expert systems and deductive databases. The state of the art in OBDA changed with the first-order rewritability property of some languages. Intuitively this property means that we can use an ontology to rewrite a query over this ontology and a data source into an equivalent first order query exclusively over the data source. Here equivalence means that the answers returned are the same set of answers, namely the “certain answers” as we will see. These first-order queries are specially interesting because they can be converted to first-order languages like SQL. A more detailed explanation of

---

this property is presented in section 2.2.

With this property, the focus was set on query rewriting as one of the key aspects in OBDA. Including query rewriting as a standard part of OBDA allows separating the ontology semantics from the rest of the system, for instance the translation using mappings. By using query rewriting, the query answering problem can be divided into several stages, as depicted in figure 1.1. These stages are the following ones:

- **Query rewriting.** In this stage, ontology-based queries are rewritten into queries that consider the inferences that can be done with the ontology.
- **Query translation.** In this stage the previous queries are transformed into the query language (or API) that complies with the schema of the underlying data sources, so that they can be evaluated by their corresponding query evaluation systems. This stage makes use of the aforementioned mappings.
- **Query evaluation.** The generated queries or API calls are evaluated or executed, and results are obtained according to the underlying data source schema.
- **Result translation.** The results obtained from the evaluation are translated into the original ontology-based schema, so that they can be interpreted by the original issuer of the ontology-based queries.

In this context, the focus of most of the work done so far in OBDA was set on exploring the logic families that are FO-rewritable, the properties of these logics and the algorithms that could be built for them, leaving aside the rest of the OBDA system. The evolution of this new branch in the state of the art was mostly qualitative, exploring what could be done with the different logics. The focus was set on the theoretical complexity of the algorithms, with the efficiency in practice in a background role. Additionally the focus was set on query rewriting, as an isolated stage of the whole process. We will see that considering other elements in the OBDA system can benefit the query rewriting process and its results.

## 1. INTRODUCTION

---

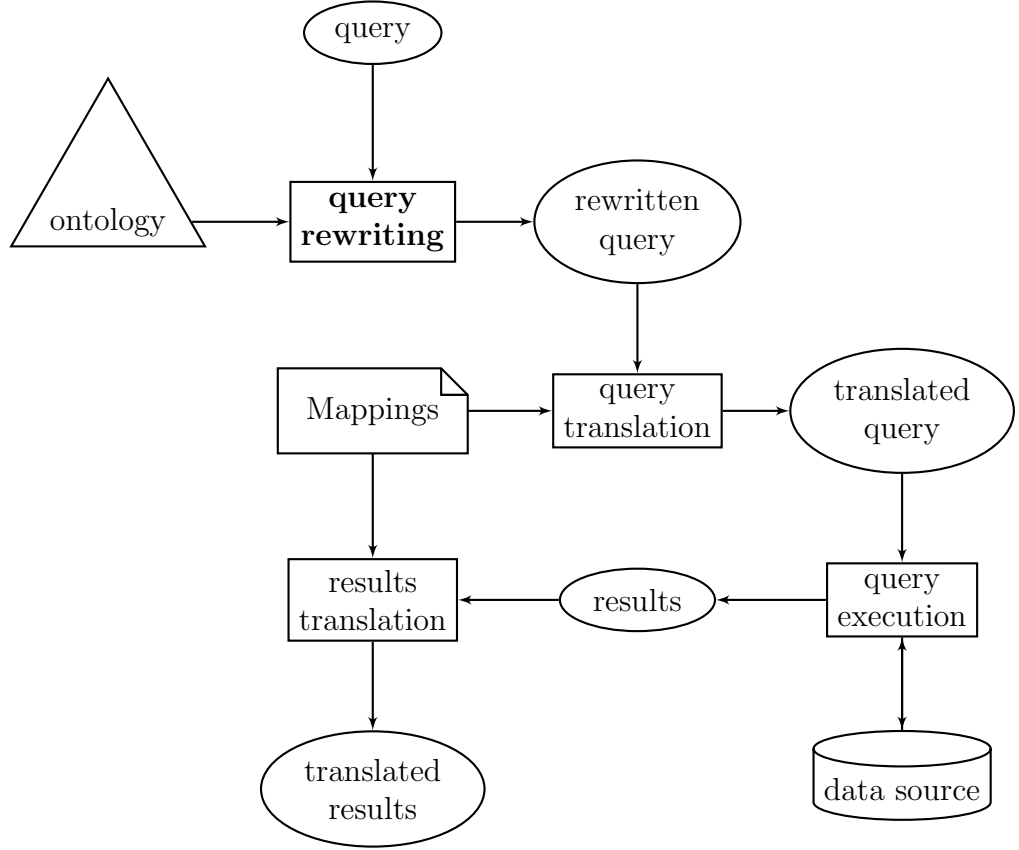


Figure 1.1: OBDA process with the usual stages for query rewriting.

In particular, a fundamental aspect of any OBDA system are mappings. Mappings are normally declarative and provide a description of the schemas that are mapped. More precisely they map two schemata and describe one of them in terms of the other, as we will see in section 2.5. This description provides valuable information about how rewritten queries may behave when posed to the underlying system. With this we can perform further optimisations, as we will see in chapter 4. Moreover, additional information can be extracted from a set of mappings, such as an EBox. An EBox describes the containment dependencies between the extension (in an ABox) of the predicates<sup>1</sup> in a TBox. Once an EBox has been described, the metainformation that it contains can be used for further optimisations, as we will see in chapter 6.

---

<sup>1</sup>We will refer to concepts and properties in a TBox as “predicates”, for convention with other logics, e.g. first-order logic.

---

So far, query rewriting approaches have focused on the query rewriting stage, considering only the query and the ontology. In our work we propose eliminating the predicates that are not mapped according to a set of mappings. We perform this elimination first in the logic program and then in the Datalog program, reducing the time needed to process the queries in the following stages. This reduction in the processing time is usually obtained through a reduction in the size of the queries in the stage where the optimisations are applied and in the following stages.

Besides, we perform some optimisations strictly from an engineering point of view. The research in this case focuses on the algorithmic aspects of the process, mostly focusing on aspects of automated resolution. As a consequence, the inputs and outputs to the process remain the same but the time needed to perform this process is significantly reduced. This is described in chapter 5.

Finally, in chapter 6 we consider not only the mappings as in chapter 4, but also the ABox dependencies contained in the EBox. This allows eliminating predicates that are not mapped and predicates whose mappings or their corresponding values are contained in some other predicate already considered. With this we can eliminate some clauses in the generated output, either Datalog or a union of conjunctive queries, without eliminating any answer to the query, as in previous optimisations.

Contrarily to other approaches that preserve the structure defined in figure 1.1 we will see how OBDA using query rewriting can be modified. We will see this through the chapters and the variations of this figure in each chapter. We will see that the modifications in the architecture require adaptations in the query rewriting step, presenting new challenges. We will also see that with these challenges there are opportunities and room for improvement. On the one hand we will be able to improve the query rewriting process, making it faster. On the other hand we will be able to improve the rewritten queries, obtaining shorter queries that return the same set of answers. With this, we have demonstrated the importance of considering the whole scenario of query rewriting and set the basis for a further exploration of the possibilities that arise in a broader picture.

## 1. INTRODUCTION

---



# Chapter 2

## Background and state of the art

Ontology-Based Data Access (OBDA) has been subject to a large amount of work in the past decades. Several approaches and algorithms for query rewriting and translation have been designed and implemented (Perfect Rewriting, REQUIEM, Rapid, etc.), each one of them focused on different aspects of the problem. In this chapter we describe these approaches, including also the description of the logic languages used in each case. We also pay special attention in this chapter to a technique called Resolution with Free Selection, which was already used in one of the existing approaches (REQUIEM) and which forms the basis for the optimisations proposed in this thesis. We analyse the properties of these implementations with respect to the input that they accept, the output that they generate and the optimisations that they implement in the query rewriting and translation process that they carry out. We consider the relationship of mappings with query rewriting, since considering them provides benefits on the process of query rewriting and the rewritten queries, without having any negative impact on the final answers obtained, as we will see.

### 2.1 Preliminaries

We briefly introduce some necessary terminology on logic programs (with a stress on Datalog), OBDA systems and resolution.

Given  $\mathcal{P}$ ,  $\mathcal{F}$  and  $\mathcal{C}$  the finite or countable sets of respectively predicate, function

## 2. BACKGROUND AND STATE OF THE ART

---

and constant symbols, and  $\mathcal{V}$  a countably infinite set of variables, where predicate and function symbols have some positive integer associated as their arity. We denote a **first-order language** by  $\mathcal{L}$ , and the language determined by  $\mathcal{P}$ ,  $\mathcal{F}$ ,  $\mathcal{C}$  and  $\mathcal{V}$  as  $\mathcal{L}(\mathcal{P}, \mathcal{F}, \mathcal{C}, \mathcal{V})$ , which may be omitted when  $\mathcal{P}$ ,  $\mathcal{F}$ ,  $\mathcal{C}$  and  $\mathcal{V}$  are clear from the context.

The set of terms  $terms(\mathcal{L})$  is the smallest set such that (i)  $\mathcal{C} \cup \mathcal{V} \subseteq terms(\mathcal{L})$ , and (ii) if  $f \in \mathcal{F}$  with arity  $n$ , and  $t_1, \dots, t_n \in terms(\mathcal{L})$  then  $f(t_1, \dots, t_n) \in terms(\mathcal{L})$ . This latter type of terms are called functional terms. We will consider only functional terms of arity 1 in the present work, since those are the only functional terms that we generate from axioms in an ontology. Terms with no variables are ground terms and the set of all ground terms in  $terms(\mathcal{L})$  is the Herbrand Universe ( $U_{\mathcal{L}}$ ).

The set of atoms  $atoms(\mathcal{L})$  is the smallest set containing every expression  $P(t_1, \dots, t_n)$  where  $P \in \mathcal{P}$  with arity  $n$  and  $t_1, \dots, t_n \in terms(\mathcal{L})$ . The Herbrand base  $base(\mathcal{L})$  is the set containing exactly every ground atom in  $atoms(\mathcal{L})$ . We will use  $pred(\beta)$  to refer to the predicate  $P$  of an atom  $\beta = P(t_1, \dots, t_n)$ .

A **Horn clause** is an expression of the form  $\beta_0 \leftarrow \beta_1 \wedge \dots \wedge \beta_n$ , where  $\beta_0$  is an atom called the head, the set of atoms  $\beta_1, \dots, \beta_n$  is the body of the clause and each  $\beta_i$  is an atom. Variables in the head are called distinguished variables and all other variables are called nondistinguished. For a clause  $\gamma$  we will refer to its body and its head with  $body(\gamma)$  and  $head(\gamma)$  respectively. The set of predicates of a clause  $preds(\gamma)$  is the set composed by the predicates of each of the atoms  $pred(\beta_i)$  for all atom  $\beta_i$  in  $\gamma$  (in the head or body). Nondistinguished variables that occur in the body at most once are called unbound variables, all other variables are called bound variables. Unbound variables may be denoted as the anonymous variable '\_'. A Horn clause  $\gamma$  is safe if all distinguished variables occur in the body. We consider only safe Horn clauses in our work, since we fully translate ontologies to safe Horn clauses. Alternatively a Horn clause can be written as  $\beta_0 \vee \neg\beta_1 \vee \dots \vee \neg\beta_n$ . A Horn clause  $\gamma_1$  subsumes another clause  $\gamma_2$  if and only if there is some most general unifier (MGU)  $\mu$  of clauses  $\gamma_1$  and  $\gamma_2$  such that  $\mu(\gamma_1) \subseteq \gamma_2$ .

A **logic program** LP is a finite set of Horn clauses. A logic program LP is considered a Datalog program if no functional terms are present in LP. In a logic

---

program we distinguish two parts:

- an **intensional** part, composed of Horn rules of the form  $\beta_0 \leftarrow \beta_1 \wedge \dots \wedge \beta_n$ , which corresponds to a TBox (terminological box) in an ontological context.
- an **extensional** part, composed of facts of the form  $\beta$  (an atom), which corresponds to an ABox (assertional box) in an ontological context.

Datalog distinguishes between intensional and extensional symbols. This classifies the set of predicates that appear in the head of some clause in the intensional part and the set of predicates in the extensional. It is sometimes assumed that both sets of predicates are disjoint. We do not make that assumption here, we consider that some predicates may have an extensional and an intensional description.

A set of mappings  $\mathcal{M}$  maps predicates<sup>1</sup> in the ontology  $\mathcal{O}$  (usually only a TBox  $\mathcal{T}$ ) with data in the database  $D$ , providing the extensional part for these predicates. Thus an OBDA system  $\mathcal{J}$  is defined by a triple  $\langle \mathcal{T}, \mathcal{M}, D \rangle$  where:

- The TBox  $\mathcal{T}$  provides inference capabilities on the data. In the future we will use  $\mathcal{T}$  to refer to the TBox as a set of ontological axioms in description logics and  $\Sigma$  to refer to a set of Horn clauses in first order logic (FOL), as corresponding in table 2.2 (page 22). The unary and binary predicates in  $\Sigma$  will correspond to concepts and properties in  $\mathcal{T}$ , respectively.
- The data source  $D$  contains the data that is accessed with the OBDA system. The predicates in  $D$  will generally be different from those present in  $\mathcal{T}$ , for example they may have any arity. Both sets of predicates will be mapped with  $\mathcal{M}$ .
- Finally,  $\mathcal{M}$  provides the mappings from the predicates in  $\Sigma$  to the predicates in  $D$ , possibly performing some transformations on the data. Mappings are characterised in section 2.5.1. Both the mappings  $\mathcal{M}$  and the data  $D$  together form an ABox  $\mathcal{A}$ . This ABox can be materialised or virtual, depending respectively on whether the ABox assertions are generated and stored (e.g. in a triplestore) or are simulated for query answering purposes.

---

<sup>1</sup>Please note that we will refer to concepts and properties in a TBox as “predicates”.

## 2. BACKGROUND AND STATE OF THE ART

---

In some other cases in the state of the art, an OBDA system  $\mathcal{J}$  is presented as an abstraction of an ontology  $\mathcal{O}$ , therefore composed of a TBox and an ABox as usual  $\langle \mathcal{T}_{\mathcal{O}}, \mathcal{A}_{\mathcal{O}} \rangle$ . In the case of OBDA systems, the ABox  $\mathcal{A}_{\mathcal{O}}$  is virtual (i.e.  $\langle \mathcal{M}, D \rangle$ ) and composed by the data source  $D$  and the set of mappings  $\mathcal{M}$ , between the predicates in  $\mathcal{T}_{\mathcal{O}}$  and  $D$ , i.e.  $\mathcal{A} = \langle \mathcal{M}, D \rangle$ . Both perspectives are equivalent in practice, therefore we can define the OBDA system  $\mathcal{J}$  as  $\langle \Sigma, \mathcal{M}, D \rangle$ ,  $\langle \Sigma, \langle \mathcal{M}, D \rangle \rangle$ ,  $\langle \mathcal{T}_{\mathcal{O}}, \langle \mathcal{M}, D \rangle \rangle$ , or  $\langle \mathcal{T}_{\mathcal{O}}, \langle \mathcal{M}, D \rangle \rangle$ . Where the differences between  $\mathcal{T}_{\mathcal{O}}$  and  $\Sigma$  are syntactical, i.e.  $\mathcal{T}_{\mathcal{O}}$  is a set of terminological DL axioms and  $\Sigma$  is the semantically equivalent set of Horn clauses. In some cases, it will be relevant to consider the mappings, and the first perspective will be used. In some other cases, usual techniques for ontology reasoning will be useful for OBDA systems, and descriptions using the last perspective will be more clear and conventional. Therefore, we will use both characterisations interchangeably, as they are equivalent in all relevant aspects, and one or another may be preferable depending on the context.

A **conjunctive query**,  $q_c$  in this case, is a Horn clause where the head predicate  $pred(head(q_c))$  does not clash with any predicate in the Datalog program that is to be queried. A union of conjunctive queries (UCQ) is a set of conjunctive queries  $q_{UCQ}$  such that every query in it has the same head predicate  $\forall q_i, q_j \in q_{UCQ}. pred(head(q_i)) = pred(head(q_j))$ . A Datalog query is the union of a given UCQ  $q_{UCQ}$  with a given Datalog program  $\Sigma$  defined on  $\mathcal{L}(\mathcal{P}, \mathcal{F}, \mathcal{C}, \mathcal{V})$ . In our case  $\Sigma$  will be provided by the OBDA system  $\langle \Sigma, \mathcal{M}, D \rangle \simeq \langle \mathcal{T}, \mathcal{M}, D \rangle$ . To avoid the previously mentioned clashes, we will consider that  $pred(head(q_{UCQ})) \notin \mathcal{P}$ . Given a query  $q$  (either a conjunctive query  $q_c$  or a UCQ  $q_{UCQ}$ ), we will denote the atom built with the predicate  $pred(head(q))$  applied to the n-tuple of constants  $\vec{\alpha}$  as  $q(\vec{\alpha})$  and equivalently as  $q(\alpha_1, \dots, \alpha_n)$ .

Given a query  $q$  over an OBDA system  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$  the certain answers of  $q$  with respect to  $\mathcal{J}$ , denoted as  $\Phi_{\mathcal{J}}^q$ , is the set containing exactly every tuple  $\vec{\alpha}$  of constants in  $\mathcal{J}$  such that  $\Sigma \cup \mathcal{M} \cup D \cup q \models q(\vec{\alpha})$ . We will use  $\Sigma$  and  $q$  to obtain a new query<sup>1</sup>  $q_{\Sigma}$  such that  $q_{\Sigma} \cup \mathcal{M} \cup D \models q(\vec{\alpha})$ , i.e. this new query obtains the same certain answers when evaluated on the OBDA system without the TBox. This means that we will be interested in the transformations  $\Delta$  that can be performed

---

<sup>1</sup> In case all clauses in  $q_{\Sigma}$  have the same head predicate we will have a UCQ.

---

on  $\Sigma \cup q$  to derive  $\Sigma \cup q \xrightarrow{\Delta} q_\Sigma$  while preserving the satisfiability and thus the possible tuples  $\vec{\alpha}$ , the answers for the query  $q$  on the system  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$ , so that  $\Phi_{\langle \Sigma, \mathcal{M}, D \rangle}^q = \Phi_{\langle \mathcal{M}, D \rangle}^{q_\Sigma}$ .

### 2.1.1 Subsumption

Especially noteworthy is the concept of subsumption, both among clauses and among atoms in a clause.

Given a clause  $\beta_0 \leftarrow \beta_1 \wedge \dots \wedge \beta_n$ , an atom  $\beta_a$  subsumes another atom  $\beta_b$  if the predicate  $p_a$  in  $\beta_a$  equals (in name and arity) to the predicate  $p_b$  in  $\beta_b$  and there is a unification  $\mu$  from the free variables in  $\beta_a$  to the terms in  $\beta_b$  such that  $\mu(\beta_a) = \beta_b$ . From a semantic perspective, an atom  $\beta_a$  subsumes another atom  $\beta_b$  in a clause  $\gamma_i$  such that  $\beta_a, \beta_b \in \text{body}(\gamma_i)$  if for a clause  $\gamma_j$  such that  $\text{head}(\gamma_i) = \text{head}(\gamma_j)$ ,  $\text{body}(\gamma_i) \setminus \{\beta_a\} = \text{body}(\gamma_j)$  and  $\gamma_j \models \gamma_i$ . Intuitively, an atom subsumes another if the subsumed atom can be removed from the clause without making it more general.

Note that any term that is not a free variable (e.g. existential variables) cannot be unified and must remain unaltered. When an atom  $\beta_a$  subsumes some other atom  $\beta_b$  in a conjunction (e.g. a clause body) the subsuming atom can be removed as the subsumed atom is more specific. The operation of removing the subsuming atoms in a clause is usually named as *clause condensation*.

A clause  $\gamma_a$  *subsumes* some other clause  $\gamma_b$ , denoted by  $\gamma_a \succeq_s \gamma_b$ , if and only if  $\gamma_a \models \gamma_b$ .

Clauses form disjunctions in Datalog programs, therefore, when a pair of subsuming and subsumed clauses is found, the clause that can be removed is the subsumed clause, which is more specific than the subsuming clause.

### 2.1.2 Resolution

Resolution is a refutationally complete theorem proving method [Bachmair and Ganzinger, 2001; Robinson, 1965], this means that a contradiction (i.e., the empty clause) can be deduced from any unsatisfiable set of clauses. The search for a contradiction proceeds by saturating the given clause set, that is, systematically (and exhaustively) applying all inference rules.

## 2. BACKGROUND AND STATE OF THE ART

---

In our work we use resolution to rewrite queries as sets of clauses by proving and adding new clauses to these sets. If resolution is sound then we can guarantee, by the definition of soundness, that for any provable set of clauses  $\Gamma$  if  $\Sigma \cup q \vdash \Gamma$  then  $\forall \vec{\alpha}. (\Sigma \cup q \cup \Gamma \models q(\vec{\alpha})) \iff (\Sigma \cup q \models q(\vec{\alpha}))$ . If the resolution is complete then we can also guarantee, by the definition of completeness, that for any  $\Gamma^{(1)}, \dots, \Gamma^{(n)}$  derivations of  $\Sigma \cup q$  it holds that:

$$\forall i \in \{1, \dots, n\}. (\Gamma^{(i)} \models q(\vec{\alpha})) \Rightarrow (\Gamma^{(i)} \vdash q(\vec{\alpha}))$$

As far as these transformations generate a set of  $\Gamma^{(i)}$  where every  $\Gamma^{(i)} \equiv \Sigma \cup q$ , by the definition of logical equivalence, we will have that  $\Gamma^{(i)} \models q(\vec{\alpha})$ . This means that in the process of query rewriting we can also use any derivation that allows obtaining  $\Gamma^{(i+1)}$  from  $\Gamma^{(i)}$  such that  $\Gamma^{(i+1)} \equiv \Gamma^{(i)}$ . In fact, for query answering purposes we can use any transformation  $\delta$  such that  $\Gamma^{(i)} \xrightarrow{\delta} \Gamma^{(j)}$  if and only if it preserves the answers obtained  $\Phi_{\Gamma^{(i)}}^q = \Phi_{\Gamma^{(j)}}^q$  for a given query  $q$ , since this happens if and only if  $\Gamma^{(i)} \models q(\vec{\alpha})$  and  $\Gamma^{(j)} \models q(\vec{\alpha})$  for the same set of  $\vec{\alpha}$  by the definition of certain answers, which happens (as previously explained) if and only if  $\Gamma^{(i)} \vdash q(\vec{\alpha})$  and  $\Gamma^{(j)} \vdash q(\vec{\alpha})$  for the same set of constant values  $\vec{\alpha}$  for complete and correct resolution methods.

In our case we will use resolution with free selection (RFS). The essence of resolution calculus can be described with two inference rules [Bachmair and Ganzinger, 2001]:

$$\text{(Binary) Resolution: } \frac{\gamma_C \vee \gamma_A \quad \gamma_D \vee \neg \gamma_B}{(\gamma_C \vee \gamma_D)\mu}$$

$$\text{(Positive) Factoring: } \frac{\gamma_C \vee \gamma_A \vee \gamma_B}{\mu(\gamma_C \vee \gamma_A)}$$

Where  $\mu$  is in both cases the most general unifier (MGU) of atomic formulas  $\gamma_A$  and  $\gamma_B$ . Both rules can be combined into one:

$$\text{Binary resolution with factoring: } \frac{\gamma \vee \gamma_A \vee \dots \vee \gamma_A \quad \gamma_D \vee \neg \gamma_A}{\mu(\gamma \vee \gamma_D)}$$

As can be seen in resolution rules two atoms are unified, one on each clause. In the case of resolution with free selection, for two atoms to be unified they have

---

to be selected by some selection function. A selection function for Horn clauses selects either the head of a clause or a non empty set of body atoms. For atoms to be selected they have to meet certain criteria. This allows prioritizing some inferences with respect to others, by selecting only some atoms, or types of atoms and giving a higher priority to the inferences that involve those atoms. Therefore resolution with Horn clauses takes the form:

$$\frac{\beta_A \leftarrow \beta_1 \wedge \dots \wedge \beta_{i-1} \wedge \underline{\beta_i} \wedge \beta_{i+1} \wedge \dots \wedge \beta_n \quad \underline{\gamma} \leftarrow \beta_{D_1} \wedge \dots \wedge \beta_{D_n}}{\mu(\beta_A \leftarrow \beta_1 \wedge \dots \wedge \beta_{i-1} \wedge \beta_{D_1} \wedge \dots \wedge \beta_{D_n} \wedge \beta_{i+1} \wedge \dots \wedge \beta_n)}$$

Where  $\mu$  is the MGU of atomic formulas  $\beta_i$  and  $\gamma$  and the underlined atoms are the ones selected and resolved. The clause for which a body atom is selected is considered the main premise and the clause for which the head is selected is considered the side premise.

Resolution with free selection has been proved to be correct and complete for Horn clauses [Lynch, 1997].

### 2.1.3 Cycles and recursion

A particularly relevant aspect to consider in query answering in description logics is the possibility of recursion and cycles [Grau et al., 2012]. This recursion and cycles may be statically detected and analysed using the dependency graph, described as follows.

A *dependency graph* for a Datalog program has a vertex for each predicate  $p$  in the program and an edge from  $p_i$  to  $p_j$  for each clause containing  $p_i$  in the body and  $p_j$  in the head, for every  $p_i, p_j$  in the Datalog program. The dependency graph will be used for the static analysis of a Datalog program, for example to check the reachability of some predicate from another predicate. A predicate  $p_i$  is *reachable* from a predicate  $p_j$  if and only if there is an edge from  $p_i$  to  $p_j$  in the dependency graph, or there is some predicate  $p_k$  such that there is an edge from  $p_k$  to  $p_j$  in the dependency graph and  $p_i$  is reachable from  $p_k$ .

This static analysis can reveal some infinite recursion in a process associated with the corresponding Datalog program. One of the common tasks in query

## 2. BACKGROUND AND STATE OF THE ART

---

answering is the *chase procedure* [Johnson, 1975; Maier et al., 1979]. The chase procedure applies forward chaining resolution, i.e. given a set of clauses and a set of assertions, the bodies of the clauses are unified with the assertions producing new assertions. It is easy to see that this can lead to an infinite process when the clauses contain existentially quantified variables in the head.

For query rewriting purposes we can restrict our attention to Datalog programs that do not contain existentially quantified variables nor constants in the head. We can also focus on backward chaining resolution, as seen in section 2.1.2, where clauses representing the query are unified among themselves, generating a rewritten query finally be unified with the assertions for query answering. However, in this context infinite recursion is also possible.

A Datalog program is said to be *recursive* when there is some cycle in its dependency graph, and nonrecursive otherwise. The recursion in a cycle  $\theta$  may lead to termination of the chase procedure or not. In general, deciding the termination of the recursion in a cycle is an undecidable problem.

To cope with this problem, some *acyclicity conditions* are often defined. These conditions are usually sufficient but not necessary to ensure that there is no infinite recursion. Given a cycle  $\theta$  in the dependency graph, one of such conditions is whether all the variables in the vertexes of  $\theta$  are distinguished, according to the edges of  $\theta$ . This means that for each edge in  $\theta$  it corresponds to a clause  $\gamma$  containing some predicate  $p$  in the body and all the variables to which  $p$  applies are distinguished in the head of  $\gamma$ . If all these variables are distinguished, then the recursion terminates, i.e. the recursion is *safe*. If there is some variable that is not distinguished, then there may be an infinite recursion, i.e. the recursion is not safe.

An example of safe recursion can be created with the axioms  $P \sqsubseteq S$  and  $S \sqsubseteq P^-$ , being  $S$  and  $P$  two object properties. These two axioms generate the following two Datalog clauses:  $S(x,y) \leftarrow P(x,y)$  and  $P(x,y) \leftarrow S(y,x)$ . Here we can see that there are two variables  $x$  and  $y$  that are distinguished in both edges, i.e. for all the clauses in which they appear in the body, they are also in the head. Therefore, the recursion is safe.

Given an object property  $P$  and a concept  $A$ , an example of recursion that is not safe is created with the axiom  $\exists P.A \sqsubseteq A$ . This axiom translates to the clause



---

$A(x) \leftarrow P(x, y), A(y)$ . Here we can see that a single clause is enough to cause recursion and that the edge that is generated from and to this clause through  $A$  contains variables that are not distinguished (exactly  $y$ ). For this reason, the recursion is not safe. In this particular example an infinite set of clauses can be derived taking the form of  $A(x) \leftarrow P(x, y), P(y, z), A(z)$ , then  $A(x) \leftarrow P(x, y), P(y, z), P(z, v), A(v)$ , and so on.

## 2.2 Query rewriting in query answering

As discussed in the introduction, Ontology Based Data Access (OBDA) consists on superimposing a conceptual layer as a view to an underlying information system, which abstracts away from how that information is maintained in the data layer and provides inference capabilities [Calvanese et al., 2007b].

It should be noted that query rewriting is previous to OBDA and that the superimposition of a view to underlying information systems has been traditionally done in information integration scenarios. These views may be defined as global as view or local as view rules (as we will see in more detail in section 2.5) or may be the intensional part of a deductive database in Datalog. OBDA inherits aspects of both.

Among the classical approaches for data integration we can consider algorithms that use relational schemata. These approaches are usually based on Datalog and two good examples are Razor [Friedman and Weld, 1997] and Internet Softbot [Etzioni and Weld, 1994]. We can also find algorithms that use extensions of Datalog, such as the inverse rules algorithm used in Infomaster [Genesereth et al., 1997]. In Information Manifold [Levy et al., 1996a] some predicates in the rules may be concepts defined by using description logics constructors. The “query reformulation” done with the bucket algorithm [Levy, 2000] considers the need of recursive Datalog queries to access sources that can only be accessed with certain patterns. PICSEL with the language CARIN [Rousset and Reynaud, 2003] uses the description logic  $\mathcal{ALN}$  in its description logic component, which is complemented by a rule component more similar to Datalog. At that time and with that state of the art several authors pointed at the use of a rich representation for the global (or mediated) schema by using description logics

## 2. BACKGROUND AND STATE OF THE ART

---

[Arens et al., 1996; Catarci and Lenzerini, 1993; Goasdoué and Reynaud, 1999; Levy et al., 1996b]. The way to use description logics at that time consisted primarily in combining DL axioms with Horn rules [Donini et al., 1991; Levy and Rousset, 1998; MacGregor, 1994]. The combination of Datalog or Horn rules with DL axioms provided a greater expressiveness but at the cost of a greater computational complexity as well [Cadoli et al., 1997].

Other solutions took an object oriented approach. This is the case of TSIMMIS [Garcia-Molina et al., 1997; Papakonstantinou et al., 1995], which uses the object-oriented language OEM for the description of the mediated schema and the views while the queries are represented with OEM-QL. MOMIS [Beneventano et al., 2000] uses the DL ODL-I3 for the description of the schemata of the sources to integrate.

The use of objects, where the focus is set on the classes, objects, members and attributes, paved to some extent the way for a smooth transition to description logics, where the focus is set on concepts, individuals, roles and properties. SIMS [Arens et al., 1993, 1998] and OBSERVER [Mena et al., 2000] use description logics for the definition of the mediated schema, the views and the queries. SIMS uses LOOM [MacGregor and Bates, 1987] while OBSERVER is based on CLASSIC [Borgida et al., 1989]. In both latter systems the problem of query rewriting is handled as a planning problem. This planning problem is solved by rewriting the query plan in the Planning-by-Rewriting approach [Ambite et al., 2000]. This rewriting of the query plan should not be confused with the rewriting of the query that we will describe soon.

We can find a general classification of the types of ontology based data integration systems in [Wache et al., 2001] as well as a general perspective of the problem. We can see that the problem can be divided into several smaller problems, each of them addressed in one of the several steps used in the integration of the information and its access as a materialized or virtual ontology. One of these problems is the so-called *impedance mismatch*. The impedance mismatch problem refers to the mismatch between the way in which data is (and can be) represented in a relational database, and the way in which the corresponding information is rendered in an ontology [Poggi et al., 2008]. The semantic upgrade of data deals with this problem so that the integration can happen naturally in the semantic

---

level. At the semantic level, information integration approaches may allow the access to distributed or federated data sources [Buil-Aranda et al., 2013; Quilitz and Leser, 2008]. Additionally, at the semantic level, the reasoning corresponding to the rich semantics in description logics can vary substantially between approaches, for example by materializing the inferences [Kontchakov et al., 2010; Rodríguez-Muro and Calvanese, 2012] or by rewriting the queries, as we see next.

The process of query rewriting in OBDA systems consists in using ontologies to transform ontology-based queries (e.g., written in SPARQL or in ad-hoc query languages) into queries that the underlying data sources are able to process (e.g., written in SQL, Datalog, etc.). This allows obtaining answers for the original queries as if they had been posed to an ontology that contains instances, where such instances are obtained from the data available in those data sources. Query rewriting gained attention due to the first-order rewritability property. This property intuitively means that the queries posed to an OBDA system can be rewritten into a first order language (FOL), e.g. SQL. These rewritten queries can then be used with systems that support those languages, e.g. relational databases. We see next the formal definition for this property.

Given an ontology language  $\mathcal{L}$  and a query language  $\mathcal{Q}$  for a data source  $D$ , we say  $\mathcal{L}$  is **first-order rewritable** if any query  $q$  expressed over an ontology  $\Sigma$  written in the ontology language  $\mathcal{L}$  can be rewritten into a perfect rewriting [Calvanese et al., 2000]  $q_\Sigma$ , written according to the query language  $\mathcal{Q}$ , such that when  $q_\Sigma$  is evaluated over the data source  $D$  the results obtained —  $\Phi_D^{q_\Sigma}$  — are the same as evaluating  $q$  over  $\Sigma$  and  $D$  —  $\Phi_{\Sigma,D}^q$ . Due to this rewritability property it is possible to separate the complexity associated with the inference required to rewrite  $q_\Sigma$  from  $q$  and  $\Sigma$  and the complexity derived from obtaining the answers to  $q_\Sigma$  from  $D$ . Several description logic languages have been identified among the languages that present this property for SQL or equivalently expressive query languages. Among the languages that present the first-order rewritability property, the ones with a special relevance in our context are: the *DL-Lite* family [Calvanese et al., 2007a], which includes *DL-Lite<sub>core</sub>*, *DL-Lite<sub>f</sub>* and *DL-Lite<sub>R</sub>*; the *QL* profile of *OWL2* [Cuenca Grau et al., 2008]; and some families in *Datalog $\pm$*  [Calì et al., 2011]. These logics are detailed further in section 2.3 along with the *ELHIO $^\top$*  family [Pérez-Urbina et al., 2009], which is more expressive and not first

## 2. BACKGROUND AND STATE OF THE ART

---

order rewritable, but has been used in query rewriting.

Nowadays, in addition to information integration scenarios, OBDA is interesting in a broader range of scenarios where its applicability has been boosted thanks to the tractability provided by the first-order language reducibility [Calvanese et al., 2007a] or first-order rewritability property [Gottlob et al., 2011], identified in some logic families. This property improves the capabilities of query answering through the use of query rewriting techniques [Calvanese et al., 2000]. In the present work we focus on query rewriting approaches and techniques, leaving aside previously described approaches that may in general take an approach closer to expert systems, with the knowledge provided by an ontology or a set of ontologies and a set of rules, or take a different approach to OBDA by focusing on providing access to documents, discovery of data sources, ontology population, etc.

We will not consider either other systems that may be more similar in the purpose and approach (e.g. using *DL-Lite*) but that work on a different context, for instance using materialization approaches, either on the data [Kontchakov et al., 2010] or the mappings [Rodríguez-Muro and Calvanese, 2012].

We will focus on query rewriting without modifying the original data sources and mappings, and as we will see, we will focus on the description logic  $\mathcal{ELHJO}$ . We will compare our approach to approaches using this logic or similar logics, under similar assumptions in terms of properties of the sources and the query language. This description logic is not first order rewritable, but the computational complexity of the rewriting in this case is still tractable and it is possible to implement a “pay-as-you-go” approach [Pérez-Urbina et al., 2009]. This means that depending on the ontology and the query, if the rewriting does not need to be recursive, then a UCQ or a non-recursive Datalog program can be produced. Therefore we have that:

- for ontologies in usual FOL-reducible logics the output will be a non-recursive Datalog program or a UCQ, depending on the selection made by the user.
- for any rewriting that requires recursive Datalog, approaches that rely on a non-recursive first-order rewriting would produce an incomplete answer.

In such cases, the rewritten query must be a recursive Datalog query.

## 2.3 OBDA-related logics

As aforementioned, OBDA-query rewriting has paid special attention to logics that are first-order rewritable [Calvanese et al., 2007a; Gottlob et al., 2011]. In this section we mention those which have a special relevance for the related approaches in the state of the art, and summarise them in table 2.1. In the examples,  $a$  refers to a constant (individual),  $B_i$  refer to basic concepts (classes) and  $R$  refers to roles (properties).

axiom <sup>1</sup> \ logic	$DL-Lite_{core}$	$DL-Lite_F$	$DL-Lite_R$	Rapid's	$\mathcal{ELH}Q^\neg$	Datalog <sup>±2</sup>	Horn-SHJQ
$B_1 \sqsubseteq B_2, B_1 \sqsubseteq \neg B_2^3$	✓	✓	✓	✓	✓	✓	✓
$\geq 2R_1 \sqsubseteq \perp$	✗	✓	✗	✗	✗	✗	✓
$R_1 \sqsubseteq R_2, R_1 \sqsubseteq \neg R_2$	✗	✗	✓	✓	✓	✓	✓
$B_1 \sqsubseteq \exists R_1, \exists R_1 \sqsubseteq B_1$	✗	✗	✓	✓	✓	✓	✓
$B_1 \sqsubseteq \exists R_1.B_2$	✗	✗	✗	✓	✓	✓	✓
$\exists R_1.B_1 \sqsubseteq B_2$	✗	✗	✗	✗	✓	✗	✓
$B_1 \sqcap B_2 \sqsubseteq B_3$	✗	✗	✗	✗	✓	✗	✓
$\{a\} \sqsubseteq B, B \sqsubseteq \{a\}, B(a)$	✗	✗	✗	✗	✓	✗	✓
$n$ -ary predicates	✗	✗	✗	✗	✗	✓	✗
$trans(R_1)$	✗	✗	✗	✗	✗	✗	✓
$B_1 \sqsubseteq \forall R_1.B_2, B_1 \sqsubseteq \leq 1R_1.B_2$	✗	✗	✗	✗	✗	✗	✓

Table 2.1: Overview of the main logics in the state of the art.

- The  $DL-Lite$  family is formed by  $DL-Lite_{core}$  and its extensions, being the main ones  $DL-Lite_{\mathcal{R}}$  and  $DL-Lite_{\mathcal{F}}$ . In  $DL-Lite_{core}$  concept inclusions are restricted to  $B_1 \sqsubseteq B_2$  and  $B_1 \sqsubseteq \neg B_2$ .  $DL-Lite_{\mathcal{R}}$  includes subsumption

<sup>1</sup>here,  $\forall i, j$ ,  $B_i$  represents a basic concept and  $R_j$  represents a role that may be basic or inverted.

<sup>2</sup>as implemented in Nyaya

<sup>3</sup>note that some systems implementing negation assume a consistent ABox

## 2. BACKGROUND AND STATE OF THE ART

---

(ISA) and disjointness assertions between roles and *DL-Lite<sub>f</sub>* includes functionality restrictions on roles. These logics are first-order reducible with a tractable complexity [Calvanese et al., 2007a].

- The *OWL2 QL* profile was inspired by the *DL-Lite* family and designed to keep the complexity of rewriting low, considering first-order rewritability. Among the main differences with *DL-Lite* we can consider the status of the unique name assumption (UNA). UNA is adopted in *DL-Lite* but not in *OWL*, which uses instead the explicit relations `sameAs` and `differentFrom` to describe that two symbols refer to the same or different entities, respectively. *OWL2 QL* also lacks constructs that would conflict with UNA and cause a greater complexity, like number restrictions, functionality constraints and keys. Among the constructs in *OWL 2* not supported in *DL-Lite* we can remark nominals, that is concepts of the form  $\{a\}$ . For a more extensive comparison check [Artale et al., 2009].
- The  $\mathcal{ELHIJ}^\cap$  logic [Pérez-Urbina et al., 2009] is more expressive than the previous ones. It extends the expressiveness of *DL-Lite<sub>R</sub>* by including basic concepts of the form  $\{a\}$ ,  $\top$ , and  $B_1 \sqcap B_2$ , as well as axioms of the form  $\exists R.B_1 \sqsubseteq B_2$ . Depending on the query and the expressiveness in the ontology, rewriting to non-recursive Datalog may result in some loss of information, thus some queries should be rewritten to recursive Datalog when considering  $\mathcal{ELHIJ}^\cap$  ontologies, what means that it is not first-order rewritable for languages like SQL. In spite of that, the computational complexity of the rewriting process remains tractable (PTIME-complete).
- Some families in  $\text{Datalog}^\pm$  preserve the property of first-order rewritability to SQL equivalent languages while offering a greater expressiveness for rewritings to SQL or non-recursive Datalog, mainly because of the fact that  $\text{Datalog}^\pm$  predicates are n-ary. Some of the Datalog paradigms that ensure decidability are chase termination, guardedness or stickiness, extended to weak-stickiness by Calì et al. [Calì et al., 2010]. Despite this greater expressiveness it should be noted that  $\mathcal{ELHIJ}^\cap$  is not covered by these  $\text{Datalog}^\pm$  families. For instance, some rewritings generate recursive Datalog, some-

---

thing that is avoided by these properties in  $\text{Datalog}^\pm$ .

- Finally,  $\text{Horn-}\mathcal{SHIQ}$  includes the role hierarchies and inverse roles as  $\mathcal{ELHIQ}$ . It does also include universal restrictions and transitive roles ( $\mathcal{S}$ ) axioms of the form  $A \sqsubseteq \forall R.B$  and  $\text{trans}(R)$ . This logic does also include qualified cardinality restrictions ( $\mathcal{Q}$ ) axioms of the form  $A \sqsubseteq \leq 1R.B$ . The Horn prefix refers to the Horn fragment of  $\mathcal{SHIQ}$ ; this means that the axioms in this fragment can be converted to Horn clauses.

### 2.3.1 $\mathcal{ELHIQ}$

For greater clarity, we specify  $\mathcal{ELHIQ}$  into detail. In  $\mathcal{ELHIQ}$ , concept ( $C$ ) and role ( $R$ ) expressions are formed according to the following syntax (where  $A$  denotes a concept name,  $P$  denotes a role name, and  $a$  denotes an individual name):

$$\begin{aligned} C &::= A \mid C_1 \sqcap C_2 \mid \exists R.C \mid \{a\} \\ R &::= P \mid P^- \end{aligned}$$

An  $\mathcal{ELHIQ}$  axiom is an expression of the form  $C_1 \sqsubseteq C_2$  or  $R_1 \sqsubseteq R_2$  where  $C_1, C_2$  are concept expressions and  $R_1, R_2$  are role expressions.

Usually, the axioms in these logics will be converted to  $\text{Datalog}$  or in some cases  $\text{FOL}$ , for example in the case of  $\mathcal{ELHIQ}$  the correspondence between clauses and axioms would be the same as in table 2.2 as described previously [Pérez-Urbina et al., 2010] for the query rewriting system  $\text{REQUIEM}$ .

Additionally to  $\mathcal{ELHIQ}$ , we will refer in some cases to  $\mathcal{ELHIQ}^-$ , as this is the logic usually mentioned for the  $\text{REQUIEM}$  system. The difference between  $\mathcal{ELHIQ}$  and  $\mathcal{ELHIQ}^-$  are the negative inclusions, which are relevant for some operations, e.g. checking the consistency of a OBDA system. However, these negative inclusions are not considered for query rewriting purposes neither in the  $\text{REQUIEM}$  system nor in the present work. For practical considerations, in the current scope of query rewriting, there is no difference between both logics. The  $\mathcal{ELHIQ}^-$  expressiveness is mentioned for coherence with the existing literature.

## 2. BACKGROUND AND STATE OF THE ART

---

$A(a)$	$A(a), \{a\} \sqsubseteq A$
$P(a, b)$	$P(a, b)$
$x \approx a \leftarrow A(x)$	$A \sqsubseteq \{a\}$
$A_2(x) \leftarrow A_1(x)$	$A_1 \sqsubseteq A_2$
$A_3(x) \leftarrow A_1(x) \wedge A_2(x)$	$A_1 \sqcap A_2 \sqsubseteq A_3$
$P(x, f(x)) \leftarrow A(x)$	$A \sqsubseteq \exists P$
$P(x, f(x)) \leftarrow A_1(x), A_2(f(x)) \leftarrow A_1(x)$	$A_1 \sqsubseteq \exists P.A_2$
$P(f(x), x) \leftarrow A$	$A \sqsubseteq \exists P^-$
$P(f(x), x) \leftarrow A_1(x), A_2(f(x)) \leftarrow A_1(x)$	$A_1 \sqsubseteq \exists P^-.A_2$
$A(x) \leftarrow P(x, y)$	$\exists P \sqsubseteq A$
$A_2(x) \leftarrow P(x, y), A_1(y)$	$\exists P.A_1 \sqsubseteq A_2$
$A(x) \leftarrow P(y, x)$	$\exists P^- \sqsubseteq A$
$A_2(x) \leftarrow P(y, x), A_1(y)$	$\exists P^-.A_1 \sqsubseteq A_2$
$S(x, y) \leftarrow P(x, y)$	$P \sqsubseteq S, \quad P^- \sqsubseteq S^-$
$S(x, y) \leftarrow P(y, x)$	$P \sqsubseteq S^-, \quad P \sqsubseteq S^-$

Table 2.2: Correspondence between  $\mathcal{ELHIJO}$  axioms and FOL clauses. Note that the Skolem functions generated are local to each axiom.

### 2.4 OBDA implementations

As we have seen in section 2.2, the area of OBDA has evolved through a considerable period of time, spawning approaches with many different characteristics and purposes. In that context, with the purpose of focusing on the most relevant approaches for the present work (summarised in table 2.3), we can consider the starting point of the current generation of OBDA-query rewriting systems the PerfectRef algorithm [Calvanese et al., 2007a], which is qualitatively different from previous approaches due to the FOL-rewritability property. This algorithm is implemented in **Quonto** and also made available as open source by Pérez-Urbina [Pérez-Urbina et al., 2009]. This approach accepts ontologies written in the *DL-Lite* family (*DL-Lite<sub>core</sub>*, *DL-Lite<sub>f</sub>* and *DL-Lite<sub>R</sub>*). More precisely the constraints of *DL-Lite<sub>f</sub>* are responsibility of the ABox, and therefore a *DL-Lite<sub>f</sub>* ontology is equivalent to a *DL-Lite<sub>core</sub>* ontology. *DL-Lite<sub>R</sub>* is the most expressive of the three logics. The PerfectRef algorithm is defined for this expressiveness and usable for the other two. This approach generates a UCQ as a result of the rewriting process. It was the first of a series and would inspire many others,



---

usually generating UCQs too while optimising the process, for instance by applying optimisations based on query decomposition or identification of connected components.

We can continue presenting the rest of the approaches in chronological order<sup>1</sup>. The **RQR algorithm** [Pérez-Urbina et al., 2009] accepts  $\mathcal{ELHI}^{\neg}$  ontologies and generates a rewriting using resolution with free selection [Bachmair and Ganzinger, 2001]. RQR is implemented in REQUIEM, which reduces the number of useless factorizations, queries generated and processing time through several optimisations, the main one being the introduction of Skolem functions when an existential quantification was converted into the head of a clause, which was handled in the previous approach as a nameless variable. The output generated by this approach is again a UCQ. Nevertheless, resolution in REQUIEM is splitted into two steps, the first one is saturation, which generates a (possibly recursive) Datalog program; the second one is unfolding, which unfolds this Datalog program to generate a UCQ. Therefore REQUIEM may produce a Datalog program for the output by simply skipping the latter stage. If the ontology includes recursion the UCQ cannot be complete. In this case only the recursive Datalog program is the complete option and the unfolding generates a recursive Datalog program. The number of different head predicates in this Datalog program is reduced in the unfolding stage, in this case to reduce the amount of information loss in the case that clauses with head predicates different from the query predicate are dropped by the system that receives the output.

The previous approaches generate a large number of queries in the UCQ, as the generation of this UCQ from Datalog presents a combinatorial blowup that depends on the length of the query. **Presto** [Rosati and Almatelli, 2010] addresses this problem on *DL-Lite<sub>R</sub>* (that is, not including expressions of the form  $\exists R.B$ , what makes the search for *most-general subsumees* computationally tractable). *Most-general subsumees* are used to remove existential join variables, to then remove unbound variables and redundant atoms. This way the query is recursively factorized and splitted, depending on the existential joins and the connectivity

---

<sup>1</sup>The reader should be aware that some of the approaches count with several publications that intertwine through time, but we have normally selected the time of the first publication for this ordering.

## 2. BACKGROUND AND STATE OF THE ART

---

in the query. Presto obtains results that are several orders of magnitude faster, in the query rewriting process, than previous approaches. Depending on the ontology and the query, the result is also normally briefer in the number of clauses, since it outputs a non-recursive Datalog program instead of a UCQ, hiding the combinatorial explosion that would result from unfolding the program. As a result of the factorization, several parts of the query are rewritten into equivalent subqueries. Checking subsumption among these subqueries is not trivial and not included in Presto, thus, there may be some redundancy.

Stamou [Stamou et al., 2010] takes a different approach in the handling of Skolem functions. This approach has evolved into the **Rapid** algorithm [Chortaras et al., 2011], which handles an expressiveness that falls between REQUIEM and Presto: expressions of the form  $\exists R.B$  are allowed in REQUIEM but not in Presto; and in the case of Rapid they are allowed in the right hand side but not in the left hand side of subsumption axioms. The strategy in Rapid consists in applying two rules alternatively, *query shrinking* and *query unfolding*. *Query shrinking* removes a bound variable by unifying it with a functional term. Skolem functions are internally handled in this resolution rule, so they do not appear after applying it. *Query unfolding* replaces a set of atoms with its unfoldings, preserving the terms in the atoms — no functional terms are used —. This strategy generates less subsumed (redundant) queries and it is possible to restrict the search for subsumed queries among subsets of the queries generated. The output is equivalent to REQUIEM as far as the ontology used is contained in the expressiveness that Rapid can handle.

A similar approach using two different resolution steps (factorization and rewriting) in a stratified strategy is the one implemented in **Nyaya** [Gottlob et al., 2011]. During the factorization step the query is compacted with unifications that preserve the query semantics, and in the rewriting step the query is unfolded. Nyaya has an optional optimisation step that eliminates atoms in the queries preventing the generation of subsumed queries. In the case of Nyaya, the expressiveness is greater than in previous cases by allowing the use of n-ary predicates. However there is no statement about which additional ontological axioms may be covered thanks to this. This expressiveness is reduced to allow further optimisations, in this case the body of the clauses is restricted to those that have

---

only one atom. With this it is possible to identify atoms in the body of a query that are implied by some other atom in the body, what means that they can be eliminated, reducing the size of the query, the UCQ and the required processing. This approach is specially tailored at reducing the size of the UCQ that are generated in the process due to the impact that this property has on the underlying software that receives this UCQ. Depending on the query, this optimisation may provide much smaller queries, in size, width or length, which are respectively, the number of queries in the UCQ, the number of joins to be performed and the number of atoms in the perfect rewriting as explained in [Gottlob et al., 2011].

Another approach that should be mentioned is the one taken by Venetis [Venetis et al., 2011] based on the previously mentioned **PerfectRef** [Calvanese et al., 2007a]. In this approach it is argued that users normally pose a succession of queries to a system refining an initial conjunctive query, by adding or removing atoms in the conjunction. In these cases it is possible to use partial results from previous rewritings in the new rewritings. New rewritings that need only to be computed partially when a similar query has been posed recently to the system and its partial results should still be available in the cache, requiring less time for the generation of the rewriting.

**Prexto** [Rosati, 2012] modifies Presto by considering extensional constraints. With these constraints, and using concept and role disjointness assertions as well as role functionality assertions, Prexto reduces the size of the rewritten query. This has a special relevance in Prexto when compared with Presto, since Presto generated non-recursive Datalog programs and the output in Prexto is a UCQ. Disjointness and role functionality assertions are considered when construcing the Datalog program along with subsumption. More precisely the process where these considerations are done is **DeleteRedundantAtoms**, which deletes the atoms considered as redundant according to these criteria. In the unfolding stage these assertions are considered again, along with the extensional constraints. In this case the process where these constraints are applied is **MinimizeViews**, which reduces the views by removing the parts considered as unnecessary according to these criteria. These considerations allow reducing the combinatorial explosion usual in the unfolding of Datalog programs generated for query rewriting. Prexto introduces the concept of EBox which has a special relevance for the optimisa-

## 2. BACKGROUND AND STATE OF THE ART

---

tions related with mappings. EBox stands for Extensional Box and models the information with a complementary purpose: an EBox does not state which predicates in the ontology are mapped or not. Instead, for two given expressions  $E_i, E_j$  it expresses whether the extension of one is contained into the extension of the other  $E_i \sqsubseteq E_j$ . For example, given some ABox with a set of individuals  $\mathcal{C}$  we can add the axiom  $B_1 \sqsubseteq B_2$  to the EBox for two given basic concepts  $B_1$  and  $B_2$  when the explicit assertions in the ABox satisfy  $\{a_i \mid a_i : B_1\} \subseteq \{a_j \mid a_j : B_2\}$ . For instance, this is the case when an ABox is redundant and the individuals for a subconcept are also specified as individuals for some superconcept.

**Clipper** [Eiter et al., 2012] aims at more expressive logics, more precisely Horn- $\mathcal{SHIQ}$ . This approach can rewrite the ontology including TBox and ABox or only the TBox. The ontology is in this case preprocessed and saturated independently of the query, and the query is rewritten into a UCQ with additional Datalog rules that “complete” the ABox, which is comparable to a Datalog program. Despite the high expressiveness, the times to obtain the rewritings and the size of the rewritings are kept low. This is achieved by aiming at this UCQ with rules output and using a Datalog system that can handle it, such as DLV [Leone et al., 2006] or Clingo [Gebser et al., 2011], instead of obtaining a UCQ as other systems do.

System	Input	Output	Reference
<b>Quonto</b>	DL-Lite <sub>R</sub>	UCQ	Calvanese et al. [2007a]
<b>REQUIEM</b>	$\mathcal{ELHIQ}$	Datalog or UCQ	Pérez-Urbina et al. [2009]
<b>Presto</b>	DL-Lite <sub>R</sub>	Datalog	Rosati and Almatelli [2010]
<b>Rapid</b>	DL-Lite <sub>R</sub> <sup>1</sup>	Datalog or UCQ	Chortaras et al. [2011]
<b>Nyaya</b>	<i>Datalog</i> <sup>±</sup>	UCQ	Gottlob et al. [2011]
<b>Venetis'</b>	DL-Lite <sub>R</sub>	UCQ	Venetis et al. [2011]
<b>Prexto</b>	DL-Lite <sub>R</sub> and EBox	Datalog or UCQ	Rosati [2012]
<b>Clipper</b>	Horn- $\mathcal{SHIQ}$	Datalog	Eiter et al. [2012]
<b>kyrie</b>	$\mathcal{ELHIQ}$	Datalog or UCQ	Mora and Corcho [2013b]
<b>kyrie2</b>	$\mathcal{ELHIQ}$ and EBox	Datalog or UCQ	Mora et al. [2014]

Table 2.3: Main systems for query rewriting in the state of the art

---

<sup>1</sup>Close to OWL2 QL,  $B_1 \sqsubseteq \exists R.B_2$  axioms are supported

---

### 2.4.1 Previous efforts in benchmarking query rewriting systems

Some of the seminal work can be attributed to the perfect reformulation proposal from Calvanese [Calvanese et al., 2007a], which is evaluated only in theoretical terms w.r.t. complexity, completeness and correctness. Pérez-Urbina compared his approach with this by using a set of ontologies that have become common across evaluations in this area:

- Adolena (A). Developed to allow OBDA for the South African National Accessibility Portal [Keet et al., 2008]. This is the largest ontology in this set of ontologies.
- path1 (P1) and path5 (P5). Synthetic ontologies to help understand (and show) the “impact of the reduction step” in REQUIEM. Despite their apparent simplicity, rewriting times for Datalog in these ontologies are significant.
- StockExchange (S). It captures information about European Union financial institutions, Rodríguez-Muro [Rodríguez-Muro et al., 2008] uses this as a driving example to explain OBDA and how users may benefit from it.
- University (U). A DL-Lite<sub>R</sub> version of LUBM [Guo et al., 2005]. LUBM focuses on the ABox more than the TBox. On the one hand this allows systems like Clipper to use the ABox for further evaluation of rewritten queries. On the other hand it has a rather flat TBox [Rodríguez-Muro and Calvanese, 2012], which means that the rewritings are simple, it takes a short time to be produce them and the rewritten queries are also short, as we will see in the next section.
- Vicodi (V). An ontology of European history developed in the EU-funded VICODI project [Nagypal, 2005].

This set of ontologies was expanded with AX, P5X, and UX, where some of the previous ontologies included auxiliary predicates. These auxiliary predicates

## 2. BACKGROUND AND STATE OF THE ART

---

replace the existential predicates by applying the encoding required by the previous approach [Calvanese et al., 2007a]. These ontologies are accompanied by a set of five queries for each of them.

For the evaluation of Presto these sets of queries are increased up to seven queries for each of the ontologies. The ontologies are also expanded with the ontologies from Kontchakov [Kontchakov et al., 2009], more examples from the LUBM benchmark (besides of U and UX) and a newly created ontology. The ontologies from Kontchakov are:

- Galen-Lite. The *DL-Lite<sub>core</sub>* approximation of the well-known medical ontology Galen [Rogers and Rector, 1996]. The interest in this ontology is mainly in its taxonomy, since few axioms involve roles.
- Core. A *DL-Lite<sub>core</sub>* representation of (a fragment of) a supply-chain management system used by the bookstore chain Ottakar’s, now rebranded as Waterstone’s. Contrary to Galen, the taxonomy in Core is smaller but it contains a rich set of axioms about roles.

These assets have been used primarily for evaluation purposes. The expansion of these assets has normally focused on showing the specific characteristics of the system evaluated at each time. However, the benchmarking of all systems has not been a priority for the development of new systems.

We find the first and most noteworthy example of benchmarking query rewriting systems in the work of Imprialou [Imprialou et al., 2012], who proposes an algorithm to generate an automatic benchmark for such systems.

This algorithm accepts an ontology as the input and generates a set of queries. There is no statement about how well these queries may represent real queries that could be posed by users. The automatic generation of the queries allows querying for all the concepts in the ontology. This coverage allowed the detection of problems in the soundness and completeness of the benchmarked algorithms. Most of the detected problems were solved in latter versions of these systems. Therefore, a good coverage is key for an appropriate benchmarking and even for the individual testing of the systems.

We will see in chapter 7 that we give a more granular and quantitative focus to queries and their results assuming soundness and completeness wrt a given

---

expressiveness. We do also analyse ontologies and the impact that their characteristics can have on the behaviour of query rewriting systems.

### 2.4.2 A general algorithm for query rewriting

In OBDA, when using the knowledge in some ontology to reformulate a query, one of the most common procedures is to convert both the query and the ontology to Datalog [Ceri et al., 1989]. Then the rewriting process is done by performing inferences on the generated Datalog program, by iteratively applying deduction steps that lead from the original query to the rewritten query. We can consider that the approaches mentioned in the previous section are derivations from a general algorithm, as represented in algorithm 2.1. The input for this general algorithm is an ontology and a query, and the output is the rewritten query. We can see this algorithm simply as a succession of steps.

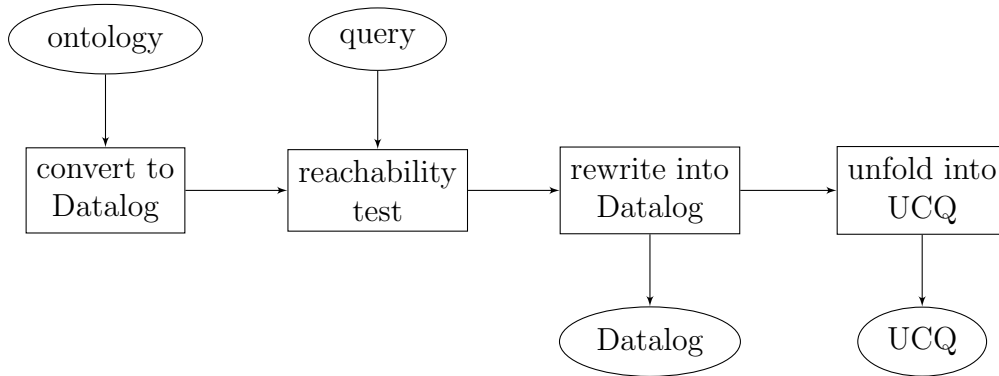


Figure 2.1: Stages in the general algorithm

1. The first step is *parsing* the ontology from the serialization in which it is made available, for instance OWL in RDF/XML or turtle format. After the ontology has been parsed it is converted into *Datalog clauses*. The part of the ontology that is out of the expressiveness handled by the algorithm at hand is usually ignored and discarded. It must be noted that in the case where part of the ontology is discarded, the results from the query rewriting may be inconsistent with the ontology iff the ontology is inconsistent with the data source considered, otherwise the results may be incomplete but will preserve soundness.

## 2. BACKGROUND AND STATE OF THE ART

---

2. The most general step is the *rewriting* itself. This is performed through inferences on the Datalog obtained from the previous stage to obtain some different Datalog or a UCQ, depending on the case. The inference is always based on the traditional resolution as can be done in Datalog, for instance backward chaining. To this basic inference several modifications are made depending on the expressiveness handled, the input accepted and the output produced. These modifications are done with the purpose of reducing the time needed for these inferences and obtaining better (usually shorter) queries from the rewriting process. During this stage many subsumed and subsuming clauses can be produced, which may represent a problem in terms of efficiency. Different systems cope with them in different ways, e.g. performing a subsumption check after resolution to remove all subsumed clauses.
3. The final step consists in the *evaluation* of the produced query by the underlying systems. The variability in this step is greater than in the other two. Some systems may use the query in Datalog, some may convert it to SPARQL, SQL or other languages and pose it to the system storing the data or to another system for query translation with some mappings. In most approaches, this step is omitted from the descriptions provided in their corresponding papers, and the evaluation focuses on the properties of the Datalog query produced, the process for its production and possibly additional inputs that can be considered for further optimisation.

Different algorithms make different modifications to this general schema.

- In the case of REQUIEM:
  - An additional stage is added between steps one and two in order to reduce the size of the ontology that is being used to only the part of the ontology that is reachable. This is because the inference done in the following stage is done through saturation of resolution with free selection instead of using backward chaining from the query. This means that all the clauses available at that stage are used in the inference among all the other clauses available and produced during that stage.



---

Thus a reduction in the number of clauses provides an important reduction in the later inferences and processing.

- The second step is split into two different resolution steps. In the case of REQUIEM the ontology does not generate a Datalog program to unfold but a logic program (with functional terms). The first resolution step performs the inferences needed to convert the logic program into a Datalog program. The Datalog program is generated by performing the resolutions that lead to new clauses that do not contain the functional terms and dropping clauses that contain some functional term. The second resolution step unfolds the Datalog program into a UCQ if it is not recursive, otherwise it is impossible. This is considered a pay-as-you-go approach because depending on the expressiveness of the ontology different results can be obtained, either a Datalog program or a UCQ.
  - between steps two and three some additional optimisations are performed to reduce the size of the query, as subsumption check and condensation of the produced queries.
- In the case of Presto:
    - Presto is the system in the comparison whose algorithm differs to a greater extent from the general algorithm previously described. The second step does not consist on inference but mainly on splitting the query into smaller fragments through the elimination of existential joins (EJ) when a most general subsumee (MGS) can be found. The inference is performed to obtain the MGS, while the subsumption checks and removals between clauses are implicit in the process to replace the atoms containing the EJ with those in the MGS.
  - In the case of Rapid:
    - The second step can perform the inference in two different ways, either to generate a Datalog program or a UCQ. The difference is that in the case of the UCQ generation more inference rules are added to the resolution to remove functional terms and unfold the program.

## 2. BACKGROUND AND STATE OF THE ART

---

- Rapid accepts a logic program which may contain functional terms, as REQUIEM does. The inferences that are possible with these functional terms are very reduced due to the limited presence of functional terms, therefore the inference is done with an additional rule that groups vertically several inferences to handle more efficiently clauses that contain functional terms. This is done with the step named “query shrinking”.
  - The third main difference in the resolution step for Rapid is how the generation of some subsumed clauses is prevented. This is done again by grouping several inference steps into one, but this time horizontally. The rule applied in this case is named “query unfolding”. This groups inferences in two ways. First, all the atoms that can replace a given atom are grouped, this is possible due to the linear nature of the Datalog used. Second, all the sets of atoms that can replace atoms in a single query are grouped. With this, all the combinations of atoms in these sets can be generated according to the atoms present in some given query. The generation of some subsumed queries is avoided by checking subsumption between these sets of atoms before producing the combinations.
- In the case of Nyaya:
    - The main difference in Nyaya consists on applying a stratified strategy to the inferences performed during the second step in a similar fashion to what Rapid does with the two resolution rules. In this case the names for the stages are “factorization” and “rewriting”.

### 2.4.3 The REQUIEM Algorithm

More precisely, we take as the starting point the general OBDA query rewriting algorithm in REQUIEM (RQR), due to the decisions taken in several aspects, including engineering aspects (e.g. modularity) and scientific aspects (e.g. expressiveness), political aspects (e.g. open source), and legal aspects (e.g. license for the code). RQR is structured in several resolution steps. Every resolution

---

step follows the general algorithm for resolution (algorithm 2.1). We reproduce it here as originally specified [Pérez-Urbina et al., 2010].

---

**Algorithm 2.1:** REQUIEM resolution algorithm (RQR)

---

**Input:** Conjunctive query  $Q$ ,  $DL\text{-}Lite_R$  ontology  $\mathcal{O}$

```

1  $R = \Xi(\mathcal{O}) \cup \{Q\}$ 
2 repeat
3   (saturation) forall the clauses  $C_1, C_2$  in  $R$  do
4      $R = R \cup \text{resolve}(C_1, C_2)$ 
5   end
6 until no query unique up to variable renaming can be added to  $R$ 
7  $Q_{\mathcal{O}} = \{C \mid C \in$ 
    $\text{unfold}(\text{ff}(R)), \text{ and } C \text{ has the same head predicate as } Q\}$  return  $Q_{\mathcal{O}}$ 
```

---

As we can see, there is a first stage (**resolve**) that generates the Datalog program by resolving the present clauses up to saturation, and a second stage (**unfold**) that unfolds this Datalog program into a UCQ.

This algorithm in REQUIEM can be specified in pseudocode with a greater level of detail, as in algorithm 2.2. This description is more detailed and similar to the algorithms that we will show in following chapters.

It is worth mentioning that REQUIEM accepts  $DL\text{-}Lite_R$  for the production of a rewriting that is a UCQ as stated in algorithm 2.2. However  $\mathcal{ELHIQ}^{\neg}$  ontologies are accepted with the exception that if the Datalog produced is recursive, the recursion will be preserved after the unfolding, and thus the result will be a (linear and recursive) Datalog program instead a UCQ.

## 2.5 OBDA mappings in query rewriting

As we have seen before, mappings are used for the translation of queries written according to an ontological model into queries written according to the model used to store and access data in the underlying data sources. The intersection between the terms or constructs used in each type of model (e.g., concepts and their properties, and tables and their columns) normally corresponds to a small fraction of the set of terms in the ontology. There are several reasons for this and they are schematically represented in figure 2.2. Ontologies in OBDA provide

## 2. BACKGROUND AND STATE OF THE ART

---



---

### Algorithm 2.2: General REQUIEM algorithm

---

**Input:** Conjunctive query  $Q$ , DL-Lite<sub>R</sub> ontology  $\mathcal{O}$ , working mode  $mode$   
**Output:** Rewritten query  $q_\Sigma$

```

1  $R = \Xi(\mathcal{O}) \cup \{Q\}$ 
2  $R = \text{reachable}('Q', R)$ 
3  $q_\Sigma = \text{requiemSaturate}(R, sfRQR, mode)$ 
4 if  $mode = \text{greedy}$  then
5    $predicates = IDBpredicates(q_\Sigma)$ 
6   for  $p \in predicates$  do
7      $q_\Sigma = \text{requiemSaturate}(q_\Sigma, sfUnfoldGreedy(p), \text{greedy})$ 
8   end
9 else
10   $q_\Sigma = \text{requiemSaturate}(q_\Sigma, sfUnfoldNaive, \text{naive})$ 
11 end
12 if  $mode \neq \text{naive}$  then
13    $q_\Sigma = \text{subsumptionCheck}(\text{pruneAUX}(\text{reachable}('Q', q_\Sigma)))$ 
14    $q_\Sigma = \text{map}(\text{condensate}, q_\Sigma)$ 
15 end
16 return  $q_\Sigma$ 

```

---



---

### Algorithm 2.3: REQUIEM saturation algorithm (`requiemSaturate`)

---

**Input:** input program  $P$ , selection function  $sf$ , working mode  $mode$   
**Output:** saturated program

```

1  $unprocessed = P$ 
2  $workedOff = \emptyset$ 
3 while  $unprocessed \neq \emptyset$  do
4    $currentClause = unprocessed.pop()$ 
5    $sf.select(currentClause)$ 
6    $workedOff.add(currentClause)$ 
7   for  $previousClause \in workedOff$  do
8     for  $\gamma \in \text{resolve}(currentClause, previousClause)$  do
9       if  $\neg \text{isRedundant}(\gamma, mode, workedOff \cup unprocessed)$  then
10         $unprocessed.add(\gamma)$ 
11      end
12    end
13  end
14 end
15 return  $\text{filter}(sf.prune, workedOff)$ 

```

---

---

**Algorithm 2.4:** REQUIEM redundancy detection algorithm (**isRedundant**)

---

**Input:** clause  $clause$ , working mode  $mode$ , set of clauses  $P$

**Output:** true or false

```
1 if  $mode = \textit{naïve}$  then
2   | return  $\exists \gamma \in P. \textit{equivalent}(clause, \gamma)$ 
3 else
4   | return  $\exists \gamma \in P. \gamma \succeq_s clause$ 
5 end
```

---

---

**Algorithm 2.5:** REQUIEM reachability algorithm (**reachable**)

---

**Input:** Root predicate  $Q$ , set of clauses  $P$

**Output:** set of clauses  $P$

```
1  $pending = \{Q\}$ 
2  $reachable = \emptyset$ 
3 while  $pending \neq \emptyset$  do
4   |  $pred = pending.pop()$ 
5   | if  $pred \notin reachable$  then
6   |   |  $reachable.add(pred)$ 
7   |   |  $pending = pending \cup \{p \in P \mid \exists \gamma \in P. pred \in head(\gamma) \wedge p \in body(\gamma)\}$ 
8   | end
9 end
10 return  $\{c \in P \mid head(\gamma) \in reachable\}$ 
```

---

## 2. BACKGROUND AND STATE OF THE ART

---

a richer (broader) terminology and a greater expressiveness. This richer terminology means that users can pose queries more comfortably in several different ways (e.g. a richer terminology allows using synonyms and providing multilingual access, so that users with different views can query the same underlying information). A richer terminology means that one single concept can encapsulate a greater semantic load, summarising what would need several terms otherwise. For instance, an ontology in a hospital can include the concept “CancerChild-Patient” to refer to the patients that are in a certain age range and suffer from cancer, which may imply two joins in the underlying datasource. This expressiveness means shorter and more compact queries in general. Besides the richer terminology, there are some usual characteristics in ontologies that make them different from regular knowledge bases. Among these characteristics are the generality of the knowledge captured by ontologies, the reusability of ontologies (*a priori*) and the links between them (reusability *a posteriori*). Due to these characteristics ontologies have usually a broader scope than the application that is considered at a specific time, and thus a broader terminology. As the terminology becomes broader, we will logically find in that terminology less terms mapped (in proportion) to the data sources.

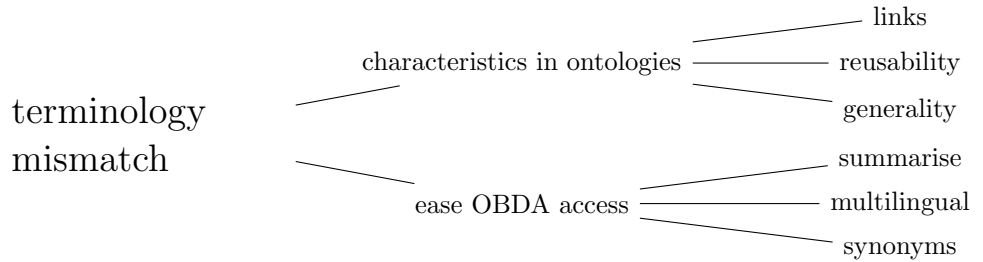


Figure 2.2: Reasons for the lack of mappings for some terms.

### 2.5.1 Mapping characterisation

Mappings may be specified as global-as-view (GAV), local-as-view (LAV) [Lenzerini, 2002], or both global-and-local-as-view (GLAV) [Friedman et al., 1999]. In the case of GAV, the mappings can be considered to be a set of assertions of the

---

form:

$$m_A = (q_{\langle D, A \rangle}(\vec{x}) \rightsquigarrow A(f(\vec{x})))$$

$$m_P = (q_{\langle D, P \rangle}(\vec{y}) \rightsquigarrow P(f_1(\vec{y}_1), f_2(\vec{y}_2)))$$

Where  $q_{\langle D, A \rangle}$  and  $q_{\langle D, P \rangle}$  are queries over the data source  $D$  to obtain the values necessary for the basic concept  $A$  or the role predicate  $P$  respectively, which belong to the ontology  $\Sigma$ ,  $\vec{y}_1 \cup \vec{y}_2 = \vec{y}$  and  $f$ ,  $f_1$  and  $f_2$  are the transformation functions that allow generating the values for the semantic upgrade that is performed when converting the information from the database to ontological instances. In a LAV context, finding the terms in the ontology that are mapped is not as straightforward as in a GAV context, and LAV mappings require a slightly longer translation process. LAV mappings are assertions of the form:

$$m_p = (q_{\langle \Sigma, p \rangle}(\vec{x}) \rightsquigarrow p(f(\vec{x})))$$

Which means that for every  $p_i$  mapped in the data source we will have a set of predicates in the body of  $q_{\Sigma, P_i}$  which are the mapped predicates in the ontology for that query. The combination of both types of mappings at the same time is possible, with GLAV mappings. A single mapping assertion  $m$  can be considered GLAV if both head and body of  $m$  contain free variables, taking the form:

$$m_{\langle \vec{p}, \vec{A}, \vec{P} \rangle} = \left( q_{\langle D, \langle \vec{A}, \vec{P} \rangle \rangle}(\vec{x}, \vec{y}) \rightsquigarrow q_{\langle \Sigma, \vec{p} \rangle}(\vec{x}, \vec{z}) \right)$$

This relates a query on the data source  $D$  to a query on the ontology  $\Sigma$ . In any case the union of all mapped predicates is the set of predicates in the ontology that are mapped, and so in short we have that:

$$ontologyPredicates(m_p) = \begin{cases} \{p\} & : isGAV(m_p) \wedge p \in head(query(m_p)) \\ \{p_i\} & : isLAV(m_p) \forall p_i \in body(query(m_p)) \\ \{p_i\} & : isGLAV(m_p) \forall p_i \in head(query(m_p)) \end{cases}$$

$$mappedPredicates(\mathcal{M}) = \bigcup_{m_p \in \mathcal{M}} ontologyPredicates(m_p)$$

## 2. BACKGROUND AND STATE OF THE ART

---

In all cases, a computationally simple syntactic processing of the mapping assertions is enough to obtain the ontology predicates that are mapped. This information can be used for optimisation purposes, as detailed in chapter 4. Intuitively, these optimisations work on the basis that rewritten queries that contain predicates that are not mapped are useless in a UCQ since they will lead to empty answers. Therefore, such queries can be added or removed from the UCQ without impact on the certain answers. The reason for this lack of impact is in the impossibility to translate these queries to obtain answers with the translated queries from the data source. In fact, the lack of mappings for some predicate in the ontology will be unavoidably noticed in the translation process, which will have to abort the translation of that query. We will see in chapter 4 that it is possible to avoid the generation of such queries in much earlier stages in the query rewriting process. This anticipation produces generally an improvement in the time needed for query rewriting, a shorter rewritten query that does not contain non-mapped predicates and a rewritten query that obtains the same certain answers from the data source.

### 2.6 EBoxes and query rewriting

Ontologies are usually decomposed into ABox (assertional box) and TBox (terminological box). The former includes the assertions or facts, corresponding to the individuals, constants or values for the previously mentioned extensional predicates. The latter describes the concepts or predicates in the ontology and how they relate among them with a set of axioms in description logics (DL). These axioms can be converted to rules or implications (and viceversa) in first order logic (more expressive) and to some extent in Datalog (less expressive than unconstrained DL).

ABox dependencies [Rodríguez-Muro and Calvanese, 2011], or extensional constraints [Rosati, 2012], are assertions that restrict the syntactic form of allowed or admissible ABoxes. These assertions can be compiled into an extensional box (EBox), with potentially (and usually conveniently) the same expressiveness as the TBox.

For example we may have a system to manage the students in a university,



---

and we may want to retrieve a list of all the students. Consider for this system the following TBox:

$$\begin{array}{ll}
\textit{UndergradStudent} \sqsubseteq \textit{Student} & \textit{MasterStudent} \sqsubseteq \textit{GradStudent} \\
\textit{IndustryMasterStudent} \sqsubseteq \textit{MasterStudent} & \textit{PhdStudent} \sqsubseteq \textit{GradStudent} \\
\textit{ResearchMasterStudent} \sqsubseteq \textit{MasterStudent} & \textit{GradStudent} \sqsubseteq \textit{Student} \\
\textit{BachelorStudent} \sqsubseteq \textit{UndergradStudent} &
\end{array}$$

The following EBox:

$$\begin{array}{llll}
\textit{IndustryMasterStudent} & \sqsubseteq & \textit{GradStudent} & \textit{Student} \sqsubseteq \perp \\
\textit{ResearchMasterStudent} & \sqsubseteq & \textit{GradStudent} & \textit{BachelorStudent} \sqsubseteq \perp \\
\textit{PhdStudent} & \sqsubseteq & \textit{GradStudent} & \textit{MasterStudent} \sqsubseteq \perp
\end{array}$$

And an ABox satisfying the previous EBox, for example an ABox with the following individuals:

- *UndergradStudent*: Al
- *GradStudent*: Ben, Cal, Don, Ed
- *ResearchMasterStudent*: Ben
- *IndustryMasterStudent*: Cal
- *PhdStudent*: Don

Querying for the most general concept (*Student*) would yield no results. Querying for the most specific concepts (*BachelorStudent*, *ResearchMasterStudent*, *IndustryMasterStudent* and *PhdStudent*) requires four queries and yields an incomplete answer, missing Ed and Al in the example. Finally querying for all concepts would provide all answers, but that implies eight queries (one for each concept) and retrieving some duplicates. In this case the duplicates are Ben, Cal and Don. Duplicated answers have no impact on the correctness of the answer set, but they are a big burden in the efficiency of the process when considering more complex queries and ontologies. In particular, in the example we only need three queries (as opposed to eight) to retrieve all answers, querying respectively for instances of *UndergradStudent*, *GradStudent* and *IndustryMasterStudent*, since the EBox states that the ABox extension of every other concept is either

## 2. BACKGROUND AND STATE OF THE ART

---

empty or contained in *GradStudent*. There are therefore six queries that are only a waste of computational resources in the query answering process.

A naïve algorithm may generate the perfect rewriting and then reduce it by checking for subsumption with the EBox. However, such a naïve algorithm could have a prohibitive cost for large rewritings and would only be applicable over non-recursive rewritings. In the following sections we will show that it is possible to face more complex scenarios and handle them better than with such a naïve algorithm.

This example illustrates that the combination of ABoxes that are already (partially) complete and a complete query rewriting on the TBox causes redundancy in the results, which is a burden for efficiency. Hence, the characterization of ABox completeness as a set of dependencies can serve to optimise TBoxes, and create ABox repositories that appear to be complete [Rodríguez-Muro and Calvanese, 2011]. Additional optimisations can be done with the Datalog query before unfolding it into a UCQ, and finally with the UCQ, reducing redundancy at every step. For instance, in our example we have in the EBox that  $PhDStudent \sqsubseteq GradStudent$  just like in the TBox. Therefore, we do not need to consider this axiom in the TBox when retrieving students: the ABox is complete in that sense and no *GradStudent* needs to be obtained from *PhDStudent*.

Using the EBox, the perfect rewriting can be reduced along with the inference required for its generation. We can redefine the perfect rewriting in the presence of EBoxes as follows [Rosati, 2012]: a perfect rewriting for a UCQ  $q$  and a TBox  $\mathcal{T}$  under an EBox  $\mathcal{E}$  is a query  $q'$  such that, for every ABox  $\mathcal{A}$  that satisfies  $\mathcal{E}$ ,  $\Phi_{\langle \mathcal{T}_{\emptyset}, \mathcal{A}_{\emptyset} \rangle}^q = \Phi_{\langle \emptyset, \mathcal{A} \rangle}^{q'}$ .

EBoxes can be manually described, or automatically extracted. The automatic extraction can be done from an ABox (materialised or virtual), by describing the extensional containment relationships at a given time and updating the EBox if these relationships change. Alternatively, EBoxes can be automatically extracted from the definition of an OBDA system, in particular from the mappings and the database schema, inferring from the implied intensional properties the constraints that the ABox will satisfy [Console et al., 2013]. Finally, combining several of the previous techniques for the production of EBoxes is technically feasible but that is a possibility not explored yet to the best of our knowledge.

---

## 2.7 Limitations in the state of the art

The problem of query rewriting is well defined, the theoretical results are formally proved to be sound and complete with respect to the handled expressiveness in each case, what provides a good foundation to work on. In such cases when the systems are not complete or correct they can be fixed to conform to the theoretical results, as seen with Imprialou [Imprialou et al., 2012].

In this case the limitations reside in the contextualisation of the problem and the quantification of the results. So far, the focus has been set on the problem of query rewriting disregarding its context. As a result, the analysis performed on query rewriting has been mainly qualitative. Due to the relevance of resolution in query rewriting, the focus has been on a set of algorithms that perform resolution on a query and an ontology for some specific expressiveness. From a theoretical point of view, the most relevant aspects of resolution are completeness and soundness. However, as a consequence, there are several aspects that have been omitted in the state of the art. We will enumerate the most relevant of these aspects<sup>1</sup>.

**SAL1.** Underlying systems capabilities. Some predicates may not be mapped.

The first aspect to consider in the context are the underlying systems that will evaluate the rewritten query. The capabilities of these systems should be considered for any pragmatical use of the rewritten query. One of the ways to consider the capabilities of these systems are the mappings defined for the data sources. For example, if some predicate is not mapped then it cannot be translated and it will normally not be possible to translate the full conjunctive query or obtaining answers from the translation. The mappings with the data sources have not been considered in general in existing approaches. They are only considered in Prexto through the use of an extensional box (EBox). However the definition of the mappings may be used as an additional input to the system. Considering the mappings available for the underlying systems opens the possibility of generating an output addressed at those mappings. This means that the query rewriting process can be made more efficient and the output shorter and easier to process.

---

<sup>1</sup>to do so we will use the prefix “SAL” (State of the Art Limitation)

## 2. BACKGROUND AND STATE OF THE ART

---

Therefore, we will consider mapping definitions to improve the query rewriting process.

**SAL2.** Qualitative aspects of the input. Expected expressiveness.

We have to consider two different aspects of the input: the ontologies and the queries.

(a) The expressiveness of the ontologies has been considered to produce algorithms that can handle some specific expressiveness. However there are no estimations about what is the expected expressiveness in an OBDA system, how users write ontologies for this task or which ontologies are commonly used. Most of the previous work has focused on *DL-Lite* and *OWL2 QL*. For our approach we set the expressiveness in  $\mathcal{ELHJO}$ , which displays two interesting properties: On the one hand it is still tractable, as mentioned in section 2.2. On the other hand it is among the most expressive logics used in the state of the art for query rewriting, and in particular it is the most expressive logic used in the state of the art that is mentioned as tractable. We will set a special focus on algorithmic optimisations, which are more generalisable to a different expressiveness.

(b) There are no estimations about the expressiveness or size of the queries either. We will perform an extensive evaluation of our system and other systems with queries of different sizes. Queries are normally limited to conjunctive queries for the systems in the state of the art. We will extend the expressiveness of the input queries from the traditional conjunctive queries to unions of conjunctive queries.

**SAL3.** Additional input information to consider. EBox.

We have seen that (only) Prexto uses an EBox to produce shorter rewritten queries that can be handled more efficiently. An EBox can contain information about mappings that may be present but redundant with respect to other mappings. This allows generating rewritten queries that are better addressed at the capabilities of the underlying systems, possibly removing parts of the query that would generate correct but redundant answers (under set semantics). We will also add the possibility of considering an EBox in our system for the  $\mathcal{ELHJO}$  expressiveness.

---

**SAL4.** Qualitative focus. Missing opportunities in the engineering optimisations.

The qualitative focus has also led authors to obtain algorithms with better theoretical complexities. These algorithms are possible for logics of a reduced expressiveness. We will see that we can obtain comparable rewriting times in the expressive logic  $\mathcal{ELHIO}$  by implementing some engineering optimisations.

## **2. BACKGROUND AND STATE OF THE ART**

---

# Chapter 3

## Objectives

In this chapter we state the work hypotheses that we want to validate, under certain assumptions and limitations. From these hypothesis, we draw the thesis objectives and discuss the associated conceptual, methodological and technological contributions proposed in this work.

### 3.1 Problem statement

This work aims at providing theoretical foundations and technical solutions for query rewriting in an OBDA context. From the analysis of the state of the art in section 2.7 we have seen that query rewriting approaches focus on a theoretical frame where some additional considerations may be made. More precisely we propose to improve the applicability, contextualisation, performance (hence applicability), maintainability and results of query rewriting techniques by:

1. Using the information provided by mappings in query rewriting.

Considering this information should improve contextualisation (awareness of mappings), results and performance .

2. Using better-engineered methods to better address the problem of query rewriting in OBDA considering its specific characteristics.

Better-engineered methods improve the performance in query rewriting and the maintainability of the software through further modifications.

### 3. OBJECTIVES

---

3. Using the information provided by an EBox to target the predicates<sup>1</sup> that will produce the set of certain answers.

Using this information should improve contextualisation (awareness of the knowledge in the EBox) and the results of query rewriting.

4. Focusing on resolution to handle the  $\mathcal{ELHIQ}$  expressiveness to maintain the applicability of the results both in theory and practice.

By keeping an expressive logic ( $\mathcal{ELHIQ}$ ) and standard procedures (resolution) we expect to preserve maintainability, obtaining solutions that can be used in a variety of systems, at least in  $\mathcal{ELHIQ}$  expressiveness and possibly in more expressive logics.

For the first problem the following research questions are addressed in this work:

- Can the information about mappings be useful for query rewriting in an OBDA context?

OBDA systems use mappings to translate the query from the terms used in the ontology to the terms in the underlying data source. Therefore the mappings can provide meta-information about the characteristics of the information contained in the data source. However this meta-information is not being used in state of the art systems. We explore in chapter 4 the possibilities that mappings offer as a necessary and already existing description of the data sources that is usually declarative.

- Can the information from mappings be used in an efficient way?

The information from the mappings allows generating rewritten queries that the underlying data source can execute more efficiently while obtaining the same results. This information may also provide additional constraints in the query rewriting process and considering these constraints may help to reduce the computational load to rewrite the queries. Using the information from mappings (if possible, according to the previous question) may be

---

<sup>1</sup>Reminder: We will refer to concepts and properties in a TBox as “predicates”, corresponding to predicates in FOL and Datalog.



---

unfeasible in practical terms. On the contrary, using the information from mappings may mean not longer times to generate the rewriting but a gain in efficiency.

For the second problem we consider the following questions:

- How can we tune current algorithms for query rewriting in OBDA?

Most query rewriting techniques are based on resolution. Resolution is a very general technique; there are many different rules for resolution and many different calculi that can be used. Resolution is also a very useful technique; there are many different purposes for which we can use resolution, such as theorem proving, validation, search for inconsistencies, etc. We may be able to tune these techniques to better address query rewriting by considering its specificity in an OBDA context.

- How can we measure the appropriateness of the solutions to the query rewriting problem?

The evaluation for resolution techniques is primarily formal, in terms of soundness and completeness. When performing optimisations in some technique the soundness and completeness should remain unaltered while obtaining better results (optimisations). This improvement in the results will usually refer to the process time (efficiency), although it may also refer to the obtained output (e.g. terseness of an equally correct solution). Since the correctness is equal for the same assumptions, the evaluation of the optimisations performed in any given algorithm has to be empirical. This means using benchmarks to perform this empirical evaluation and assessing the quality of these benchmarks.

Finally for the third problem we can consider the following questions:

- Can EBoxes be used in query rewriting?

An EBox contains information about the extension of the ontology predicates in a given data source. Similarly to mappings, the information in an EBox can potentially be used to improve the process and results of query

### 3. OBJECTIVES

---

rewriting. In this case we are interested in using an EBox in a general context within query rewriting in OBDA, in a general way that can be later addressed to potentially any given algorithm.

- What is the impact of the different EBoxes?

Probably not all EBoxes are the same. We study the characteristics of the different EBoxes and try to draw conclusions about the impact of these characteristics in the query rewriting process and results. This could suggest transformations to perform on EBoxes or ABoxes (if possible) so that query rewriting algorithms can obtain a better performance.

Finally, the fourth problem is transverse and present in all the previous problems. All previous problems will be tackled without reducing the expressiveness from

## 3.2 Hypotheses

We formalise the work hypotheses that follow from the research challenges discussed in the previous section. We include also the assumptions made in the formulation of these hypotheses.

**H1.** The information about the presence or absence of a mapping for a given predicate generally allows reducing the size of the rewritten queries and the time needed to produce them.

**H2.** Additional computations can be done during query rewriting so as to obtain a gain in efficiency that compensates for the cost of these computations.

**H3.** EBoxes provide information that may allow reducing the size of the rewritten queries and reducing the time needed in the query rewriting process.

### 3.2.1 Assumptions

The hypothesis are stated considering the following assumptions (not all assumptions apply to all hypotheses):

---

**A1.** It is possible to obtain an exhaustive list of the predicates in the ontology that are mapped to some data source according to a given set of mappings.

**A2.** Predicates that are not mapped cannot be retrieved from the data source. They are represented by an empty set of facts and therefore a join with one of these predicates produces an empty set of answers.

**A3.** EBoxes can be derived from the schema and data available in data sources.

**A4.** We assume the use of set semantics in the answers, instead of the bag semantics that is assumed in SPARQL query evaluation. This means that redundant answers to queries have no impact on the correctness of the queries (e.g. there is no `COUNT` operator).

**A5.** Mappings between a data source and an ontology do not necessarily map all the predicates in the ontology in all ontologies.

### 3.2.2 Limitations

The goals of this PhD thesis are bound to the following limitations:

**L1.** We restrict the input and output of the system to UCQs with the output being possibly recursive Datalog if the query involves a recursive definition in the ontology. Datalog, SPARQL and SQL have been proved to be equally expressive under some circumstances [Angles and Gutierrez, 2008] and similarly expressive otherwise. Those circumstances are not part of the assumptions, hence the input and output are only guaranteed to be similarly expressive. For example, from a logic perspective, two individuals that are exactly equal represent the same individual, therefore only `distinct` individuals can be counted.

**L2.** The reduction in the time and size of the rewritten queries is determined by how many predicates in the ontology are not mapped. An ontology that is completely mapped will produce no reduction or a negative reduction in time if the operations related with mappings are attempted with it.

### 3. OBJECTIVES

---

**L3.** The impact of additional computations in query rewriting may not be significant or may even be negative in the most simple queries or in synthetically generated queries specifically designed to render the additional computations useless. We aim at improving the average performance in a representative set of queries.

## 3.3 Contributions

The main contribution of this work is a novel set of methods and techniques for query rewriting in OBDA. This body of work stems from the investigation of this problem in isolation as well as in a broader context. The conceptual contributions are:

- the focus shift in taking a broader scope for the problem of query rewriting, including mappings.
- the focus shift in taking a more empirical and quantitative evaluation of query rewriting solutions.
- an algorithm to reduce redundancy and combinatorial explosions in query rewriting (focused on  $\mathcal{ELHIJ}$ ).
- an algorithm that considers the presence of mappings in query rewriting.
- an algorithm that considers  $\mathcal{ELHIJ}$  EBoxes in query rewriting.

The technological contributions are:

- the implementation of the previously mentioned algorithms, separately.
- a system that implements and combines the previously mentioned algorithms.
- a benchmark suite for the evaluation of query rewriting systems.

## Chapter 4

# Query rewriting optimisations in the presence of RDB2RDF mappings

Two aspects need to be considered when analysing a query rewriting approach from a practical perspective: the time required for the generation of the rewritten query and the time required for query evaluation. In this chapter we describe how we can use the information available in RDB2RDF mappings to optimise the query rewriting process as well as to reduce the size of rewritten queries, what may have a positive influence in query evaluation. Our approach consists in avoiding as soon as possible working with predicates that are not mapped, since such predicates that have only an intensional definition will not produce any answers during query evaluation, and can be removed safely.

This enables an early reduction in the number of clauses, which in turn constrains the search space before other processing stages, reducing the size of the Datalog program and the inferences that have to be carried out by latter stages. All these reductions have a cost, but we will see that in realistic scenarios this cost usually pays off and the overall performance of the whole query rewriting process is significantly improved. This reduction in the total number of predicates and therefore in the number of clauses allows for producing rewritten queries faster and these queries are also simpler due to these reductions. This means that later

## 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

processing stages will face a lower computational load.

Producing simpler queries is of tantamount importance. This is evidenced by the evaluation performed by previous work [Gottlob et al., 2011; Pérez-Urbina et al., 2009; Rosati and Almatelli, 2010], where one measure (or even the only measure) about the quality of the rewritings is precisely its briefness (number of clauses) as an approximation of their simplicity. We will also see that there are additional factors to consider when assessing the simplicity of a rewritten query.

As described in chapter 1 and represented in figure 4.1, once the rewriting has been done, mappings are used for the translation between the ontological model and the model used in the data sources. For instance RDB2RDF mappings would be used to transform the obtained Datalog (possibly a UCQ) predicates into SQL queries. In OBDA contexts, several mapping languages have been proposed (for instance in the case of RDB2RDF we can find D2R [Bizer, 2003] or R2O [Barrasa et al., 2004], now deprecated, or more recently, R2RML [Das et al., 2012]). Mappings defined with one of these languages may be complete or partial for some ontology in an OBDA system. If mappings are complete, then all the predicates in the signature of the ontology will be mapped. If the mappings are partial, then some predicates in the ontology will not be mapped, i.e. the mappings will simply not refer to these predicates. Partial mappings may allow further reductions in the size of the rewritten queries that are generated by enabling the rewriting process to generate an equivalent query with respect to the provided mappings. Considering mappings changes the OBDA scenario that we have seen in figure 1.1 to the scenario that we can see in figure 4.1. The changes may seem subtle, they consist simply on a dependency on the mappings for the rewriting process. However, considering the mappings has relevant consequences on how the rewriting can be done, the time that it requires and the output that is generated, as we will see.

The impact of this reduction will depend on the number of predicates that are mapped by a set of mappings. To illustrate the potential impact in a realistic context we can take the mappings generated for the Food and Agriculture

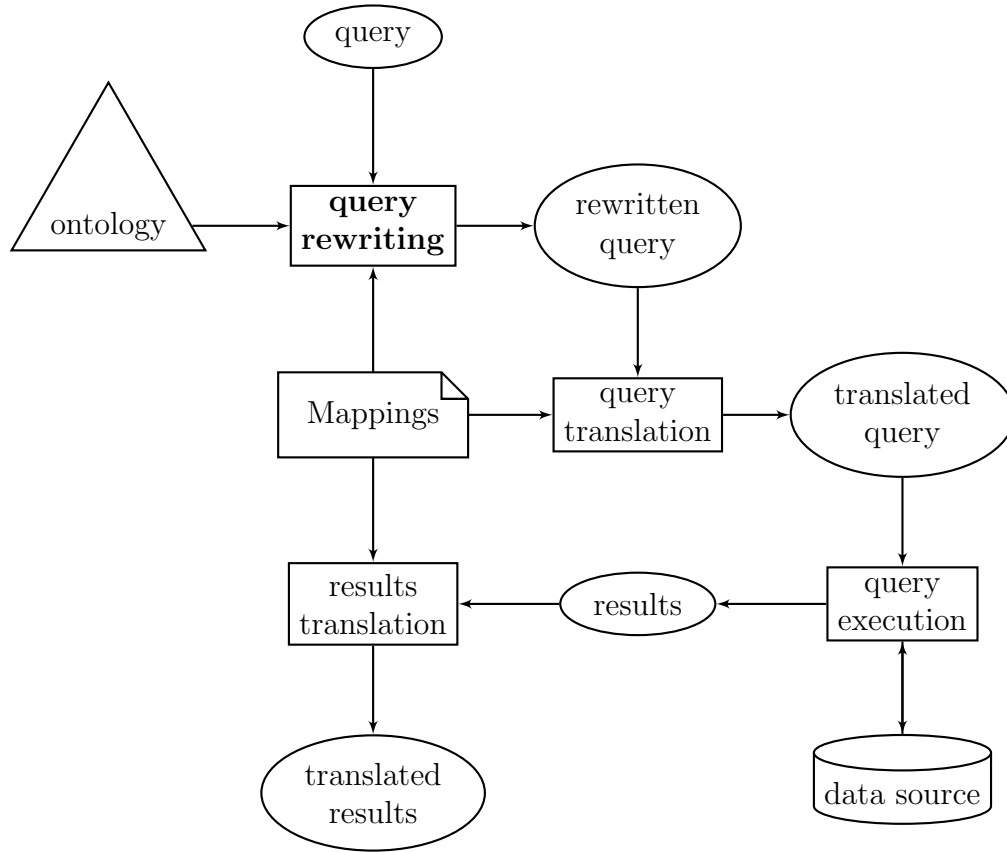


Figure 4.1: OBDA process with mapping-based query rewriting.

Organization of the United Nations (FAO)<sup>1</sup> in the context of the NeOn project<sup>2</sup>. These mappings were used to populate a set of fisheries ontologies [Caracciolo et al., 2010]. These ontologies<sup>3</sup> were developed for use within the Fish Stock Depletion Assessment System (FSDAS). Table 4.1 shows the ontology coverage provided by R2O (RDB2RDF) mappings in these test cases. Each row contains the information regarding a FAO test case, as a group composed by an ontology and a set of mappings. The columns show the number of elements, the number of mappings and the percentage of elements covered by the mappings with respect to concepts, object properties and datatype properties on the ontology respec-

<sup>1</sup>The test cases enumerating the specific elements in table 4.1 can be found at the URL <http://purl.org/net/jmora/mappings/fao/analysis>

<sup>2</sup><http://www.neon-project.org>

<sup>3</sup><http://www.fao.org/aims/neon.jsp>

## 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

tively. The last column displays the percentage of the ontology that is covered. As can be seen, the elements mapped with the databases are less than one third of those present in the ontologies, far from covering the whole ontology.

Group	Ontology Concepts			Object Properties			Datatype Properties			Total
	Num.	Mapped	Coverage	Num.	Map.	Coverage	Num.	Map.	Coverage	
1	7	5	28.57%	15	14	6.67%	126	100	20.63%	19.59%
2	3	1	66.67%	1	0	100.00%	8	2	75.00%	75.00%
3	6	0	100.00%	25	9	64.00%	30	10	66.67%	68.85%
4	5	0	100.00%	20	8	60.00%	90	32	64.44%	65.22%
5	7	3	57.14%	0	0	0.00%	104	70	32.69%	34.23%
6	5	0	100.00%	20	8	60.00%	90	32	64.44%	65.22%
7	3	1	66.67%	1	0	100.00%	8	2	75.00%	75.00%
8	16	15	6.25%	0	0	0.00%	201	198	1.49%	1.84%
9	16	15	6.25%	0	0	0.00%	201	198	1.49%	1.84%
Total	68	40	41.18%	82	39	52.44%	858	644	24.94%	28.27%

Table 4.1: Test cases in FAO, ontology coverage by RDB2RDF mappings. Each group consists of an ontology and a set of mappings.

Given a UCQ, removing the clauses that contain some predicate that is not mapped produces another UCQ that is equivalent with respect to the answers obtained, since those clauses could not be translated anyway. Thus we focus on optimising the rewriting process, with respect to the size of the rewritten query and time to obtain this query, in a context where there are several equivalent  $q'_\Sigma \dots q_\Sigma^{(n)}$ , obtaining the queries with the smallest size while still equivalent.

### 4.1 Preliminaries

In this section we introduce some notions that will be useful to explain the contents of the chapter as well as future chapters.

**Definition 1.**  $\varphi_{\mathcal{J}}(\gamma)$  *Contributions of a clause  $\gamma$  in an OBDA system  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$ . Let  $p$  be the predicate in the head of  $\gamma$ , we define the contributions of  $\gamma$  on  $\mathcal{J}$  as the set:*

$$\varphi_{\mathcal{J}}(\gamma) = \{\vec{\alpha} \mid \exists \mu. (\mathcal{J} \models \mu(\text{body}(\gamma))) \wedge (\mu(\text{head}(\gamma)) = p(\vec{\alpha}))\}$$

Where  $\mu$  is a substitution of the variables in  $\gamma$  with the constants  $\vec{\alpha}$ . Please note



---

that  $\mathcal{J} \models \mu(\text{body}(\gamma))$  means that  $(\Sigma \cup \mathcal{M} \cup D) \models \mu(\text{body}(\gamma))$ , i.e. the values for the contribution may be implied by other clauses in  $\Sigma$ .

**Definition 2.**  $v_{\mathcal{J}}(p)$  **Values for a predicate  $p$  on  $\mathcal{J}$ .** For a given OBDA system  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$ , we define as the values for a predicate  $p$  in  $\mathcal{J}$  as the set:

$$v_{\mathcal{J}}(p) = \{\vec{\alpha} \mid \mathcal{J} \models p(\vec{\alpha})\}$$

Moreover, the values for a predicate  $p$  on  $\mathcal{J}$ , i.e.  $v_{\mathcal{J}}(p)$ , are divided into the extensional values  $v_{\mathcal{J}}^e(p)$  and the intensional values  $v_{\mathcal{J}}^i(p)$ , so that  $v_{\mathcal{J}}(p) = v_{\mathcal{J}}^e(p) \cup v_{\mathcal{J}}^i(p)$ , defined as:

$$v_{\mathcal{J}}^e(p) = \{\vec{\alpha} \mid \langle \mathcal{M}, D \rangle \models p(\vec{\alpha})\}$$

$$v_{\mathcal{J}}^i(p) = \{\vec{\alpha} \mid \exists \gamma, \mu. \gamma \in \Sigma \wedge \mu(\text{head}(\gamma)) = p(\vec{\alpha}) \wedge \vec{\alpha} \in \varphi_{\mathcal{J}}(\gamma)\}$$

Where  $\mu$  is the most general unifier (MGU) applied to  $\text{head}(\gamma)$ , from the variables in  $\gamma$  to the constants in  $\vec{\alpha}$ .

Intuitively, the intensional values for a predicate  $p$  are the contributions of the clauses where  $p$  is in the head, while the contributions of a clause are a projection and selection of the values for the predicates in its body. Note that the intersection between the extensional and the intensional values of a predicate may not be empty.

Let us consider an example composed uniquely of the following two clauses:

- $\gamma_1$ : `Professor(x) :- AssociateProfessor(x)`, and
- $\gamma_2$ : `Professor(x) :- AssistantProfessor(x)`.

We may have the following facts in our ABox ( $\mathcal{A} = \langle \mathcal{M}, D \rangle$ ):

- `AssociateProfessor(Al)`, `AssociateProfessor(Ben)`,
- `AssistantProfessor(Cal)`,
- `Professor(Al)`, `Professor(Cal)`, and `Professor(Don)`.

In this example:

## 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

1. The extensional values for **Professor** on  $\mathcal{J}$  are:  $v_{\mathcal{J}}^e(Professor) = \mathbf{Al}, \mathbf{Cal}$  and **Don**.
2. The intensional values for **Professor** on  $\mathcal{J}$  are:  $v_{\mathcal{J}}^i(Professor) = \mathbf{Al}, \mathbf{Ben}$  and **Cal**.
3. The values for the predicate  $v_{\mathcal{J}}(Professor)$  are the union of both sets: **Al**, **Ben**, **Cal** and **Don**.
4. The contributions from clause  $\gamma_1$ , i.e.  $\varphi_{\mathcal{J}}(\gamma_1)$ , are **Al** and **Ben**.
5. The contributions from clause  $\gamma_2$ , i.e.  $\varphi_{\mathcal{J}}(\gamma_2)$ , are just one: **Cal**.

Note that the intensional values of a predicate  $p$  are defined according to the contributions of the clauses that have that  $p$  in their heads. At the same time, the contributions of a clause depend on the values of the predicates in its body. More precisely, we can say that the contributions of a clause  $\gamma$  are a *projection* to the variables in the head of the clause of the *product* of the values of the predicates in the body of that clause  $\gamma$ . This consideration can be made from a general mathematical perspective or from a relational algebra perspective and it is just the normal semantics for Datalog and logic [Abiteboul et al., 1995]. However, this is worth noting for a more clear understanding of how the contributions of clauses and the values for predicates (both in an OBDA system) relate to each other.

## 4.2 The OBDA algorithm for kyrie

In the following sections we describe the algorithm for our system kyrie, highlighting the optimisations that have been performed.

### 4.2.1 Intuitive description

The algorithm in kyrie is divided in several stages, what provides a modular approach to query rewriting. One of the most noteworthy benefits in this modular approach is its maintainability. A good example of this maintainability is in the different versions of the algorithm that we will see throughout their respective

---

chapters. In figure 4.2 we can see the stages in the kyrie algorithm. These stages are:

**Convert to Datalog.** The ontology is converted to Datalog. The query is specified in Datalog or converted to Datalog. Both are joined into a Datalog program.

**Test reachability.** Only clauses that lead to reachable mapped predicates are preserved.

**Remove functions.** Functional terms are removed by removing the clauses that contain them.

**Remove non-mapped predicates.** Non-mapped predicates are removed by removing the clauses that contain them.

**Greedy unfold.** The obtained Datalog program is unfolded greedily.

**Naïve unfold.** The obtained Datalog program is unfolded naïvely.

**Optimise.** The rewritten query is optimised by condensing the obtained clauses and removing subsumed clauses and unreachable clauses.

The algorithm in kyrie is based on the algorithm implemented in REQUIEM. The main differences with the REQUIEM algorithm are highlighted in a different background color. The stages in which the algorithm is decomposed produce transformations in the logic program obtained from the ontology and in the query, up to obtaining a rewriting. Most of the stages are based on the use of saturating a set of clauses using resolution with free selection (RFS).

### 4.2.2 A running example

Consider the following minimal ontology:

<i>Student</i>	$\sqsubseteq$	<i>Person</i>	<i>UndergradStudent</i>	$\sqsubseteq$	<i>Student</i>
<i>GradStudent</i>	$\sqsubseteq$	<i>Student</i>	<i>MasterStudent</i>	$\sqsubseteq$	<i>GradStudent</i>
<i>PostDoc</i>	$\sqsubseteq$	<i>Person</i>	<i>PhDStudent</i>	$\sqsubseteq$	<i>GradStudent</i>
<i>Professor</i>	$\sqsubseteq$	<i>Person</i>	<i>Bachelor</i>	$\sqsubseteq$	<i>UndergradStudent</i>

#### 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

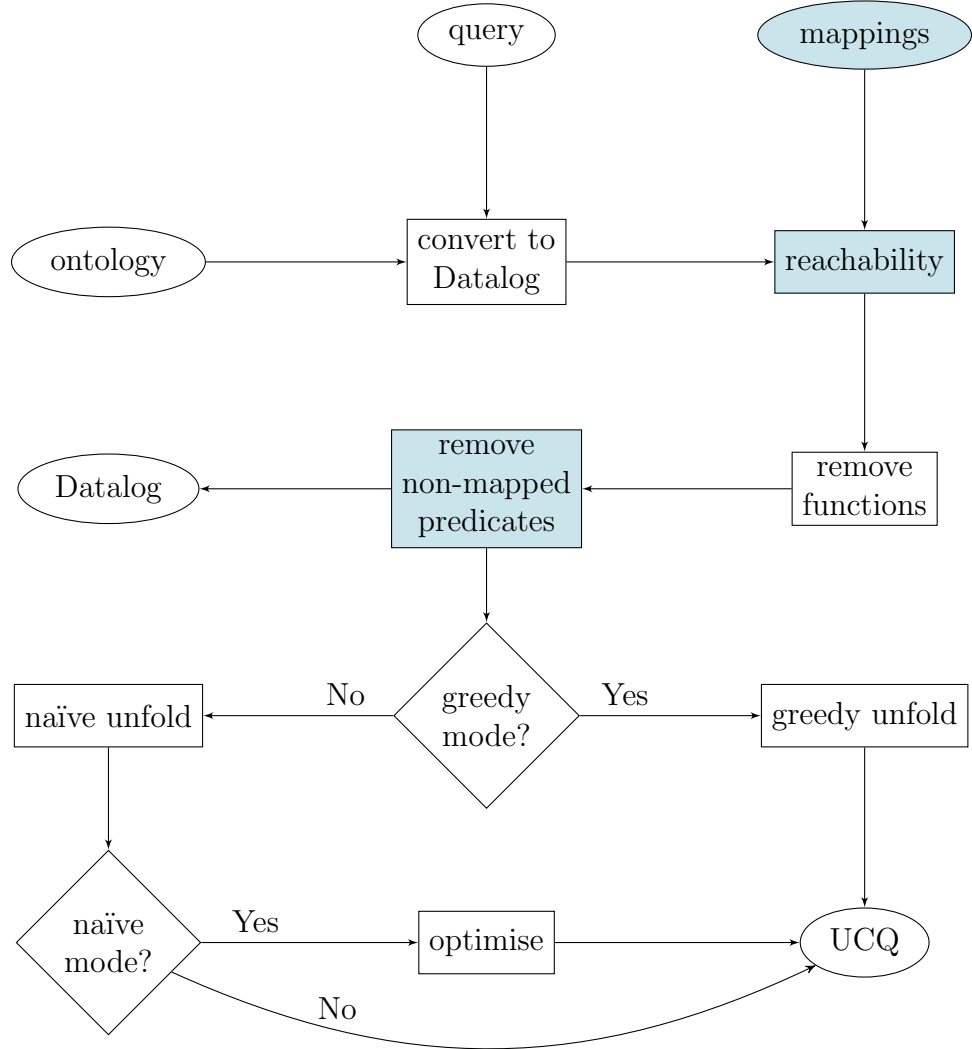


Figure 4.2: Stages in the kyrie algorithm, in a different color the modifications explained.

The set of mapped predicates: *Student*, *Bachelor*, *MasterStudent*, and *PhDStudent*.

And the query:  $Q(x) \leftarrow \text{Person}(x)$

In this case there are two possible UCQs that we may expect as the result, depending on the assumptions that are done on the OBDA system. First, under the assumption of ABox completeness, the mappings for *Student* do already provide the instances for *Bachelor*, *MasterStudent* and *PhDStudent*. Therefore

---

the result should be simply  $Q(x) \leftarrow \text{Student}(x)$ .

On the contrary, if this assumption cannot be made, then the UCQ should also query for these predicates, and the UCQ would result in the following:

$$\begin{aligned} Q(x) &\leftarrow \text{Student}(x) & Q(x) &\leftarrow \text{MasterStudent}(x) \\ Q(x) &\leftarrow \text{Bachelor}(x) & Q(x) &\leftarrow \text{PhDStudent}(x) \end{aligned}$$

We can see that the original predicate in the query (*Person*) is not present in any of the final rewriting possibilities. We can also see that in both cases we can disregard predicates *Professor* and *PostDoc* because they do not lead to any mapping and therefore they produce no results. If we have full branches of predicates that do not lead to any mappings we can disregard them in any case. This is normal when importing or reusing ontologies which may model a wider domain than the information available for the mappings. The domain may also be defined with a different granularity. Depending on the completeness of these mappings we may require to obtain all mapped predicates or only the most general ones. For instance in the first example only *Student* needs to be preserved, since all the individuals that are students (and thus people) are provided by this mapping. However, if we cannot guarantee that all individuals in *MasterStudent* are included in the mapping for *Student* then we will have to query for master students as well.

We will see in more detail the conditions for the elimination of non-mapped predicates. Intuitively non recursive predicates can be removed in any case by bypassing them or simply removing them. We have seen this with *UndergradStudent*, which was bypassed, and *PostDoc* which is not mapped and does not lead to any mappings, so it can be safely ignored.

If there is any cycle we will need to preserve one predicate for each one of the cycles to keep some potential answers. This preserved predicate would preferably be mapped but some cycles may be composed entirely by non-mapped predicates.

### 4.2.3 Pseudocode of the algorithm

Algorithm 4.1 shows the general algorithm of the system as a whole and following algorithms focus on specific procedures in this general algorithm. The

#### 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

algorithms use several functions, for example *count*, defined as  $\text{count}(p, \Theta) = \text{card}(\{\theta \in \Theta \mid p \in \theta\})$ , i.e. *count* applied to a predicate  $p$  and a set of cycles  $\Theta$  counts in how many cycles the predicate  $p$  appears. All other functions have the usual meaning, for instance “pop” removes an element from a set and returns its value. We can observe that the general algorithm for *kyrie* is very similar to the one in *REQUIEM*, with the following differences:

---

**Algorithm 4.1:** First *kyrie* algorithm

---

**Input:** Conjunctive query  $q$ ,  $\mathcal{ELHIO}$  ontology  $\mathcal{O}$ , mappings  $\mathcal{M}$ , working mode  $mode$

**Output:** Rewritten query  $q_\Sigma$

```

1  $\Sigma = \Xi(\mathcal{O}) \cup \{q\}$ 
2  $\Sigma = \text{reachableMaps}(q, \mathcal{M})$ 
3  $q_\Sigma = \text{requiemSaturate}(\Sigma, sfRQR, mode)$ 
4  $\Gamma = \{\gamma \in q_\Sigma \mid \}$ 
5  $\mathcal{M}' = \text{reducedRecursiveSet}(\Gamma)$ 
6  $q_\Sigma = \text{requiemSaturate}(q_\Sigma, sfMaps(\mathcal{M} \cup \mathcal{M}'), mode)$ 
7 if  $mode = greedy$  then
8    $P = IDBpredicates(q_\Sigma)$ 
9   for  $p \in P$  do
10     $q_\Sigma = \text{requiemSaturate}(q_\Sigma, sfUnfoldGreedy(p), greedy)$ 
11  end
12 else
13    $q_\Sigma = \text{requiemSaturate}(q_\Sigma, sfUnfoldNaive, naive)$ 
14 end
15 if  $mode \neq naive$  then
16    $q_\Sigma = \text{subsumptionCheck}(\text{pruneAUX}(\text{reachableMaps}(q_\Sigma, \mathcal{M})))$ 
17    $q_\Sigma = \text{map}(\text{condensate}, q_\Sigma)$ 
18 end
19 return  $q_\Sigma$ 

```

---

The first difference is that reachability tests consider mappings too. This reachability test does not simply retrieve the set of clauses that can be reached from the query, but also considers which clauses lead to some useful value. E.g. clauses that lead to no mappings do not lead to any result and can thus be removed. Two variants or working modes are proposed: H (reach to highest) and L (reach to lowest). The difference between both of them strives in the stopping

---

**Algorithm 4.2:** Find reduced sets of recursive predicates:  
**reducedRecursiveSet**

---

**Input:** Datalog program  $\Sigma$

```

1  $\Theta_\Sigma = \text{loopsIn}(\Sigma)$ 
2  $P_r = \emptyset$ 
3 while  $\Theta_\Sigma \neq \emptyset$  do
4    $goal = \max(\{count(p_j, \Theta_\Sigma) \mid p_j \in \Theta_\Sigma \wedge \neg auxiliary(p_j)\})$ 
5    $P_c = \{p_i \in \Theta_\Sigma \mid count(p_i, \Theta_\Sigma) = goal\}$ 
6   if  $P_c = \emptyset$  then
7      $goal = \max(\{count(p_j, \Theta_\Sigma) \mid p_j \in \Theta_\Sigma\})$ 
8      $P_c = \{p_i \in \Theta_\Sigma \mid count(p_i, \Theta_\Sigma) = goal\}$ 
9   end
10   $p = \text{randomSelect}(P_c)$ 
11   $P_r = P_r \cup \{p\}$ 
12   $\Theta_\Sigma = \Theta_\Sigma \setminus \{\theta \in \Theta_\Sigma \mid p \in \theta\}$ 
13 end
14 return  $P_r$ 

```

---



---

**Algorithm 4.3:** kyrie strict reachability algorithm (**reachable**) (variant H)

---

**Input:** Root predicate  $q$ , set of clauses  $\Gamma$ , mappings  $\mathcal{M}$   
**Output:** set of clauses  $\Gamma$

```

1  $reachable = \{q\}$ 
2  $pending = \{q\}$ 
3 while  $pending \neq \emptyset$  do
4    $p = \text{pop}(pending)$ 
5   if  $p \notin reachable \wedge p \notin \mathcal{M}$  then
6      $reachable = reachable \cup \{p\}$ 
7      $pending = pending \cup \{p' \mid \exists \gamma \in \Gamma. p' \in body(\gamma) \wedge p \in head(\gamma)\}$ 
8   end
9 return  $\{\gamma \in \Gamma \mid head(\gamma) \in reachable\}$ 

```

---

#### 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---



---

**Algorithm 4.4:** kyrie inclusive reachability algorithm (`reachable`) (variant L)

---

**Input:** Root predicate  $q$ , set of clauses  $\Gamma$ , mappings  $\mathcal{M}$   
**Output:** set of clauses  $\Gamma$

```

1 reachable = { $q$ }
2 instantiateable = { $m \in \mathcal{M}$ }
3 rps = ips = 0
4 while  $\text{card}(\text{reachable}) \neq \text{rps} \vee \text{card}(\text{instantiateable}) \neq \text{ips}$  do
5   rps =  $\text{card}(\text{reachable})$ 
6   ips =  $\text{card}(\text{instantiateable})$ 
7   for  $\gamma \in \Gamma$  do
8     if  $\text{head}(\gamma) \in \text{reachable}$  then
9        $\text{reachable} = \text{reachable} \cup \{p \in \text{body}(\gamma)\}$ 
10    if  $\forall p \in \text{body}(\gamma). p \in \text{instantiateable}$  then
11       $\text{instantiateable.add}(\text{head}(\gamma))$ 
12    end
13 end
14 return { $\gamma \in \Gamma \mid \text{head}(\gamma) \in \text{reachable} \wedge \forall p \in \text{body}(\gamma). p \in \text{instantiateable}$ }
```

---



---

**Algorithm 4.5:** Selection algorithm in `sfMaps` (`sfMaps.select`)

---

**Input:** Clause  $\gamma$   
**Output:** Specific atoms in  $\gamma$  are marked selected

```

1 check = True
2 foreach  $p \in \text{body}(\gamma)$  do
3   if  $p \notin \text{mappedPredicates}(\mathcal{M})$  then
4      $\text{select}(p)$ 
5     check = False
6   end
7 end
8 if  $\text{head}(\gamma) \notin \text{mappedPredicates}(\mathcal{M}) \wedge \text{check}$  then
9    $\text{select}(\text{head}(\gamma))$ 
10 end
```

---



---

condition. The first variant retrieves all clauses that lead to some mapping. The second variant retrieves all clauses that lead to some mapping up to the first mapping, ignoring mappings that are contained in this mapping, according to the TBox. The assumption in the second variant is that mappings are complete with respect to the answers that they provide, i.e. ABox completeness. In such a case each concept should be mapped to all its individuals, i.e. the answers corresponding to that concept. Intuitively, given such completeness in the mappings, the answers for a Datalog program are complete and the reachability test preserves the minimal number of predicates and clauses when the reachability tree built from the query satisfies the condition that a predicate is leave in the tree if and only if it is mapped. The behaviour of the algorithm is parametrized to work with both types of mappings, those that fulfil the completeness assumption and those that do not necessarily fulfil it.

The second difference is an additional inference stage after saturation. After saturation we obtain a Datalog program that is already reduced with respect to the size of the Datalog program that would have been obtained from REQUIEM due to the reachability test that considers mappings. This Datalog program is reduced again by removing predicates that are not mapped but that had to be included, since they were present in some clause in the path from the query predicates to the mapped predicates. With this inference step we perform the inferences that are necessary to bypass this path and avoid these predicates, as we will see in section 4.3.2.

After the unfolding, if the working mode is not `naïve` then an additional pruning stage is performed (both in REQUIEM and `kyrie`). The difference in `kyrie` is that the reachability test considers mappings again, thus being able to reduce the size of the rewriting further.

The differences in the algorithm may seem minimal. However, the differences in performance are significant, as we will see in chapter 7. Intuitively, we can see that the same rewritten queries obtained in `kyrie` could be obtained with the REQUIEM algorithm appending a reachability test after the rewriting process. This would effectively eliminate clauses or conjunctive queries that contain predicates that are not mapped and do not lead to other clauses, producing the same rewriting as in `kyrie`. Applying these optimisations after the original rewriting

## 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

process would lead to query rewriting times that are longer than the original query rewriting time of REQUIEM. Pruning the results in previous stages reduces the size of the program that is being processed and thus reduces processing times along the rewriting process. This allows producing queries that are shorter taking also shorter times, as we will show in chapter 7.

### 4.3 Optimisations

In this section we describe in more detail the mapping-based optimisations that can be applied to query rewriting algorithms. In particular, we focus on how these optimisations work in the kyrie system, which at this point differs from REQUIEM mainly on these optimisations.

#### 4.3.1 Mapping-based reachability

Reachability usually refers to the predicates in the ontology that are reachable (in a backward chaining fashion) from some given query  $q$ . Predicates that are reachable can potentially participate in the answers for  $q$ . When considering reachability and the description of some mappings we are going to focus on the reachability of the mappings and thus the instances (results) provided by them. The reason for this is that predicates that are not mapped cannot provide any results in an OBDA scenario since results are actually provided by the evaluation of the mappings. If a predicate is not mapped then any clause containing that predicate will not participate in the answers of the query.

We consider reachability in two different manners depending on the type of mappings that are considered for the system, i.e. whether they are complete or not. We consider a mapping for a concept to be complete if it provides all individuals for a concept and its subconcepts.

- If mappings are complete (corresponding to a virtual complete ABox) then the set of individuals  $i$  provided by any mapping  $m_i$  that is reachable from another one  $m_j$  will be contained ( $i \subseteq j$ ) in the set of individuals  $j$  provided by the latter mapping  $m_j$ . Therefore, the search for reachable mappings ends as soon as a mapped predicate is found.

- 
- If mappings are not guaranteed to be complete, then all reachable mappings could potentially add answers to the query and all of them should be included. Therefore the reachability should explore clauses (exactly) up to the last mapped predicate.

These two different ways in which mappings can be used are respectively explained in algorithms 4.3 and 4.4. Intuitively, the reachability testing algorithm constructs a tree of reachable predicates in a backward chaining fashion. The root of this tree is the query predicate, and it expands as we visit the predicates that appear in the body of clauses containing previously visited predicates. Considering such a tree, both variants of the reachability testing algorithm will still preserve some predicates that are not mapped. The H (to the highest, more strict) variant of the reachability test will preserve non-mapped predicates between the query and the first mapped predicate. The L (to the lowest, more inclusive) variant will preserve the predicates between the query and the last mapped predicate. These predicates need to be preserved for the next resolution stage, and will be eliminated afterwards, as explained in the next section.

### 4.3.2 Resolution with mappings

Normally, query rewriting systems may produce a Datalog program or a UCQ. This Datalog program will generally include predicates that are not mapped, despite the removal of some of these predicates with the reachability test, in the case of kyrie. These predicates are preserved due to the clauses that they participate in and the inferences that are possible with those clauses. Therefore, we can intuitively see that if we perform those inferences then we will be able to remove the clauses that participate in them, as they will be redundant. By removing these clauses, the predicates that are not mapped will be removed with them.

To do this, we use a selection function such that (1) body atoms are selected if and only if they are not mapped and (2) head atoms are selected if they are not mapped and no body predicate is selected. This resolution could generate an infinite set of clauses if the predicates that are not mapped form a cycle and generate new clauses. To prevent these cycles we mark as mapped some predicate

## 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

for each cycle. This is done heuristically selecting first the predicates that are in several cycles to minimise the set of predicates that are marked as mapped. Predicates that are not auxiliary are prioritised to be preserved, as specified in algorithm 4.2. This keeps some predicates that are not mapped in the Datalog and prevents an infinite inference by breaking the cycles formed by non-mapped predicates.

For example, if the predicate **Person** is not mapped then we can perform the following inference:

$$\begin{array}{ll} Q(x) & \leftarrow \text{Person}(x) \\ \text{Person}(x) & \leftarrow \text{Student}(x) \\ \therefore Q(x) & \leftarrow \text{Student}(x) \end{array}$$

And after this inference we can remove both premises and keep the consequent if this has no impact on the reachability of other mapped predicates that may provide additional answers. Intuitively, we can consider for correctness the two steps involved in this stage. First, the deduced clauses are added, the deduction is correct and adding the deduced clauses is also correct according to [Pettorossi and Proietti, 1994] (R1 - Unfolding). Once these clauses have been added we can see that the values for the head predicate (**q**) in the first clause are the same as in the third plus the extensional values for **Person(x)**. However, if **Person** is not mapped then these values are the empty set and thus the third clause provides the same values as the first clause. Therefore, either one of those two clauses can be removed. If we remove the first one and do so for every clause containing **Person** in the body then we can remove all predicates with **Person** as the head predicate, according to [Pettorossi and Proietti, 1994] (R5 - Definition Elimination).

The efficiency gain for this kind of resolution is evaluated in section 7.3.

### 4.4 Proofs

In this section we provide proofs for the soundness and completeness of the previously described algorithms. For this we use definitions introduced in section 4.1.

**Proposition 1.** *Let  $\mathcal{A}$  be an ABox,  $\Sigma$  and  $\Sigma'$  be two sets of Datalog clauses such that  $\Sigma' = \Sigma \cup \{\gamma_r\}$  with  $p_r$  as the predicate in the head of  $\gamma_r$ . For the two*

---

corresponding OBDA systems  $\mathcal{J} \equiv \langle \Sigma, \mathcal{A} \rangle$  and  $\mathcal{J}' = \langle \Sigma', \mathcal{A} \rangle$  if  $\varphi_{\mathcal{J}'}(\gamma_r) \subseteq v_{\mathcal{J}}(p_r)$  then  $v_{\mathcal{J}}(p) = v_{\mathcal{J}'}(p)$  for every predicate  $p$  in  $\Sigma$ .

*Proof.* The values in  $\Sigma$  for every predicate  $p$  of  $\Sigma$  are  $v_{\Sigma}(p) = v_{\Sigma}^e(p) \cup v_{\Sigma}^i(p)$ . The extensional values  $v_{\Sigma}^e(p)$  depend by definition solely on  $\mathcal{A}$ , which suffers no changes, thus they remain unaltered. The intensional values  $v_{\Sigma}^i(p)$  depend by definition on the contributions of the sets of clauses  $\{\gamma \in \Sigma \mid \exists \mu. \text{head}(\gamma) = \mu(p(\vec{x}))\}$  where  $\mu$  is a MGU from the variables in  $\vec{x}$  to the terms in  $\gamma$ . The contributions for  $p_r$  do not change with the addition or removal of  $\gamma_r$  due to the condition in the proposition. Therefore, the values for  $p_r$  do not change. Inductively, the contributions for all other clauses and the values for all other predicates in  $\Sigma$  do not change.  $\square$

**Corollary 1.** *Maintaining the previous definitions for  $\mathcal{A}$ ,  $\Sigma$ ,  $\Sigma'$ ,  $\gamma_r$ ,  $\mathcal{J}$  and  $\mathcal{J}'$ , for any given query  $q$ ,  $\Phi_{\mathcal{J}}^q = \Phi_{\mathcal{J}'}^q$ .*

*Proof.* The values for the predicates in  $\mathcal{J}$  are proven to be preserved. The answers to the query  $q$  are equally preserved, i.e. the values for the head predicate of  $q$  in the context of  $\mathcal{J}$ .  $\square$

**Proposition 2.** *Let  $\Sigma_q$  and  $\Sigma'_q$  be two Datalog queries for the same OBDA system  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$ , let  $p$  be a predicate in  $\Sigma_q$  and  $\Sigma'_q$ , and let  $\Gamma$  be a set of clauses, such that:*

1.  $\Gamma \subseteq \Sigma_q$ ,
2.  $\Sigma'_q = \Sigma_q \setminus \Gamma$ ,
3. for every clause  $\gamma$  in  $\Sigma_q$  such that  $p$  appears in its body,  $\gamma \in \Gamma$ ,
4.  $p$  does not appear in any mapping assertion in  $\mathcal{M}$ , and
5.  $\Sigma_q$  is closed with respect to  $p$ , i.e. no resolution can be performed in  $\Sigma_q$  by unifying atoms with  $p$ .

*If these conditions hold, then the values for every predicate  $p_i$  in  $\Sigma'_q$  are equal to the values for that predicate  $p_i$  in  $\Sigma_q$  and both queries,  $\Sigma_q$  and  $\Sigma'_q$ , have the same certain answers.*

## 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

*Proof.* Condition 4 implies that  $p$  has only intensional values, i.e.  $v_j(p) = v_j^i(p)$  and  $v_j^e(p) = \emptyset$ . By definition, these intensional values correspond to the union of the contributions of the set of clauses  $\Gamma_a$  composed by the clauses in  $\Sigma_q$  that have  $p$  in their heads. Due to condition 5, for each  $\gamma_b \in \Gamma$ , i.e. containing  $p$  in its body, there is a set of clauses  $\Gamma_b$  generated through resolution where  $p$  has been unified with all the clauses containing  $p$  in its head, and that do not contain  $p$  in their bodies. In this context, for each  $\gamma_b$  the set of clauses  $\Gamma_b$  covers the contributions of  $\gamma_b$ , i.e.  $\varphi_j(\gamma_b) \subseteq v_{j \setminus \{\gamma_b\}}(\text{head}(\gamma_b))$ . Therefore, according to proposition 1, for every  $\gamma_b \in \Gamma$ ,  $\gamma_b$  has no impact on the values of the predicates in  $\Sigma_q$ . The clauses in  $\Gamma$  are the difference between  $\Sigma_q$  and  $\Sigma'_q$ , and none of them has impact on the values of the predicates in  $\Sigma_q$ , therefore they can be individually and recursively removed, obtaining  $\Sigma'_q$  from  $\Sigma_q$ , without modifying the values for the predicates in  $\Sigma_q$  nor the certain answers.  $\square$

Proposition 2 allows removing the clauses that contain predicates that are not mapped. The removal of the clauses that contain the predicates in the body is immediate from the proposition. The removal of the clauses that contain these predicates in the head is a consequence of applying a reachability test after it. The clauses that contain the predicates both in the body and the head may produce a cycle and cannot be removed, similarly at least one clause may need to be preserved for each cycle. This limitation with respect to the cycles is implicit in the condition 5, if the resolution selecting some predicate  $p$  is not finite, then that predicate needs to be preserved to preserve the certain answers of the query, even if it is not mapped. In such a case, the Datalog query cannot be unfolded into a UCQ.

### 4.5 Conclusions

We have seen that using information about mappings can help to produce queries that are better suited to obtain answers from these mappings. We will also see in section 7.3 that this is relevant for the performance of query rewriting. This is highly dependent on the number of predicates in the ontology that have a mapping. If all the predicates in the ontology are mapped then the impact of our

---

computation is going to be neutral or negative. This is rarely the case, in the real world: we have seen a set of examples or real world situations where the part of the ontology that is mapped is far less than one half of it. One of the reasons for this is that on the one hand ontologies are linked and reused among themselves to obtain some broader consensus in their semantics. On the other hand knowledge bases may be local to an expert system, addressing in a specific way the requirements for this system. This relative broadness or specificity influences the coverage of the knowledge for the predicates in the data source.

Considering mappings points at the relevance of considering the framework and additional information that can be available for query rewriting. When addressing complex problems a common technique is to divide them into smaller problems. This has been done effectively in the state of the art for the case of OBDA, being query rewriting and mapping translation two of these smaller problems. The limitation in the isolated analysis of subproblems is the likelihood of falling into local minima. When solutions are available for these subproblems, their integration may serve to find better solutions though a broader scope. We have seen how the scope for query rewriting can be broadened out by including only a small consideration about mappings, simply considering whether predicates in the ontology are mapped or not. Due to the simplicity of this consideration, obtaining this type of information is usually simple. A mapping definition should be available for any OBDA system and any mapping definition (GAV, LAV or GLAV) will refer to the set of predicates that are mapped. Therefore there are no additional requirements for this consideration and its applicability is in principle broad.

We have also introduced the property of redundancy irrelevance (proposition 1). This property holds when the expected result is the set of certain answers, as commonly expected in OBDA. The set semantics is a tacit assumption that is usually made in OBDA systems by describing the queries as FOL or Datalog programs. These queries are rewritten to SQL thanks to the FOL-reducibility property, which was discussed in section 2.2. However, in SQL and languages influenced by SQL (e.g. SPARQL, XQuery) we can usually find the `count` predicate. When using this predicate, the elimination of redundant solutions is not transparent and has implications. From a logic perspective, two individuals that

#### 4. QUERY REWRITING OPTIMISATIONS IN THE PRESENCE OF RDB2RDF MAPPINGS

---

are exactly equal represent the same individual, therefore only `distinct` individuals can be counted. This limits the applicability of OBDA query rewriting approaches in cases where counting repetitions is relevant.



## Chapter 5

# Engineering optimisations for query rewriting

In this chapter we propose a series of optimisations that can be performed during the rewriting process. As explained in chapter 3 we focus on the  $\mathcal{ELHIQ}$  expressiveness. By focusing on this very expressive logic we expect the optimisations performed here to be generalisable to a broad range of less expressive logics and possibly some more expressive logics. Despite this expressiveness, the efficiency of the algorithm obtained is comparable to those addressed at query rewriting with less expressive logics. These optimisations are implemented in the system *kyrie2*.

Query rewriting for first-order reducible logics is from its origin meant to be efficient, focusing on restricting the inference to the TBox — usually small in size when compared with the ABox — and producing a query for a relational database to obtain the answers on the ABox. Therefore, OBDA systems rely on using relational database technology, which is normally considered as more mature, stable and efficient. Queries are only restricted by the set of terms available in the TBox. Because of the combinatorial explosion that this represents (exemplified in section 8.1) and the arbitrary length of queries, materialization of all queries and answers is unfeasible. Queries have to be rewritten and resolved at run-time. In addition, query answering is often interactive, i.e. involving a human in the loop; thus reduced response times are desired and expected for

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

these systems.

Similarly to efficiency, the more expressiveness the better, since this is the main feature of a query rewriting system that includes an ontology when compared with the database that answers the rewritten query. However expressiveness comes at a cost in efficiency. Most expressive logics imply a greater computational complexity to be rewritten, sometimes requiring also a more expressive target language, for instance rewriting recursive Datalog programs into a finite UCQ is dependent on the ABox. In this context we propose a series of optimisations that are applicable to query rewriting in the context of the  $\mathcal{ELHIQ}^+$  expressiveness.

In our case, we reduce the query rewriting time by performing some of the tasks needed for the rewriting before any query is handled to the system. We also avoid the generation of subsumed clauses by the combination of some optimisations, which consist on checking clause subsumption as soon as possible and producing shorter clauses first. We produce shorter clauses first and reduce the time required to check clause subsumption by condensing clauses and ordering the clauses depending on their length. Additionally, clauses are separated into two groups, candidates for main premise and candidates for side premise, when possible, with an additional efficiency gain.

Analogously to the previous chapter, these optimisations change the general OBDA scenario from figure 1.1 to the one that we can see in figure 5.1. Again the changes in the whole picture are subtle, simply adding the preprocessing stage to the whole OBDA scenario. In this case there are also some changes inside the rewriting stage, as explained in figure 5.2. We can see that the mappings are not used this time, allowing for a better comparison of the impact of these optimisations without considering mappings. Since mappings are not used, the results obtained after applying these optimisations are the perfect rewriting [Calvanese et al., 2000] when possible. This means that the output is the same as previous approaches when the ontology is not beyond the expressiveness that approaches other than REQUIEM may handle. In those cases where a recursive Datalog may be obtained, as in REQUIEM, the strategy for the unfolding is different. REQUIEM produces a linear Datalog program, meaning that the recursive clauses have at most one intensional database (IDB) predicate in the body. However

---

in kyrie2 the unfolding goes further and minimizes the number of different IDB predicates that appear in the head of some clause, instead of those that appear in some body. This way less subqueries are produced whenever the query has to be fragmented into subqueries. The results obtained and the conclusions that can be derived from them will be evaluated and analysed in chapter 7.

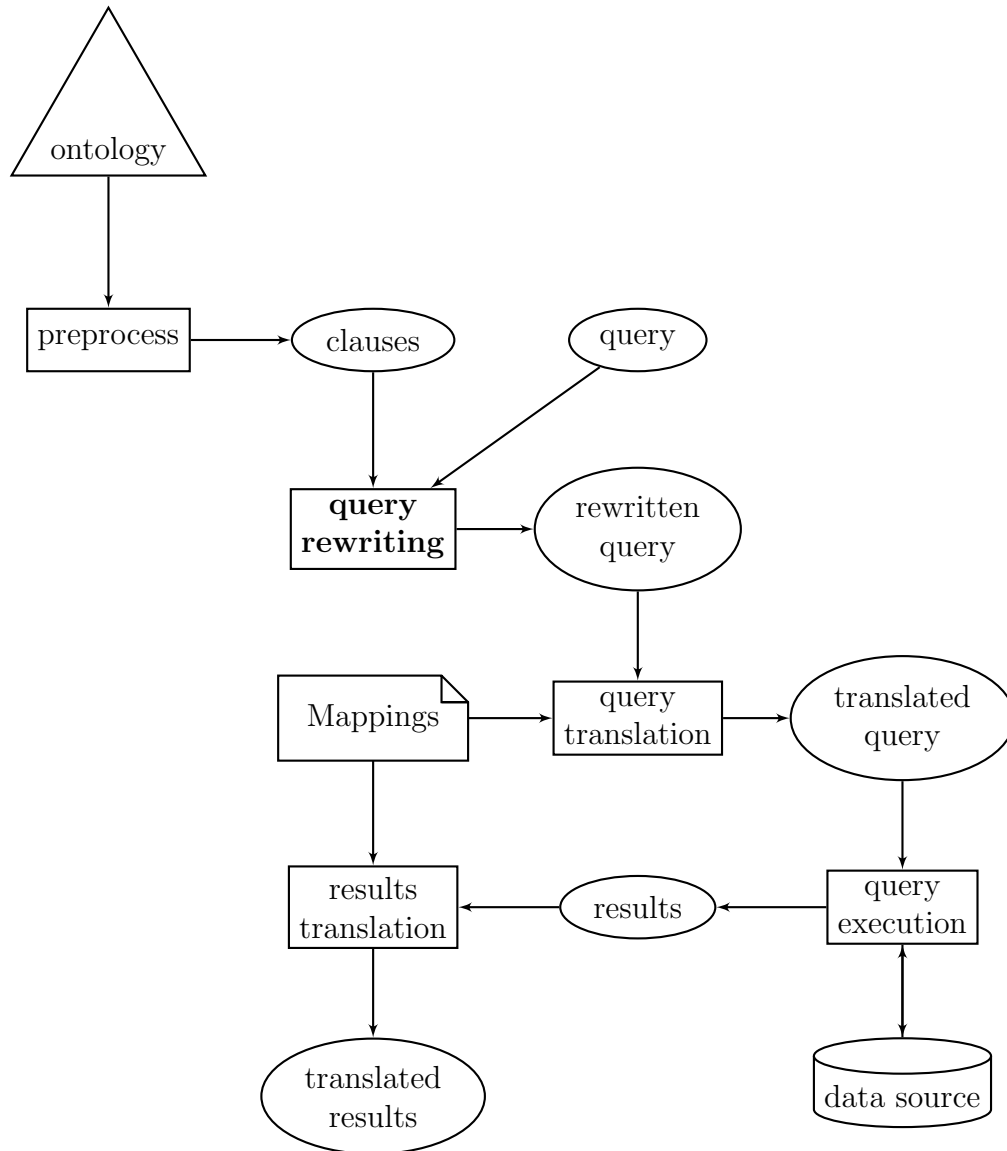


Figure 5.1: OBDA process with query rewriting and a preprocessing stage.

In the following sections we go through the different optimisations proposed.

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

We will provide a running example in section 5.1.2 and illustrate each of these optimisations with a conventional ontology in the domain of university studies. First, we consider the preprocessing in section 5.2.1. The subsumption check, used to remove subsumed atoms and clauses, is explained in section 5.2.2. The heuristic prioritization of some inferences is explained in section 5.2.3. Finally, in section 5.2.4 we explain how some searches for valid inferences can be constrained. For comparisons we focus especially on REQUIEM, which is the only approach that covers the  $\mathcal{ELHJO}$  expressiveness.

### 5.1 The OBDA algorithm for kyrie2

Here we describe the algorithm for our system kyrie2, highlighting the optimisations that have been performed and that are explained in the following sections.

#### 5.1.1 Intuitive Description

The REQUIEM algorithm is revised and optimised for kyrie2. As a consequence, the working modes (naïve, full subsumption and greedy) are dropped to unify the behaviour of the algorithm in an optimal mode. Therefore, the flow diagram, in figure 5.2 appears simplified by this aspect while new resolution stages are added to perform these optimisations.

The optimisations are based on a single fundamental principle: obtaining the rewritten query from the original query faster. Intuitively, this means performing less computations since the query arrives until the rewritten query is produced. An example of such type of optimisations is given by previous approaches, like Rapid or Nyaya. These approaches focus on avoiding the production of clauses that are subsumed by other clauses. Intuitively, we can see that subsumed clauses provide no additional answers nor partial answers, therefore those clauses can be discarded and its generation is unnecessary. In table 4 in [Chortaras et al., 2011] we can see that the time needed to remove subsumed clauses is greater than the time needed to generate the preliminary results for the most complex queries (check 4th and 5th queries in each group). Avoiding the generation of subsumed clauses reduces the time needed for the clause generation but more importantly

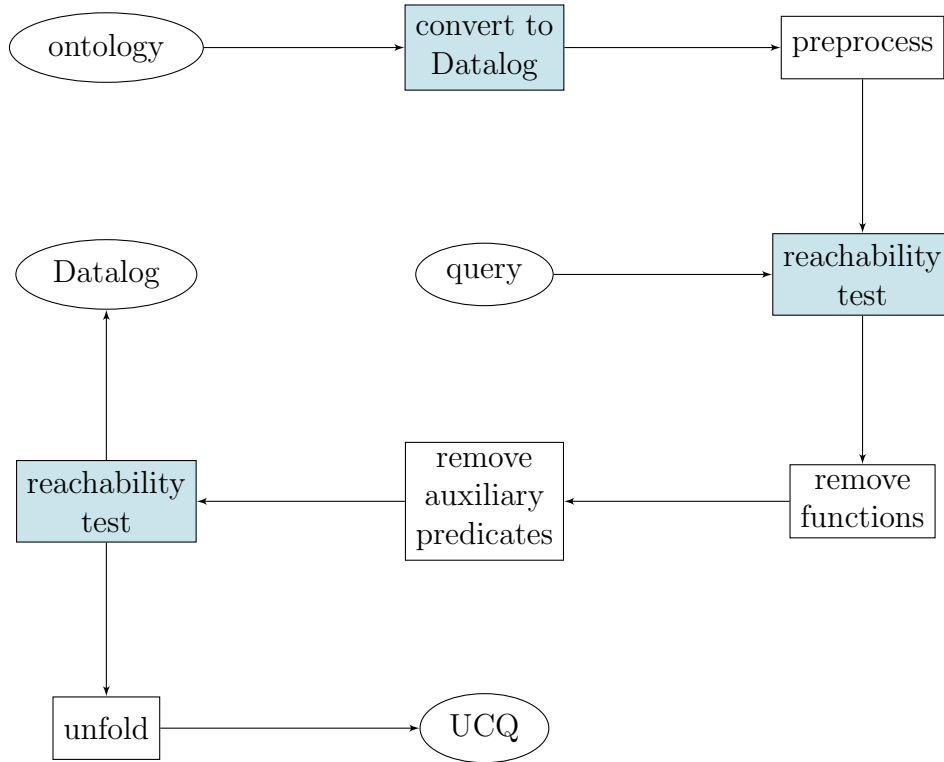


Figure 5.2: Stages in the kyrie2 algorithm, in a different color the parts left unchanged

reduces the time needed to check clause subsumption after the clauses have been generated.

One of the main optimisations in kyrie2 relies on early subsumption removal. Two subsumption check steps are performed for each query before any other processing. The first one (condensate) removes subsuming atoms in each one of the clauses. The second one removes subsumed clauses from the program. These subsumption checks are included in the resolution processes, as we will see in algorithm 5.3.

### 5.1.2 A running example

For the examples in this chapter we will extend the simple ontology used in the previous chapter. The new ontology is:

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

$Student \sqsubseteq Person$	$UndergradStudent \sqsubseteq Student$
$GradStudent \sqsubseteq Student$	$MasterStudent \sqsubseteq GradStudent$
$PostDoc \sqsubseteq Person$	$PhDStudent \sqsubseteq GradStudent$
$Professor \sqsubseteq Person$	$Bachelor \sqsubseteq UndergradStudent$
$\exists hasStudent^- \sqsubseteq Student$	$PotentialCourse \sqsubseteq Course$
<hr/>	
$\exists hasStudent.Student \sqsubseteq \exists hasProfessor.Professor$	
$\exists hasProfessor.Professor \sqsubseteq Course$	
$ImpartedCourse \sqsubseteq \exists hasStudent.Student$	

We will consider the example where the query used is:

$Q(?0) \leftarrow Course(?0), hasStudent(?0,?1), hasProfessor(?0,?2)$

We will follow the query rewriting process as described in figure 5.2, comparing the results of each stage in original algorithm with the results obtained when applying the optimisations. When the ontology is converted to Datalog (as described in section 2.3.1) we obtain the following set of clauses:

```

Person(?0) <- Student(?0)
ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
Student(f0(?0)) <- ImpartedCourse(?0)
hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
Professor(f1(?0)) <- AUX$0(?0)
hasProfessor(?0,f1(?0)) <- AUX$0(?0)
Student(?0) <- UndergradStudent(?0)
Course(?0) <- Professor(?1), hasProfessor(?0,?1)
Person(?0) <- Professor(?0)
Person(?0) <- PostDoc(?0)
UndergradStudent(?0) <- Bachelor(?0)
Student(?0) <- GradStudent(?0)
Student(?0) <- hasStudent(?1,?0)
Course(?0) <- PotentialCourse(?0)
GradStudent(?0) <- MasterStudent(?0)
GradStudent(?0) <- PhDStudent(?0)

```

---

First we will see how the process would be done without any optimisation. We start with a reachability test, obtaining the following logic program:

```

    Person(?0) <- Student(?0)
    ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
    Student(f0(?0)) <- ImpartedCourse(?0)
    hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
    AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
    Professor(f1(?0)) <- AUX$0(?0)
    hasProfessor(?0,f1(?0)) <- AUX$0(?0)
    Student(?0) <- UndergradStudent(?0)
    Course(?0) <- Professor(?1), hasProfessor(?0,?1)
    Person(?0) <- Professor(?0)
    Person(?0) <- PostDoc(?0)
    UndergradStudent(?0) <- Bachelor(?0)
    Student(?0) <- GradStudent(?0)
    Student(?0) <- hasStudent(?1,?0)
    Course(?0) <- PotentialCourse(?0)
    GradStudent(?0) <- MasterStudent(?0)
    GradStudent(?0) <- PhDStudent(?0)
    Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)

```

Then saturation is performed to obtain a Datalog program. The inferences that are performed are the following:

<pre>         ImpartedCourse(?0) &lt;- Student(?1), hasStudent(?0,?1)         hasStudent(?0,f0(?0)) &lt;- ImpartedCourse(?0) </pre>	
<hr/>	
<pre>         ∴ ImpartedCourse(?0) &lt;- ImpartedCourse(?0), Student(f0(?0)) </pre>	
<hr/>	
<pre>         hasStudent(?0,f0(?0)) &lt;- ImpartedCourse(?0) </pre>	

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

AUX\$0(?0) <- Student(?1), hasStudent(?0,?1)

---

∴ AUX\$0(?0) <- ImpartedCourse(?0), Student(f0(?0))

hasProfessor(?0,f1(?0)) <- AUX\$0(?0)

Course(?0) <- Professor(?1), hasProfessor(?0,?1)

---

∴ Course(?0) <- AUX\$0(?0), Professor(f1(?0))

hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)

Student(?0) <- hasStudent(?1,?0)

---

∴ Student(f0(?0)) <- ImpartedCourse(?0)

hasProfessor(?0,f1(?0)) <- AUX\$0(?0)

Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)

---

∴ Q(?0) <- AUX\$0(?0), Course(?0), hasStudent(?0,?1)

hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)

Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)

---

∴ Q(?0) <- Course(?0), ImpartedCourse(?0),

hasProfessor(?0,?1)

Student(f0(?0)) <- ImpartedCourse(?0)

AUX\$0(?0) <- ImpartedCourse(?0), Student(f0(?0))

---

∴ AUX\$0(?0) <- ImpartedCourse(?0)

Professor(f1(?0)) <- AUX\$0(?0)

Course(?0) <- AUX\$0(?0), Professor(f1(?0))

---

∴ Course(?0) <- AUX\$0(?0)

hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)

Q(?0) <- AUX\$0(?0), Course(?0), hasStudent(?0,?1)

---

∴ Q(?0) <- AUX\$0(?0), Course(?0), ImpartedCourse(?0)

hasProfessor(?0,f1(?0)) <- AUX\$0(?0)



---

```

Q(?0) <- Course(?0), ImpartedCourse(?0),
      hasProfessor(?0,?1)

```

---

```

∴ Q(?0) <- AUX$0(?0), Course(?0), ImpartedCourse(?0)

```

And the resulting Datalog program is the following:

```

ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
  AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
  Student(?0) <- UndergradStudent(?0)
  Course(?0) <- Professor(?1), hasProfessor(?0,?1)
UndergradStudent(?0) <- Bachelor(?0)
  Student(?0) <- GradStudent(?0)
  Student(?0) <- hasStudent(?1,?0)
  Course(?0) <- PotentialCourse(?0)
GradStudent(?0) <- MasterStudent(?0)
GradStudent(?0) <- PhDStudent(?0)
  Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)
  Q(?0) <- AUX$0(?0), Course(?0), hasStudent(?0,?1)
  Q(?0) <- Course(?0), ImpartedCourse(?0), hasProfessor(?0,?1)
  AUX$0(?0) <- ImpartedCourse(?0)
  Course(?0) <- AUX$0(?0)
  Q(?0) <- AUX$0(?0), Course(?0), ImpartedCourse(?0)

```

Then saturation is performed again over the Datalog program to obtain a UCQ. The inferences that are performed are the following:

```

ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
  Student(?0) <- UndergradStudent(?0)

```

---

```

∴ ImpartedCourse(?0) <- UndergradStudent(?1), hasStudent(?0,?1)

```

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
Student(?0) <- UndergradStudent(?0)


---


∴ AUX$0(?0) <- UndergradStudent(?1), hasStudent(?0,?1)
```

```
ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
Student(?0) <- GradStudent(?0)


---


∴ ImpartedCourse(?0) <- GradStudent(?1), hasStudent(?0,?1)
```

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
Student(?0) <- GradStudent(?0)


---


∴ AUX$0(?0) <- GradStudent(?1), hasStudent(?0,?1)
```

```
ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
Student(?0) <- hasStudent(?1,?0)


---


∴ ImpartedCourse(?0) <- hasStudent(?0,?1), hasStudent(?2,?1)
```

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
Student(?0) <- hasStudent(?1,?0)


---


∴ AUX$0(?0) <- hasStudent(?0,?1), hasStudent(?2,?1)
```

```
Course(?0) <- PotentialCourse(?0)
Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)


---


∴ Q(?0) <- PotentialCourse(?0), hasProfessor(?0,?1),
    hasStudent(?0,?2)
```

```
Course(?0) <- PotentialCourse(?0)
Q(?0) <- AUX$0(?0), Course(?0), hasStudent(?0,?1)


---


∴ Q(?0) <- AUX$0(?0), PotentialCourse(?0), hasStudent(?0,?1)
```

```
Course(?0) <- PotentialCourse(?0)
Q(?0) <- Course(?0), ImpartedCourse(?0), hasProfessor(?0,?1)


---


∴ Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0),
    hasProfessor(?0,?1)
```

---

```

Q(?0) <- AUX$0(?0), Course(?0), hasStudent(?0,?1)
AUX$0(?0) <- ImpartedCourse(?0)

```

---

```

∴ Q(?0) <- Course(?0), ImpartedCourse(?0), hasStudent(?0,?1)

```

---

```

Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)
Course(?0) <- AUX$0(?0)

```

---

```

∴ Q(?0) <- AUX$0(?0), hasProfessor(?0,?1), hasStudent(?0,?2)

```

---

```

Q(?0) <- AUX$0(?0), Course(?0), hasStudent(?0,?1)
Course(?0) <- AUX$0(?0)

```

---

```

∴ Q(?0) <- AUX$0(?0), hasStudent(?0,?1)

```

---

```

Q(?0) <- Course(?0), ImpartedCourse(?0), hasProfessor(?0,?1)
Course(?0) <- AUX$0(?0)

```

---

```

∴ Q(?0) <- AUX$0(?0), ImpartedCourse(?0), hasProfessor(?0,?1)

```

---

```

AUX$0(?0) <- ImpartedCourse(?0)
Q(?0) <- AUX$0(?0), Course(?0), ImpartedCourse(?0)

```

---

```

∴ Q(?0) <- Course(?0), ImpartedCourse(?0)

```

---

```

Course(?0) <- PotentialCourse(?0)
Q(?0) <- AUX$0(?0), Course(?0), ImpartedCourse(?0)

```

---

```

∴ Q(?0) <- AUX$0(?0), ImpartedCourse(?0), PotentialCourse(?0)

```

---

```

Course(?0) <- AUX$0(?0)
Q(?0) <- AUX$0(?0), Course(?0), ImpartedCourse(?0)

```

---

```

∴ Q(?0) <- AUX$0(?0), ImpartedCourse(?0)

```

---

```

UndergradStudent(?0) <- Bachelor(?0)
ImpartedCourse(?0) <- UndergradStudent(?1), hasStudent(?0,?1)

```

---

```

∴ ImpartedCourse(?0) <- Bachelor(?1), hasStudent(?0,?1)

```

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

UndergradStudent(?0) <- Bachelor(?0)

AUX\$0(?0) <- UndergradStudent(?1), hasStudent(?0,?1)

---

∴ AUX\$0(?0) <- Bachelor(?1), hasStudent(?0,?1)

GradStudent(?0) <- MasterStudent(?0)

ImpartedCourse(?0) <- GradStudent(?1), hasStudent(?0,?1)

---

∴ ImpartedCourse(?0) <- MasterStudent(?1), hasStudent(?0,?1)

GradStudent(?0) <- PhDStudent(?0)

ImpartedCourse(?0) <- GradStudent(?1), hasStudent(?0,?1)

---

∴ ImpartedCourse(?0) <- PhDStudent(?1), hasStudent(?0,?1)

GradStudent(?0) <- MasterStudent(?0)

AUX\$0(?0) <- GradStudent(?1), hasStudent(?0,?1)

---

∴ AUX\$0(?0) <- MasterStudent(?1), hasStudent(?0,?1)

GradStudent(?0) <- PhDStudent(?0)

AUX\$0(?0) <- GradStudent(?1), hasStudent(?0,?1)

---

∴ AUX\$0(?0) <- PhDStudent(?1), hasStudent(?0,?1)

AUX\$0(?0) <- ImpartedCourse(?0)

Q(?0) <- AUX\$0(?0), PotentialCourse(?0), hasStudent(?0,?1)

---

∴ Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0),  
hasStudent(?0,?1)

Course(?0) <- PotentialCourse(?0)

Q(?0) <- Course(?0), ImpartedCourse(?0), hasStudent(?0,?1)

---

∴ Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0),  
hasStudent(?0,?1)

Course(?0) <- AUX\$0(?0)

Q(?0) <- Course(?0), ImpartedCourse(?0), hasStudent(?0,?1)

---

∴ Q(?0) <- AUX\$0(?0), ImpartedCourse(?0), hasStudent(?0,?1)

---

```

AUX$0(?0) <- ImpartedCourse(?0)
Q(?0) <- AUX$0(?0), hasProfessor(?0,?1), hasStudent(?0,?2)

```

---

```

∴ Q(?0) <- ImpartedCourse(?0), hasProfessor(?0,?1),
    hasStudent(?0,?2)

```

```

AUX$0(?0) <- ImpartedCourse(?0)
Q(?0) <- AUX$0(?0), hasStudent(?0,?1)

```

---

```

∴ Q(?0) <- ImpartedCourse(?0), hasStudent(?0,?1)

```

```

AUX$0(?0) <- ImpartedCourse(?0)
Q(?0) <- AUX$0(?0), ImpartedCourse(?0), hasProfessor(?0,?1)

```

---

```

∴ Q(?0) <- ImpartedCourse(?0), hasProfessor(?0,?1)

```

```

Course(?0) <- PotentialCourse(?0)
Q(?0) <- Course(?0), ImpartedCourse(?0)

```

---

```

∴ Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0)

```

```

Course(?0) <- AUX$0(?0)
Q(?0) <- Course(?0), ImpartedCourse(?0)

```

---

```

∴ Q(?0) <- AUX$0(?0), ImpartedCourse(?0)

```

```

AUX$0(?0) <- ImpartedCourse(?0)
Q(?0) <- AUX$0(?0), ImpartedCourse(?0), PotentialCourse(?0)

```

---

```

∴ Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0)

```

```

AUX$0(?0) <- ImpartedCourse(?0)
Q(?0) <- AUX$0(?0), ImpartedCourse(?0)

```

---

```

∴ Q(?0) <- ImpartedCourse(?0)

```

```

AUX$0(?0) <- ImpartedCourse(?0)
Q(?0) <- AUX$0(?0), ImpartedCourse(?0), hasStudent(?0,?1)

```

---

```

∴ Q(?0) <- ImpartedCourse(?0), hasStudent(?0,?1)

```

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

And the resulting UCQ is the following:

```
ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
  AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
  Course(?0) <- Professor(?1), hasProfessor(?0,?1)
    Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)
    Q(?0) <- AUX$0(?0), Course(?0), hasStudent(?0,?1)
    Q(?0) <- Course(?0), ImpartedCourse(?0), hasProfessor(?0,?1)
    Q(?0) <- AUX$0(?0), Course(?0), ImpartedCourse(?0)
ImpartedCourse(?0) <- UndergradStudent(?1), hasStudent(?0,?1)
  AUX$0(?0) <- UndergradStudent(?1), hasStudent(?0,?1)
ImpartedCourse(?0) <- GradStudent(?1), hasStudent(?0,?1)
  AUX$0(?0) <- GradStudent(?1), hasStudent(?0,?1)
ImpartedCourse(?0) <- hasStudent(?0,?1), hasStudent(?2,?1)
  AUX$0(?0) <- hasStudent(?0,?1), hasStudent(?2,?1)
    Q(?0) <- PotentialCourse(?0), hasProfessor(?0,?1),
      hasStudent(?0,?2)
    Q(?0) <- AUX$0(?0), PotentialCourse(?0), hasStudent(?0,?1)
    Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0),
      hasProfessor(?0,?1)
    Q(?0) <- Course(?0), ImpartedCourse(?0), hasStudent(?0,?1)
    Q(?0) <- AUX$0(?0), hasProfessor(?0,?1), hasStudent(?0,?2)
    Q(?0) <- AUX$0(?0), hasStudent(?0,?1)
    Q(?0) <- AUX$0(?0), ImpartedCourse(?0), hasProfessor(?0,?1)
    Q(?0) <- Course(?0), ImpartedCourse(?0)
    Q(?0) <- AUX$0(?0), ImpartedCourse(?0), PotentialCourse(?0)
    Q(?0) <- AUX$0(?0), ImpartedCourse(?0)
ImpartedCourse(?0) <- Bachelor(?1), hasStudent(?0,?1)
  AUX$0(?0) <- Bachelor(?1), hasStudent(?0,?1)
ImpartedCourse(?0) <- MasterStudent(?1), hasStudent(?0,?1)
ImpartedCourse(?0) <- PhDStudent(?1), hasStudent(?0,?1)
  AUX$0(?0) <- MasterStudent(?1), hasStudent(?0,?1)
  AUX$0(?0) <- PhDStudent(?1), hasStudent(?0,?1)
    Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0),
      hasStudent(?0,?1)
```

---

```

Q(?0) <- AUX$0(?0), ImpartedCourse(?0), hasStudent(?0,?1)
Q(?0) <- ImpartedCourse(?0), hasProfessor(?0,?1),
    hasStudent(?0,?2)
Q(?0) <- ImpartedCourse(?0), hasStudent(?0,?1)
Q(?0) <- ImpartedCourse(?0), hasProfessor(?0,?1)
Q(?0) <- ImpartedCourse(?0), PotentialCourse(?0)
Q(?0) <- ImpartedCourse(?0)

```

Then unnecessary clauses are pruned, obtaining the following UCQ:

```

Course(?0) <- Professor(?1), hasProfessor(?0,?1)
Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)
ImpartedCourse(?0) <- hasStudent(?0,?1)
AUX$0(?0) <- hasStudent(?0,?1)
Q(?0) <- PotentialCourse(?0), hasProfessor(?0,?1),
    hasStudent(?0,?2)
Q(?0) <- AUX$0(?0), hasStudent(?0,?1)
Q(?0) <- ImpartedCourse(?0)

```

Now we will see how the process can be done with the optimisations presented in this chapter. The first optimisation and the first step to perform is preprocessing. This ontology can be preprocessed into the following set of clauses:

```

Person(?0) <- Student(?0)
Student(f0(?0)) <- ImpartedCourse(?0)
hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
Student(?0) <- UndergradStudent(?0)
Person(?0) <- Professor(?0)
Person(?0) <- PostDoc(?0)
UndergradStudent(?0) <- Bachelor(?0)

```

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

```
Student(?0) <- GradStudent(?0)
Student(?0) <- hasStudent(?1,?0)
Course(?0) <- PotentialCourse(?0)
GradStudent(?0) <- MasterStudent(?0)
GradStudent(?0) <- PhDStudent(?0)
Person(f0(?0)) <- ImpartedCourse(?0)
Professor(f1(?0)) <- ImpartedCourse(?0)
hasProfessor(?0,f1(?0)) <- ImpartedCourse(?0)
Person(f1(?0)) <- ImpartedCourse(?0)
Course(?0) <- ImpartedCourse(?0)
ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
Course(?0) <- Professor(?1), hasProfessor(?0,?1)
Professor(f1(?0)) <- Student(?1), hasStudent(?0,?1)
hasProfessor(?0,f1(?0)) <- Student(?1), hasStudent(?0,?1)
Person(f1(?0)) <- Student(?1), hasStudent(?0,?1)
Course(?0) <- Student(?1), hasStudent(?0,?1)
```

The preprocessing is done through several steps. First the normal resolution strategy is applied, with the following inferences performed:

```
Person(?0) <- Student(?0)
Student(f0(?0)) <- ImpartedCourse(?0)


---


∴ Person(f0(?0)) <- ImpartedCourse(?0)
```

```
Person(?0) <- Professor(?0)
Professor(f1(?0)) <- AUX$0(?0)


---


∴ Person(f1(?0)) <- AUX$0(?0)
```

```
Student(?0) <- hasStudent(?1,?0)
hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)


---


∴ Student(f0(?0)) <- ImpartedCourse(?0)
```



---

$\text{ImpartedCourse}(\text{?0}) \leftarrow \text{Student}(\text{?1}), \text{hasStudent}(\text{?0}, \text{?1})$ $\text{hasStudent}(\text{?0}, \text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{ImpartedCourse}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0}), \text{Student}(\text{f0}(\text{?0}))$

---

$\text{AUX\$0}(\text{?0}) \leftarrow \text{Student}(\text{?1}), \text{hasStudent}(\text{?0}, \text{?1})$ $\text{hasStudent}(\text{?0}, \text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0}), \text{Student}(\text{f0}(\text{?0}))$

---

$\text{Course}(\text{?0}) \leftarrow \text{Professor}(\text{?1}), \text{hasProfessor}(\text{?0}, \text{?1})$ $\text{hasProfessor}(\text{?0}, \text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0})$
$\therefore \text{Course}(\text{?0}) \leftarrow \text{AUX\$0}(\text{?0}), \text{Professor}(\text{f1}(\text{?0}))$

---

$\text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0}), \text{Student}(\text{f0}(\text{?0}))$ $\text{Student}(\text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$

---

$\text{Course}(\text{?0}) \leftarrow \text{AUX\$0}(\text{?0}), \text{Professor}(\text{f1}(\text{?0}))$ $\text{Professor}(\text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0})$
$\therefore \text{Course}(\text{?0}) \leftarrow \text{AUX\$0}(\text{?0})$

---

Then auxiliary predicates can be removed, except for those involved in cycles composed entirely of auxiliary predicates. The most simple of these cycles would be  $\text{AUX}_i(\text{f}_j(\text{?0})) \leftarrow \text{AUX}_i(\text{?0})$ . There are no such cycles in this example, hence all auxiliary predicates can be removed. The inferences to do this are the following:

---

$\text{Professor}(\text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0})$ $\text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{Professor}(\text{f1}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$

---

$\text{hasProfessor}(\text{?0}, \text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0})$ $\text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
---

---

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

---

```
∴ hasProfessor(?0,f1(?0)) <- ImpartedCourse(?0)
```

```
Person(f1(?0)) <- AUX$0(?0)
```

```
AUX$0(?0) <- ImpartedCourse(?0)
```

---

```
∴ Person(f1(?0)) <- ImpartedCourse(?0)
```

```
Course(?0) <- AUX$0(?0)
```

```
AUX$0(?0) <- ImpartedCourse(?0)
```

---

```
∴ Course(?0) <- ImpartedCourse(?0)
```

```
Professor(f1(?0)) <- AUX$0(?0)
```

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
```

---

```
∴ Professor(f1(?0)) <- Student(?1), hasStudent(?0,?1)
```

```
hasProfessor(?0,f1(?0)) <- AUX$0(?0)
```

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
```

---

```
∴ hasProfessor(?0,f1(?0)) <- Student(?1), hasStudent(?0,?1)
```

```
Person(f1(?0)) <- AUX$0(?0)
```

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
```

---

```
∴ Person(f1(?0)) <- Student(?1), hasStudent(?0,?1)
```

```
Course(?0) <- AUX$0(?0)
```

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
```

---

```
∴ Course(?0) <- Student(?1), hasStudent(?0,?1)
```

This allows removing the auxiliary predicate `AUX$0`, which is the only one in this example, resulting in the preprocessed ontology mentioned before. With the query the reachability test can be performed, resulting in the following set of clauses:

---

```

    Student(f0(?0)) <- ImpartedCourse(?0)
hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
    Student(?0) <- UndergradStudent(?0)
UndergradStudent(?0) <- Bachelor(?0)
    Student(?0) <- GradStudent(?0)
    Student(?0) <- hasStudent(?1,?0)
    Course(?0) <- PotentialCourse(?0)
    GradStudent(?0) <- MasterStudent(?0)
    GradStudent(?0) <- PhDStudent(?0)
    Professor(f1(?0)) <- ImpartedCourse(?0)
hasProfessor(?0,f1(?0)) <- ImpartedCourse(?0)
    Course(?0) <- ImpartedCourse(?0)
    ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
    Course(?0) <- Professor(?1), hasProfessor(?0,?1)
    Professor(f1(?0)) <- Student(?1), hasStudent(?0,?1)
hasProfessor(?0,f1(?0)) <- Student(?1), hasStudent(?0,?1)
    Course(?0) <- Student(?1), hasStudent(?0,?1)

```

Note that the clauses where **Person** is in the head have been removed. After this we can perform saturation and only three resolution steps are performed due to the work already done in preprocessing.

```

    Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)
hasProfessor(?0,f1(?0)) <- ImpartedCourse(?0)


---


    ∴ Q(?0) <- Course(?0), ImpartedCourse(?0), hasStudent(?0,?1)

    Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)
hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)


---


    ∴ Q(?0) <- Course(?0), ImpartedCourse(?0),
        hasProfessor(?0,?1)

```

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

```
Q(?0) <- Course(?0), ImpartedCourse(?0), hasStudent(?0,?1)
hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)


---


∴ Q(?0) <- Course(?0), ImpartedCourse(?0)
```

We can see that the last clause obtained subsumes the two clauses previously obtained, the result is therefore the following logic program:

```
Q(?0) <- Course(?0), ImpartedCourse(?0)
Q(?0) <- Course(?0), hasProfessor(?0,?2), hasStudent(?0,?1)
Student(?0) <- UndergradStudent(?0)
UndergradStudent(?0) <- Bachelor(?0)
Student(?0) <- GradStudent(?0)
Student(?0) <- hasStudent(?1,?0)
Course(?0) <- PotentialCourse(?0)
GradStudent(?0) <- MasterStudent(?0)
GradStudent(?0) <- PhDStudent(?0)
Course(?0) <- ImpartedCourse(?0)
ImpartedCourse(?0) <- Student(?1), hasStudent(?0,?1)
Course(?0) <- Professor(?1), hasProfessor(?0,?1)
Course(?0) <- Student(?1), hasStudent(?0,?1)
```

Another saturation step is run to remove auxiliary predicates that may be now removable. There are none in this case and the number of inferences performed is zero. Our logic program is already a Datalog program free of auxiliary predicates. We continue by unfolding this Datalog program. The number of inferences in this unfolding is much smaller due to the optimisations, as we can see:

```
Q(?0) <- Course(?0), ImpartedCourse(?0)
Course(?0) <- PotentialCourse(?0)


---


```

---



---

$\therefore Q(?0) \leftarrow \text{ImpartedCourse}(?0), \text{PotentialCourse}(?0)$
$Q(?0) \leftarrow \text{Course}(?0), \text{ImpartedCourse}(?0)$ $\text{Course}(?0) \leftarrow \text{ImpartedCourse}(?0)$
$\therefore Q(?0) \leftarrow \text{ImpartedCourse}(?0)$
$Q(?0) \leftarrow \text{Course}(?0), \text{ImpartedCourse}(?0)$ $\text{Course}(?0) \leftarrow \text{Professor}(?1), \text{hasProfessor}(?0,?1)$
$\therefore Q(?0) \leftarrow \text{ImpartedCourse}(?0), \text{Professor}(?1), \text{hasProfessor}(?0,?1)$
$Q(?0) \leftarrow \text{Course}(?0), \text{ImpartedCourse}(?0)$ $\text{Course}(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1)$
$\therefore Q(?0) \leftarrow \text{ImpartedCourse}(?0), \text{Student}(?1), \text{hasStudent}(?0,?1)$
$Q(?0) \leftarrow \text{Course}(?0), \text{ImpartedCourse}(?0)$ $\text{ImpartedCourse}(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1)$
$\therefore Q(?0) \leftarrow \text{Course}(?0), \text{Student}(?1), \text{hasStudent}(?0,?1)$
$Q(?0) \leftarrow \text{ImpartedCourse}(?0)$ $\text{ImpartedCourse}(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1)$
$\therefore Q(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1)$
$Q(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1)$ $\text{Student}(?0) \leftarrow \text{UndergradStudent}(?0)$
$\therefore Q(?0) \leftarrow \text{UndergradStudent}(?1), \text{hasStudent}(?0,?1)$
$Q(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1)$ $\text{Student}(?0) \leftarrow \text{GradStudent}(?0)$
$\therefore Q(?0) \leftarrow \text{GradStudent}(?1), \text{hasStudent}(?0,?1)$
$Q(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1)$ $\text{Student}(?0) \leftarrow \text{hasStudent}(?1,?0)$

---

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

---

```
∴ Q(?0) <- hasStudent(?0,?1), hasStudent(?2,?1)
```

With this we obtain the unfolded UCQ as follows:

```
Q(?0) <- ImpartedCourse(?0)
Q(?0) <- hasStudent(?0,?1)
```

With this example we can see in detail the effect of the optimisations. First, some operations are performed in the preprocessing because the query is not necessary for them. To produce the Datalog rewriting the original algorithm requires 10 inferences, while the optimised algorithm requires only 3. Additionally, the Datalog rewriting has 16 clauses in the original approach, while it has 13 in the optimised version, due to the removal of auxiliary predicates. Therefore, the Datalog program is produced faster and it is shorter, this does also help in the unfolding. The original algorithm performs 33 inferences to produce a UCQ with 36 clauses, that have to be checked for subsumption to reduce the final number to 7 clauses. Due to the optimisations and the shorter Datalog program to be unfolded, in the optimised version the unfolding requires only 9 inferences and produces only 2 clauses. The example shows how the removal of auxiliary predicates allows the production subsuming clauses that reduce the size of the final UCQ, from 7 to 2 in this case.

### 5.1.3 Pseudocode of the algorithms

Algorithm 5.1 shows the general algorithm of the system as a whole and the following listings focus on specific procedures in this general algorithm.

Among the specific algorithms, the first one to consider is the algorithm for preprocessing (algorithm 5.2). Preprocessing saturates the ontology with the same selection function previously used in REQUIEM (`sfRQR`), although in this

---

case after the saturation some clauses containing functional terms must be preserved, because they may participate in the rewriting with the query. This Datalog can only contain clauses that fit in some of the groups defined in the selection function to get the expected behaviour from the selection function. This restriction about the types of clauses that can be handled is overcome by introducing auxiliary predicates.

These auxiliary predicates can be removed (line 2) with the exception of auxiliary predicates that form loops, as the resolution applied for their removal would not finish in that case. If there are such predicates these will have to be preserved at least for the moment. Finally, the two usual subsumption checks (for atoms and clauses) are also performed in this program.

---

**Algorithm 5.1:** General kyrie2 algorithm

---

**Input:** Preprocessed  $\mathcal{ELHIJ}\mathcal{O}$  TBox  $\Sigma$ , UCQ  $q$ , working mode  
 $mode \in \{\text{Datalog}, \text{UCQ}\}$   
**Output:** Rewritten query  $q_\Sigma$

- 1  $q = \text{removeSubsumed}(\text{condensate}(q))$
- 2  $\Sigma_q = \text{reachable}(\Sigma, q)$
- 3  $q_\Sigma = \text{saturate}(\mathbf{s}, sfRQR, q, \Sigma_q)$
- 4  $q_\Sigma = \text{saturate}(\mathbf{s}, sfAux, q_\Sigma)$
- 5  $q_\Sigma = \text{reachable}(q_\Sigma)$
- 6 **if**  $mode = \text{Datalog}$  **then return**  $q_\Sigma$
- 7  $\Sigma_q = \{q_i \in q_\Sigma \mid \text{head}(q_i) \neq \text{head}(q)\}$
- 8  $q_\Sigma = \{q_i \in q_\Sigma \mid \text{head}(q_i) = \text{head}(q)\}$
- 9  $q_\Sigma = \text{saturate}(\mathbf{u}, sfNonRec, q_\Sigma, \Sigma_q)$
- 10 **return**  $q_\Sigma$

---



---

**Algorithm 5.2:** kyrie2 preprocess algorithm

---

**Input:**  $\mathcal{ELHIJ}\mathcal{O}^\top$  ontology  $\Sigma$   
**Output:** Preprocessed ontology  $\Sigma$

- 1  $\Sigma = \text{saturate}(\mathbf{p}, sfRQR, \Sigma)$
- 2  $\Sigma = \text{saturate}(\mathbf{u}, sfAux, \Sigma)$
- 3  $\Sigma = \text{removeSubsumed}(\text{condensate}(\Sigma))$
- 4 **return**  $\Sigma$

---

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---



---

### Algorithm 5.3: kyrie2 saturation algorithm

---

**Input:** Working mode  $mode$ , selection function  $sf$ , Datalog program  $q$ ,  
[optional Datalog clauses  $\Sigma$ ]

**Output:** Datalog program  $q_\Sigma$

```

1   $pending = \text{new } SortedQueue(q, \text{shortestFirst})$ 
2   $done = \text{new } Queue()$ 
3  if  $undefined(\Sigma)$  then
4     $\Sigma \equiv done$ 
5  end
6  else
7    for  $\gamma \in \Sigma$  do
8       $sf.select(\gamma)$ 
9    end
10 end
11 while  $\neg pending.isEmpty()$  do
12    $q_i = pending.pop()$ 
13    $sf.select(q_i)$ 
14    $done.push(q_i)$ 
15   forall the  $q_j \in \Sigma$  do
16      $Q_{i,j} = resolve(q_i, q_j)$ 
17     forall the  $q_k \in Q_{i,j}$  do
18        $q_k = condensate(q_k)$ 
19       if  $\forall q \in pending \cup done. q \not\preceq_s q_k$  then
20          $done = \{q \in done \mid q_k \not\preceq_s q\}$ 
21          $pending = \{q \in pending \mid q_k \not\preceq_s q\}$ 
22          $pending.push(q_k)$ 
23       end
24     end
25   end
26 end
27 if  $mode \neq u$  then  $done = done \cup \Sigma$ 
28 if  $mode \neq p$  then  $done = sf.prune(done)$ 
29 return  $done$ 

```

---



---

**Algorithm 5.4:** Selection algorithm in sfAux (**sfAux.select**)

---

**Input:** Clause  $\gamma$   
**Output:** Specific atoms in  $\gamma$  are marked selected

```
1 check = True
2 foreach  $p \in \text{body}(\gamma)$  do
3   if isAux( $p$ ) then
4     select( $p$ )
5     check = False
6   end
7 end
8 if isAux(head( $\gamma$ ))  $\wedge$  check then
9   select(head( $\gamma$ ))
10 end
```

---

The second algorithm to consider is the resolution algorithm (algorithm 5.3). This resolution algorithm uses subsumption checks to limit the explosion produced by blind resolution on all combinations of clauses. As can be seen, this algorithm has a parameter (**p**, **s** or **u**):

- **p** ignores the pruning definitions corresponding to the selection function, *preserving* clauses that would otherwise be removed in the corresponding cleaning stage.
- **s** *separates* the clauses that are obtained anew from the old ones, returning only those that are new, e.g. when saturating the query with the clauses derived from the TBox all produced clauses will be query clauses.
- **u** for *unfolding*, this method does not skip the cleaning stage and does not separate the results.

Clauses that are subsumed can be safely removed as soon as they are generated, and by doing so the generation of other subsumed clauses is avoided, limiting the explosion in the resolution. These subsumption checks (line 1 in algorithm 5.1 and loop in line 17 in algorithm 5.3) are one of the optimisations, explained in section 5.2.2.

The rest of the algorithms are also explained in more detail in the following sections (e.g. algorithm 5.2 is explained in section 5.2.1). Using first the shortest

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

clauses, prioritizing some resolution steps, as in line 12 in algorithm 5.3 has some benefits as we will see in section 5.2.3. Finally, in section 5.2.4 we consider the cases in which the condition for line 4 in algorithm 5.3 does not hold. This means that the clauses that act as side premises and the clauses that act as main premises in the current resolution can be separated into two sets and all new resolved clauses will act as main premises. In this case, once a preliminary Datalog program has been produced, this Datalog program is pruned out of auxiliary predicates, with the exception of those that need to be kept for being recursive, as we have seen with algorithm 4.2. These additional resolution steps simplify the Datalog program without losing any answer, as we will see in section 4.3.2.

Finally, the rewritten query in Datalog is returned, or if the system should unfold the query for the underlying systems then the query is unfolded as much as possible. This implies either having the minimum number of different predicates appearing in the head of some clause or only the query predicate (obtaining a UCQ) in case of non-recursive Datalog.

## 5.2 Optimisations

### 5.2.1 Ontology preprocessing

We propose a preprocessing stage that consists in performing inferences with ontology clauses before any query is executed. These inferences only depend on the ontologies used, which are normally more stable than the data sources. Hence, preprocessing is done only once as part of the configuration and once again every time the ontologies are updated. Its objective is to save time when queries arrive to our system. The preprocessing does not include the data in the data sources but only the ontology, contrary to other approaches that include preprocessing in the data (e.g. [Kontchakov et al., 2010]). To clarify, this means that we make no assumptions about the data sources, i.e. data in the sources may be as dynamic as data streams [Calbimonte et al., 2012]. Basically, we can see that the rewriting generates a query  $q_\Sigma$  from an original query  $q$  using the TBox  $\Sigma$ . As long as there are no changes in  $\Sigma$ , the rewriting process remains the same and so does the

---

result ( $q_\Sigma$ ). This happens regardless of how many changes may happen in the data sources and how much the answers to  $q_\Sigma$  may vary through time.

Among existing OBDA approaches, preprocessing is only done by Venetis [Venetis et al., 2011], which focuses on reusing partial results among similar queries. Some approaches are capable of some form of preprocessing; for example computing the dependency graph in the case of Nyaya.

We have seen in section 2.5 how resolution with free selection works. Part of this saturation are inference steps where both premises are clauses in the ontology or derived exclusively from the ontology. The query is not necessary for these inference steps, therefore they can be performed before the query is available and only once for all the queries.

To understand this optimisation we can simply consider that there are three types of clause combinations that happen in the search for valid resolution steps: (a) ontology clauses with ontology clauses, (b) ontology clauses with query clauses and (c) query clauses with query clauses. Among these three groups, the first one can be done before any query is available and is the basis for this optimisation. The second group is the normal resolution performed after the query is available. Finally, the third group does not contain any valid inference, as a query clause can only be a main premise, not a side premise. This is also the basis for the optimisation explained in section 5.2.4.

The resolution corresponding to the preprocessing is done with the algorithm introduced in section 4.2 (algorithm 5.3) by using the same selection function defined previously for REQUIEM, which is correct and complete, with one modification: all clauses that are not subsumed are preserved after the resolution. Clauses containing functional terms, among others, would be discarded at the end of the saturation in REQUIEM, obtaining a Datalog program. In this case, these clauses may still lead to relevant results when the query is available, and thus are preserved until the second type of inferences, with the query, can be performed. For instance we can consider the following resolution step:

$$\begin{array}{c}
 \text{Person}(\text{?0}) \leftarrow \text{Student}(\text{?0}) \\
 \text{Student}(\text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0}) \\
 \hline
 \therefore \text{Person}(\text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})
 \end{array}$$

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

Besides this resolution, some additional inferences are performed in a second stage of resolution to reduce the inferences needed later. As explained in the preprocess algorithm (algorithm 5.2), some auxiliary predicates need to be introduced for the previous saturation function to work as expected, selecting appropriately main and side premises.

By removing auxiliary predicates we obtain a Datalog program that is equivalent, and a reduction in the total number of clauses that may potentially be generated since all the clauses that contain some of the removed auxiliary predicates can no longer be generated. This can be done at this point due to the nature of the following resolution stage and its selection function. Inferences in the following resolution step involve a query clause and an ontology clause. This means that the head atom will always be selected for the ontology clause and will be the only atom selected for this type of clauses. Therefore, we can use this selection function even if we obtain clauses that are not contained in any of the groups previously defined in REQUIEM and obtain a finite resolution for the generation of the Datalog program by separating ontology and query clauses (section 5.2.4).

To remove auxiliary predicates, the selection function selects the auxiliary predicates excluding one of them for each cycle through auxiliary predicates (as defined in section 2.1). This cycle detection is also used for the unfolding stage. Cycle detection works in a similar way in REQUIEM, producing a linear Datalog in case the unfolding is not possible. In our case, we reduce the number of different predicates that appear in the head of some clause by using this cycle detection.

For the inferences with auxiliary predicates, we can consider as an example the following axiom:

$$\exists hasStudent.Student \sqsubseteq \exists hasProfessor.Professor$$

This axiom produces the clauses:

```
AUX$0(?0) <- Student(?1), hasStudent(?0,?1),  
Professor(f1(?0)) <- AUX$0(?0) and  
hasProfessor(?0,f1(?0)) <- AUX$0(?0)
```

From these Datalog clauses we can obtain by resolution:

---

```

Professor(f1(?0)) <- Student(?1), hasStudent(?0,?1) and
hasProfessor(?0,f1(?0)) <- Student(?1), hasStudent(?0,?1)

```

If the auxiliary predicate `AUX$0(x)` is not chosen for exclusion for any cycle through auxiliary predicates then this can be done for all clauses containing this predicate. The auxiliary predicates have no real correspondence with any predicate in the ontology. Their only function is to allow these inferences. Once the inferences are performed, the predicates and the clauses that contain them are no longer needed, and can be discarded. Therefore, the production of new clauses is leveraged with the elimination of some clauses, controlling the number of clauses produced after the preprocessing, as can be seen in table 7.4.

Obviously, newly produced clauses can at the same time produce additional inferences. Any inference done during the preprocessing stage reduces the number of inferences to be performed in later stages. As we have seen the inferences are separated into two different saturation stages. In the first stage, no unary atoms are selected for the inference. This means that the only inferences that are performed are the inferences that will be needed to produce the Datalog program (i.e. to remove functional terms). This avoids some types of recursion and limits the inferences in this stage and the size of the preprocessed ontology. Despite these limits, the generation of results is notably faster due to the preprocessing stage, specially Datalog results, as we will see in section 7.4.

## 5.2.2 Query subsumption check

Subsumption checks consist on checking whether a part of the query that is being rewritten is subsumed by another. When this happens one of the two parts of the query can be removed, resulting in a shorter query.

Subsumption checks are usually performed by checking pairs of clauses, checking whether one subsumes the other and removing the clause that is subsumed. Another usual optimisation operation, clause condensation, is performed by checking intra-clause subsumption, that is whether an atom subsumes another. In this case the subsuming atom is removed, since they are grouped by conjunctions, as opposed to the disjunctions that we find between clauses.

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

In the case of atom subsumption all previous approaches perform a similar check.

- REQUIEM checks atom subsumption in the “condensation” optimisation step.
- In the case of Rapid this is done with the “shrinking” resolution rule.
- Nyaya performs a similar operation in the “factorization” step.
- Finally, Presto has the function “DeleteRedundantAtoms”, naturally with a properly descriptive name.

In the case of clause subsumption check the situation is similar.

- REQUIEM performs this in a separate stage after the resolution has finished by checking all clauses. When using the “F” mode, REQUIEM does also perform a “full” subsumption check, as described in [Bachmair and Ganzinger, 2001]. This means that during resolution newly derived resolvents may be deleted if they are subsumed by old or processed clauses. However, the subsumption check is not performed the other way around with respect to old and new clauses.
- Rapid performs a similar subsumption check, but the generation of subsumed clauses is reduced and more controlled, what allows limiting the check to subsets of the generated clauses. This check is performed after each unfolding step.
- In the case of Nyaya, subsumed clauses are removed with the elimination step, which is optionally performed after every rewriting step.
- Presto produces a factorized Datalog where subsumption check becomes less tractable, and thus there is no subsumption check between clauses. However, subsumption between sets of atoms that share some variable is considered with the *most general subsumees*.

In our context, the clever handling of the unfolding sets done in Rapid cannot be added in a straightforward way, since the Datalog used is not linear, what

---

means that unfolding sets are not composed of atoms but of conjunctions of atoms. Our optimisation consists in performing the subsumption check among all generated clauses, as in *REQUIEM*. However, instead of doing this as a separate stage from resolution, we add subsumption checks as part of the resolution process (line 17 in algorithm 5.3). We formally prove the correctness of this optimisation in section 5.3 (proposition 4). More precisely, subsumed clauses can be removed at any time obtaining an equivalent query, i.e. a query equally satisfiable and that provides the same answers. We evaluate the efficiency of this optimisation along with the other optimisations in section 7.4.

With this optimisation, new clauses are generated using resolution with free selection and every time a new clause is generated its subsumption is checked. Therefore subsumed clauses are identified — and removed — as soon as a pair (subsuming-subsumed) is generated. This allows finding clauses that are equivalent and processing them only once. The subsumed clause can be safely removed, avoiding all the inferences it would take part in the resolution with free selection, i.e. preventing the generation of the tree of clauses that could be inferred by using that clause and its successors. As a result, all later stages have a lesser computational load and require less time, including this subsumption check, which in this case checks less clauses since the number of generated clauses is reduced by removing subsumed clauses as soon as possible. For this reason, early subsumption checks contribute to the efficiency of the rewriting process. Additionally to that, in early stages of the query rewriting (also known as query expansion) process, there are less clauses in the query, and therefore the subsumption checks are less computationally costly.

Therefore, these subsumption checks that can be performed during resolution can be performed in any resolution stage. Finally, there is an additional advantage in performing some resolution and subsumption checks in the preprocessing stage: once the query is available, in the latter resolution process (the base clause is always the query) we know that the head of the generated clauses always belongs to the query. Therefore, subsumption check between clauses can be further restricted to query clauses, excluding ontology clauses in the latter resolution stages where ontology and query clauses are processed separately. This separation between query and ontology clauses is detailed further in section 5.2.4.

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

For example we can consider the following resolution step:

$$\begin{array}{lcl}
 \gamma_1: & Q(?0) <- \text{Course}(?0), \text{ImpartedCourse}(?0) \\
 \gamma_2: & \text{Course}(?0) <- \text{ImpartedCourse}(?0) \\
 \hline
 \gamma_3: & Q(?0) <- \text{ImpartedCourse}(?0), \text{ImpartedCourse}(?0)
 \end{array}$$

In this case  $\gamma_3$  is derived and immediately simplified (condensed) into  $Q(?0) <- \text{ImpartedCourse}(?0)$ . As a result,  $\gamma_1$  is subsumed by  $\gamma_3$  and immediately discarded for further inferences, avoiding other resolution steps and the generation of other queries, such as:

$$\begin{array}{lcl}
 \gamma_1: & Q(?0) <- \text{Course}(?0), \text{ImpartedCourse}(?0) \\
 \gamma_2: & \text{Course}(?0) <- \text{Student}(?1), \text{hasStudent}(?0, ?1) \\
 \hline
 \gamma_3: & Q(?0) <- \text{Student}(?1), \text{hasStudent}(?0, ?1), \text{ImpartedCourse}(?0)
 \end{array}$$

By avoiding the further use of this query clause, we prevent the generation of all the clauses that could be derived from it, recursively, avoiding the generation of a full tree of clauses. More precisely in this example we avoid the generation of all the clauses that involve the taxonomy of students.

This can happen in the preprocessing stage as well. For another example in the context of the previous section we can consider the clause following resolution step:

$$\begin{array}{lcl}
 \gamma_4: & \text{AUX\$0}(?0) <- \text{ImpartedCourse}(?0), \text{Student}(f0(?0)) \\
 \gamma_5: & \text{Student}(f0(?0)) <- \text{ImpartedCourse}(?0) \\
 \hline
 \gamma_6: & \text{AUX\$0}(?0) <- \text{ImpartedCourse}(?0), \text{ImpartedCourse}(?0)
 \end{array}$$

Again this can be condensed into  $\text{AUX\$0}(?0) <- \text{ImpartedCourse}(?0)$ . Analogously to the previous example,  $\gamma_4$  is subsumed and can be removed immediately. Doing this in the preprocessing stage does not only save time for the inferences, but also for the latter subsumption checks, reducing the size expansion of the preprocessed ontology.

### 5.2.3 Prioritising some inferences

Different resolution strategies have different specifications with respect to the order in which inferences should be performed. Resolution with free selection



---

may establish some order, depending on the selection function, by prioritizing the selection of some atoms in some sets of clauses, but it does also allow some degree of freedom with respect to which clauses should be resolved first. In this case we add some ordering criteria by using first some clauses in the resolution, with the expectation of producing subsuming clauses earlier and increasing the effect of the optimisation described in section 5.2.2.

Two of the state-of-the-art systems indicate the order of the inference steps that are performed during their resolution: REQUIEM and Rapid.

- REQUIEM uses a selection function, which establishes some order in the inference rules that are applied, but there are no more ordering criteria and hypothetically using some clauses in the resolution could help to reduce the time required.
- In Rapid clauses are selected more carefully during the resolution process, we can find that:
- *shrinking* and *unfolding* rules are applied alternatively, and shorter query clauses are considered first for unification, since those are the ones that more likely will subsume others.

However, to make subsumption checks effective in this scenario, another modification should be included. REQUIEM selects clauses that have been previously generated for resolution in a FIFO fashion, what causes the exploration of the search space to be similar to breadth first. An earlier generation of subsuming clauses and thus a greater reduction in the clause space explored can be obtained by selecting previously generated clauses in a LIFO fashion. Preserving the saturation method in REQUIEM but using first the latest clauses generated, as in depth-first search instead of breadth-first search, provided a good improvement on its own. The rationale behind this is that:

- results suggest that depth first search helps to find subsuming clauses earlier. These clauses are generated after a condensation (atom subsumption in one clause). This condensation is more likely as a deeper path is traversed in the resolution. Because a deeper path explores the atoms that can be unified and possibly subsumed in a specific clause.

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

- when clause subsumption is detected, branches are pruned. A subsumed clause means a subsumed branch of clauses. The less explored the pruned branch is, the more exploration is avoided. Depth-first search delays the exploration of the remaining branches, which may be prevented, gaining more efficiency.

Furthermore, choosing shortest clauses first produces a greater improvement. The rationale for this is that shortest clauses in a UCQ have less atoms in the body and can more easily subsume other clauses, using these clauses first when checking for subsumption prevents more subsumed clauses from being used and thus generated.

This optimisation can be included in any system where the resolution does not have any specific order. However it must be noted that this optimisation is relevant when combined with the subsumption check every time a new clause is generated, as in the previous section. In these cases, this optimisation allows rewriting queries faster — as evaluated in section 7 — without refusing to any expressiveness. If the subsumption check is not performed and subsumed clauses are eliminated after resolution, then the order in which clauses are generated during resolution is irrelevant for any optimisation purposes, since in the end the same number of clauses will be generated: the full saturation.

Additionally, as explained in section 5.2.1, the inferences that may happen between two ontology clauses can be performed before the query is available, what means that once it is available we can consider only the remaining inferences, which will always use a query clause as the base clause, ordering further the combinations of clauses that are considered. For example we have seen in previous sections that the next inference happens at an early stage:

```
AUX$0(?) <- ImpartedCourse(?), Student(f0(?))  
Student(f0(?)) <- ImpartedCourse(?)  
∴ AUX$0(?) <- ImpartedCourse(?)
```

Had different clauses and atoms in them been selected this other inferences similar to the next one could have been performed.

---

```

AUX$0(?0) <- ImpartedCourse(?0), Student(f0(?0))
Student(?0) <- GradStudent(?0)
∴ AUX$0(?0) <- ImpartedCourse(?0), GradStudent(f0(?0))

```

Basically the whole taxonomy for students would have been explored. All the clauses involved in this taxonomy produce longer clauses that are subsumed by the already mentioned `AUX$0(?0) <- ImpartedCourse(?0)`. This would produce a tree of subsumed clauses if the inference continued using that clause instead of the shorter `AUX$0(?0) <- ImpartedCourse(?0)`.

## 5.2.4 Constraining searches

There are two main searches within the resolution we have been describing so far. First of all, to apply a resolution rule the clauses that fulfill the roles of main premise and side premise must be found. Second, once a resolution step has been performed, according to section 5.2.2, subsumed and subsuming clauses are searched for each newly generated clause.

Among the analysed systems, two of them apply a similar technique: Rapid and Nyaya. In the case of Rapid, the base clause for the inference is always a query clause. Nyaya applies the same technique but only in its optimised version.

In our case, during our resolution we have two types of clauses depending on their provenance. On the one hand there are some clauses that come from the ontology, on the other hand there is a set of clauses composed by the query and clauses that have the query as an ancestor. These two groups are easily differentiable: any clause that has been derived from the query will keep the query predicate as the head predicate. For example for a query predicate 'Q' we have that:

```

This is an ontology clause:  Course(?0) <- Student(?1), hasStudent(?0,?1)
This is a query clause:      Q(?0) <- Course(?0), ImpartedCourse(?0)

```

The role of these two groups of clauses in resolution is also clear when both are available (after preprocessing). Query clauses will act as main premises, because their head cannot be unified. Ontology clauses will act as side premises, because all the inferences that could be done with ontology clauses as main premises are

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

among ontology clauses, and thus could and were performed during the preprocessing stage, as described in section 5.2.1. Thus, the search for main premises and side premises can be restricted to these two sets of clauses that can be separated before starting the resolution process.

After preprocessing, all generated clauses are query clauses, what means that subsumption check needs only to be performed among query clauses, saving a good number of subsumption checks. This optimisation by separating query and ontology clauses is applicable to systems that use resolution with free selection (e.g. REQUIEM). Systems that perform a different resolution procedure may use different methods to constrain searches. Other resolution strategies may use different techniques to constrain searches. For instance in the case of SLD-resolution as used in logic programming [Apt, 1988] indexing the clauses depending on the head predicate would help to find clauses that can be used in the resolution.

### 5.3 Proofs

The optimisations performed in the process are basically engineering optimisations. Therefore there are not many theoretical considerations to be done in this regard.

In the case of the **preprocessing**, the transformations that are being done in the Datalog program are the same that would be done after having the query. There are two groups of transformations: addition of clauses, i.e. inferred clauses, and removal of clauses, due to subsumption or other reasons. The addition of clauses can be safely performed at any time, as long as the resolution is sound. The only limitation without having an available query consists on not being able to perform the resolution steps in which the query participates. These steps in the resolution will need to be performed later. The removal of clauses containing function symbols is done later, producing the Datalog program. Some operations in this removal step cannot be performed during the preprocessing stage, since some clauses may participate in the resolution with the query. The solution adopted in this case consists in not eliminating any clauses containing function symbols to conform to Datalog, but instead keeping a partially saturated TBox in FOL.

---

In short, this optimisation does not affect what is done but when it is done, therefore correctness remains unaltered. This means that this optimisation is applicable to any system that performs this type of inferences where the query does not participate.

In the case of **query subsumption check**, the feasibility of removing subsumed clauses is already stated by Bachmair [Bachmair and Ganzinger, 2001]: “a (partial) proof (attempt) that is subsumed by another is redundant and should be deleted to avoid useless computations”. Therefore it is well known not only that subsumed clauses can be removed but that they should be better removed. “Deletion of subsumed clauses” is also stated as a valid rule of transformation of logic programs in [Pettorossi and Proietti, 1994]. More specifically we can find deletion of subsumed clauses as the rule 11 in the search for the least Herbrand model and equivalently in our case in the search of the *set* of certain answers. However, we prove the correctness of this operation and we introduce a helping proposition for it (proposition 3) as it will be also useful in the next chapter.

**Proposition 3.** *Let  $\mathcal{J} = \langle \Sigma, \mathcal{A} \rangle$  and  $\mathcal{J}' = \langle \Sigma', \mathcal{A} \rangle$  two OBDA systems such that  $\gamma_a, \gamma_b$  are two clauses in  $\Sigma$  with the same head predicate  $p$  and  $\Sigma \setminus \gamma_b = \Sigma'$ . If  $\varphi_{\mathcal{J}}(\gamma_b) \subseteq \varphi_{\mathcal{J}}(\gamma_a)$  then  $\varphi_{\mathcal{J}}(\gamma_b) \subseteq v_{\mathcal{J}'}(p)$ .*

*Proof.* The proof follows from proposition 1 and definitions 2 and 1. According to the definitions,  $\varphi_{\mathcal{J}'}(\gamma_a) \subseteq v_{\mathcal{J}'}(p)$ . The proposition has the preconditions that  $\varphi_{\mathcal{J}}(\gamma_b) \subseteq \varphi_{\mathcal{J}'}(\gamma_a)$  and  $p$  is the head predicate of  $\gamma_a$  and  $\gamma_b$ . Therefore  $\varphi_{\mathcal{J}}(\gamma_b) \subseteq v_{\mathcal{J}'}(p)$ , as in proposition 1, with equal consequences.  $\square$

**Proposition 4.** *Let  $\mathcal{J} = \langle \Sigma, \mathcal{A} \rangle$  be a OBDA system and let  $\gamma_a, \gamma_b$  be two clauses in  $\Sigma$  such that  $\gamma_a \succeq_s \gamma_b$  and let  $\mathcal{J}' = \langle \Sigma', \mathcal{A} \rangle$  where  $\Sigma' = \Sigma \setminus \gamma_b$ . Then,  $\varphi_{\mathcal{J}}(\gamma_b) \subseteq \varphi_{\mathcal{J}'}(\gamma_a)$ .*

*Proof.* The proof is immediate due to the definitions of clause subsumption and the contributions of a clause, section 2.1.1 and definition 1 respectively. For all  $\vec{\alpha}$  in the contributions of  $\gamma_b$  (i.e.  $(\mathcal{J} \models \mu(\text{body}(\gamma_b))) \wedge (\mu(\text{head}(\gamma_b)) = p(\vec{\alpha}))$ ) it must be in the contributions of  $\gamma_a$  because  $\gamma_a \models \gamma_b$  according to the definition of subsumption.  $\square$

## 5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING

---

Due to the results of proposition 4 and previous, subsumed clauses can be removed, while preserving the values for all predicates in the TBox.

The **prioritisation of some inferences** is a similar case. This optimisation changes nothing in the implementation or formalisation but the ordering of the resolution steps. The order is enforced by the prioritization of the inferences may be equal to the normal ordering if by chance they happen to coincide. However by establishing a specific order according to the heuristics previously described we can avoid some worst case scenarios and obtain a good behaviour in a process that would have been executed in an indeterministic way otherwise.

Finally **constraining the searches** simply improves the way clauses that are going to take part in inferences or subsumption checks are found. This is an optimisation in the search of clauses by using more sophisticated data structures that speed up the resolution by not attempting resolutions or subsumption checks that lead to no results. All these optimisations are evaluated empirically in section 7.4.

### 5.4 Conclusions

We have presented some optimisations that can be done to a query rewriting process by focusing on the engineering part. With this we mean that no additional constraints are added to the problem and that the handled expressiveness is not modified. Adding constraints, reducing the input expressiveness or increasing the output expressiveness may ease the process from a computational point of view. Instead we focus on implementation details, algorithms and data structures, that can be extrapolated and integrated into other approaches, as already pointed out. We have shown that these details have a strong impact on the efficiency of the process, as will be seen in the evaluation performed in section 7.4, especially when compared with REQUIEM.

There is no doubt about the relevance of obtaining better theoretical complexities or worst case behaviours. However algorithms with the same theoretical complexity, or an equivalent worst-case behaviour, can produce results several order of magnitude different. This may happen not only in the most extreme cases, but on average on a set of representative cases. This highlights the relevance of

---

proper benchmarks in the area for a quantitative evaluation.

Besides, when better behaviours are obtained through a change in the supported expressiveness, this raises the question about the relevance of the lost expressiveness (or additional expressiveness manageable in the output). The estimation of this relevance can only be done through the statistical measure of the use cases and the actual problems in the area. This has been done in the Linked Data context [Glimm et al., 2012], but it remains to be done in a OBDA context, to the best of our knowledge.

In addition, the difference between the results obtained as Datalog and those obtained as a UCQ is astounding in most of the cases. Obviously, a UCQ is simpler to manage than a Datalog program. However, simpler from a conceptual point of view does not imply that it is simpler from a computational point of view. Datalog is more algorithmically complex to handle, but it provides a more compact representation, which may be more convenient for some systems. In this context, further optimisations would benefit from a model of the underlying systems, the ones that will receive the rewritten query, or several models for different groups of underlying systems with some similar behaviour. This model would allow addressing the particular characteristics of these systems and producing a query that is simpler to process by these systems. Without such a model to compare the differences in the Datalog and UCQ rewritings we must leave the relative relevance of the differences in both rewritings at the discretion of the reader.

## **5. ENGINEERING OPTIMISATIONS FOR QUERY REWRITING**



## Chapter 6

# Optimisations in the presence of an EBox

As we have seen in section 2.6, EBoxes provide meta-information about the extension of the predicates in the ontology, in particular about extensional containment relationships among predicates in the ontology. This meta-information can be used to optimise the time required for the query rewriting process and its results (the size of the rewritings). In this chapter we will see a set of optimisations that are enabled by using the information in an EBox.

As in the previous chapter, the optimisations that we perform in this case change again the OBDA general scenario that we have seen in figure 1.1. In this case, the preprocessing stage of the previous chapter is preserved, the dependency of the mappings for the rewriting is also present, with an intermediate step in the generation of the EBox. Therefore the process in figure 6.1 subsumes the ones that we have seen in previous chapters.

As we will see, the optimisations presented here subsume the ones presented in chapter 4. With the EBox we model how the extension of some predicates relates with some other predicates. We can also model if the extension of some predicate  $p$  is empty for some OBDA system  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$ , which we express as  $v_{\mathcal{J}}^e(p) = \emptyset$  and can be modeled in the EBox as  $p \sqsubseteq \perp$ . These predicates that have no extension can be treated in the same way as the predicates with no mappings in chapter 4.

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

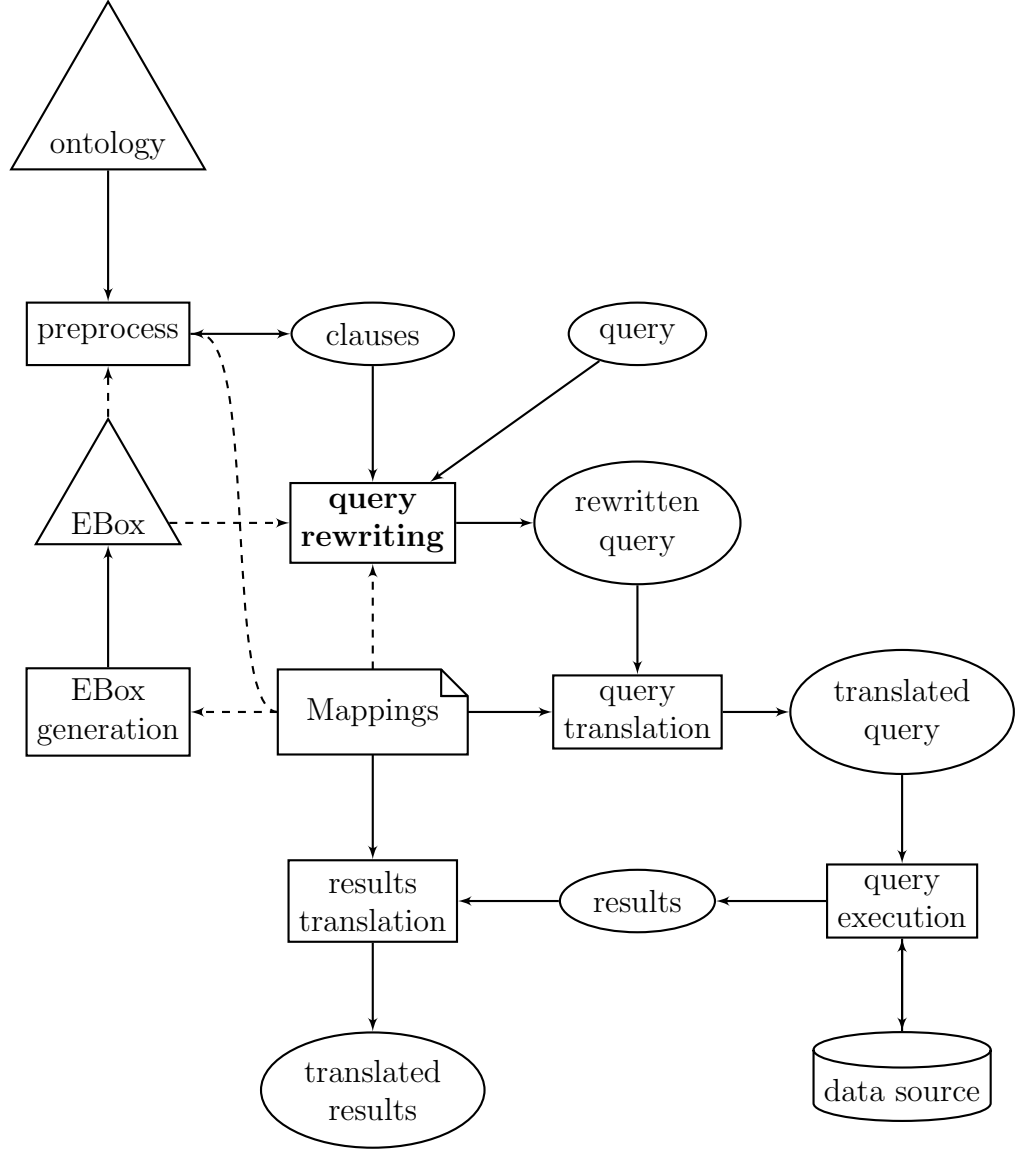


Figure 6.1: OBDA process with query rewriting, a preprocessing stage and alternatively the use of some mappings or an EBox.

The optimisations that consider the EBox are independent from the ones presented in the previous chapter, i.e. their application is not necessary (it is optional). In our case we will apply the optimisations from the previous chapter additionally to the optimisations that make use of an EBox. We will see that there are several stages in the algorithm where resolution is applied. Therefore

---

the optimisations in the previous chapter are beneficial. From a technical point of view, this chapter focuses on additional optimisations that use an EBox.

We have seen in chapter 4 that we can use the information about the presence or absence of a mapping to improve the query rewriting at the present mappings. We have also seen in chapter 4 that we can consider mappings as complete or partial, stopping the reachability algorithm on the first mapping found or the last one. So far, we have considered that the completeness of a mapping specification was a property of the whole specification, i.e. the completeness of a mapping specification could be guaranteed or otherwise it could not be assumed. However, we can consider the completeness of mappings as well as other extensional inclusion dependencies between mapping assertions into more detail. These dependencies are modeled as ABox dependencies [Rodríguez-Muro and Calvanese, 2011]. These dependencies can be expressed as DL axioms, and can be collected together into a new type of box, the so called EBox [Rosati, 2012]. EBox assertions are written in the same syntactical form as TBox assertions. In our case, we will use EBoxes that are expressed in  $\mathcal{ELHIJ}$ .

Using an EBox provides additional information about the individuals (and thus answers) that can be provided by particular mapping assertions. We extend the notion of  $\mathcal{ELHIJ}$  EBoxes to include the information presented in chapter 4, using the bottom entities to specify the absence of a mapping for some predicate. Therefore, for the case of EBoxes we can consider some greater expressiveness that can be denoted as  $\mathcal{ELHIJ}_\perp$ .

In  $\mathcal{ELHIJ}_\perp$ , concept ( $C$ ) and role ( $R$ ) expressions are formed according to the following syntax (where  $A$  denotes a concept name,  $P$  denotes a role name, and  $a$  denotes an individual name):

$$\begin{aligned} C &::= A \mid C_1 \sqcap C_2 \mid \exists R.C \mid \{a\} \\ R &::= P \mid P^- \end{aligned}$$

An  $\mathcal{ELHIJ}_\perp$  axiom is an expression of the form  $C_1 \sqsubseteq C_2$ ,  $B \sqsubseteq \perp$ ,  $R_1 \sqsubseteq R_2$  or  $P \sqsubseteq \perp$  where  $C_1, C_2$  are concept expressions and  $R_1, R_2$  are role expressions. As usual,  $\perp$  denotes the bottom entity, i.e. `owl:Nothing`, `owl:bottomObjectProperty`, and `owl:bottomDataProperty` in the case of OWL2. An  $\mathcal{ELHIJ}_\perp$  TBox  $\mathcal{T}$  is a set

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

of  $\mathcal{ELHI}\mathcal{O}_\perp$  axioms. For all considerations in the remainder of this work there are no distinctions between  $\mathcal{ELHI}\mathcal{O}$  and  $\mathcal{ELHI}\mathcal{O}_\perp$  EBoxes, therefore we will refer to them simply as  $\mathcal{ELHI}\mathcal{O}$  EBoxes.

We have seen that the absence of a mapping for a predicate  $p$  can be modelled as  $p \sqsubseteq \perp$ . In addition, we can also consider axioms of the form  $A_1 \sqsubseteq A_2$ . This means we do not need to specify a reachability algorithm globally for an ontology, the behaviour may vary from one predicate to another depending on these axioms. We will see more examples and possibilities along the chapter. In general, this information can be used to address more specifically the query rewriting to the set of mapping assertions that provide the entirety of the set of certain answers and reduce redundancy in the results. By reducing this redundancy we expect to reduce the computational load in the system. This means shorter rewriting times, in general, but especially shorter rewritten queries that can obtain all the certain answers more efficiently.

Once again, in this chapter we present the same sections. We present the algorithm in section 6.2. We present the optimisations implemented in this algorithm in section 6.3. We present the proofs for the correctness and completeness of these optimisations in section 6.4. Finally, we present some conclusions in section 6.5.

### 6.1 Preliminaries

We introduce in this section some definitions that will be useful for the remainder of the chapter. First of all, for some axiom  $\psi$  we will refer to its left hand side and right hand side as respectively  $LHS(\psi)$  and  $RHS(\psi)$ .

We define the *axiom graph* of a set of DL axioms and the *clause graph* of a set of Horn clauses. These definitions allow defining some relevant properties more concisely, as they make a more clear distinction on whether an axiom (or clause) is contained in a subgraph.

**Definition 3.** We define  $dlgraph(\mathcal{T}_\mathcal{O})$ , the axiom graph for an  $\mathcal{ELHI}\mathcal{O}$  TBox  $\mathcal{T}_\mathcal{O}$ , as the directed graph  $\mathcal{G}(\psi) = \langle \mathcal{V}(\psi), \mathcal{W}(\psi) \rangle$  such that: (i) for each axiom  $\psi \in \mathcal{T}_\mathcal{O}$ ,  $\psi \in \mathcal{V}(\psi)$ ; and (ii) for each  $\langle \psi_a, \psi_b \rangle$  such that  $\psi_a, \psi_b \in \mathcal{T}_\mathcal{O}$  if there exists

---

a predicate  $p$  such that  $p$  appears in  $RHS(\psi_b)$  and  $p$  appears in  $LHS(\psi_a)$  then  $\langle \psi_a, \psi_b \rangle \in \mathcal{W}(\psi)$ .

The definition of clause graph is analogous to axiom graph but defined over clauses, i.e. the differences are syntactical. Both definitions are needed as analogous operations will be done in different contexts, after and before the syntactical conversion of DL axioms to Horn clauses is performed.

**Definition 4.** We define  $cgraph(\Gamma)$ , the clause graph for a set of clauses  $\Gamma$ , as the directed graph  $\mathcal{G}(\gamma) = \langle \mathcal{V}(\gamma), \mathcal{W}(\gamma) \rangle$  such that: (i) for each clause  $\gamma \in \Gamma$ ,  $\gamma \in \mathcal{V}(\gamma)$ ; and (ii) for each  $\langle \gamma_a, \gamma_b \rangle$  such that  $\gamma_a, \gamma_b \in \Gamma$  if there exists a predicate  $p$  such that  $p$  appears in  $body(\gamma_b)$  and  $p$  appears in  $head(\gamma_a)$  then  $\langle \gamma_a, \gamma_b \rangle \in \mathcal{W}(\gamma)$ .

We now introduce the definition of accessible clauses (from some other clause) which is similar to the definition of reachability. Intuitively, we say that a clause  $\gamma_a$  is accessible from another clause  $\gamma_b$  when  $\gamma_b$  is reachable from  $\gamma_a$ . However, the context in which these definitions will be used is different, as reachability is defined in a set of clauses with respect to a query and accessibility is defined with respect to the values of the atoms in clauses, starting from the extensional values. Therefore, we can intuitively consider that both reachability and accessibility are transitive properties, that differ in the way in which they propagate. In particular, reachability propagates in a backward chaining fashion and accessibility propagates in a forward chaining fashion.

Another relevant concept when considering directed graphs is the notion of Strongly Connected Components (SCCs). Analogously to accessible clauses, we say that a vertex  $\nu_i$  is accessible from some other vertex  $\nu_j$  in a graph  $\mathcal{G}(g) = \langle \mathcal{V}(g), \mathcal{W}(g) \rangle$  when there is an edge  $(\nu_j, \nu_i)$  or there is some other vertex  $\nu_k$ , an edge  $(\nu_j, \nu_k)$  and  $\nu_i$  is accessible from  $\nu_k$ . In a directed graph  $\mathcal{G}(d)$ , the SCCs are the maximal sets of vertexes such that for each SCC  $\mathcal{V}(c)$  every pair of vertexes  $\nu_i, \nu_j \in \mathcal{V}(c)$  are mutually accessible. The vertexes that do not satisfy this property with any other vertex are considered SCCs of cardinality 1. Therefore, the SCCs define a partition of the original graph.

**Definition 5.** We say that a clause  $\gamma_a$  is accessible from another clause  $\gamma_b$  in a set of clauses  $\Gamma$  when: (i) the head predicate of  $\gamma_b$  appears in the body of  $\gamma_a$ , or

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

(ii) there exists some other clause  $\gamma_c$  such that the head predicate of  $\gamma_b$  appears in the body of  $\gamma_c$  and  $\gamma_a$  is accessible from  $\gamma_c$ .

EBox and ABox extensional constraints have been previously defined, as we have seen in section 2.6. However, we introduce a new and equivalent definition for EBox to relate it with the definitions used in the thesis, especially the definitions in section 4.1.

**Definition 6.** *A Horn EBox is a set of clauses  $\mathcal{E}$  that specify containment relations between the values of the predicates in an ABox. These relations are satisfied by an ABox  $\mathcal{A}$  when  $\forall \gamma \in \mathcal{E}. \varphi_{\mathcal{A}}(\gamma) \subseteq v_{\mathcal{A}}^e(\text{pred}(\text{head}(\gamma)))$ .*

There are two things to note in this definition. First, that an EBox is defined according to an ABox that satisfies it, i.e. that fulfills the relations between the predicates expressed by the EBox. In the case of OBDA systems this ABox is normally virtual, i.e. composed of a set of mappings and a data source. Second, that an EBox refers to the extensional values in the ABox and it must not be mistaken with a TBox. An axiom (equivalently a clause) in the TBox expands the values for some predicate with intensional values, while an axiom (equivalently a clause) in the EBox constrains the possible extensional values for some predicate, and therefore the admissible ABoxes. Logically, the DL EBox is the syntactic conversion of the Horn EBox according to the rules in table 2.2, and its definition is analogous.

Finally, we introduce the definition of pure EBox. Intuitively, the pure EBox is composed of the part of the EBox that is composed by predicates that have no intensional definition in the TBox. We will see that this part of the EBox has especial characteristics and properties that we will use. Note that the pure EBox  $\mathcal{E}_p$  is defined not only with respect to an ABox but also with respect to a TBox.

**Definition 7.** *Let  $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$  be an ontology and  $\mathcal{E}$  an EBox satisfied by the ABox  $\mathcal{A}$ . The set of clauses  $\{\gamma_e \in \mathcal{E} \mid \forall \gamma_t \in \mathcal{T}. \text{pred}(\text{head}(\gamma_t)) \notin \text{preds}(\gamma_e)\}$  is the pure EBox, and we represent it with  $\mathcal{E}_p$ .*

Analogous considerations can be made by replacing the ontology  $\mathcal{O}$  with an OBDA system  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$ .

---

## 6.2 The OBDA algorithm for kyrie3

We have seen that an OBDA system superimposes a conceptual layer as a view to an underlying information system, which abstracts away from how that information is maintained in the data layer and may provide inference capabilities. This abstraction usually implies not considering the extension on the predicates that are being handled at each time. In this algorithm we include the use of an EBox to model this metainformation, which can be taken into account for the application of further optimisations.

### 6.2.1 Intuitive description

In this section we describe the algorithm for kyrie3 (figure 6.2). This algorithm applies the optimisations in chapters 4 and 5, together with some additional optimisations for the use of an EBox. We will focus now on the optimisations that involve an EBox.

### 6.2.2 A running example

The optimisations in kyrie3 can be understood more easily with an example, by seeing them into action. Similarly to previous chapters and examples we will describe a familiar and simple TBox with students, professors and courses:

$$\begin{array}{ll} \textit{AssistantProfessor} \sqsubseteq \textit{Professor} & \exists \textit{hasCourseSyllabus.Syllabus} \sqsubseteq \textit{Course} \\ \textit{AssociateProfessor} \sqsubseteq \textit{Professor} & \exists \textit{hasCourseSyllabus}^- \sqsubseteq \textit{Syllabus} \\ \textit{FullProfessor} \sqsubseteq \textit{Professor} & \textit{UndergradStudent} \sqsubseteq \textit{Student} \\ \textit{MasterStudent} \sqsubseteq \textit{GradStudent} & \textit{ApprovedSyllabus} \sqsubseteq \textit{Syllabus} \\ \textit{PhDStudent} \sqsubseteq \textit{GradStudent} & \textit{CourseSyllabus} \sqsubseteq \textit{Syllabus} \\ \textit{Bachelor} \sqsubseteq \textit{UndergradStudent} & \textit{DraftSyllabus} \sqsubseteq \textit{Syllabus} \\ \textit{ImpartedCourse} \sqsubseteq \exists \textit{hasStudent.Student} & \textit{GradStudent} \sqsubseteq \textit{Student} \\ \exists \textit{hasStudent.Student} \sqsubseteq \exists \textit{hasProfessor.Professor} & \textit{Professor} \sqsubseteq \textit{Person} \\ \exists \textit{hasProfessor.Professor} \sqsubseteq \exists \textit{hasCourseSyllabus.Syllabus} & \textit{Student} \sqsubseteq \textit{Person} \\ \textit{PotentialCourse} \sqsubseteq \exists \textit{hasCourseSyllabus.Syllabus} & \end{array}$$

For the current example we will also use an EBox with the following axioms:

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

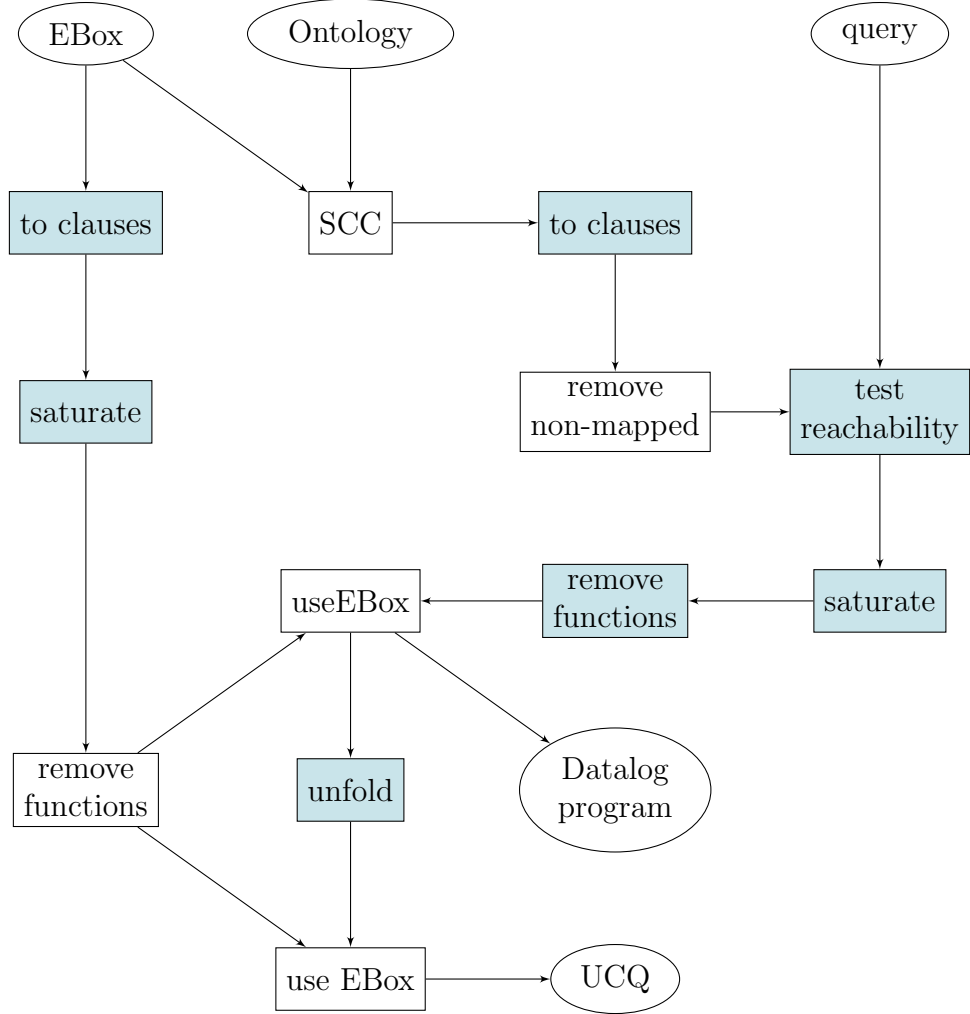


Figure 6.2: Stages in the kyrie3 algorithm, in a different color the parts left unchanged

$Bachelor \sqsubseteq UndergradStudent$	$hasProfessor \sqsubseteq \perp$
$Student \sqsubseteq PhDStudent$	$PotentialCourse \sqsubseteq \perp$
$PhDStudent \sqsubseteq MasterStudent$	$\exists hasStudent^- \sqsubseteq Student$
$FullProfessor \sqsubseteq AssociateProfessor$	$GradStudent \sqsubseteq \perp$

If the TBox was converted to Datalog, as in the previous examples, we would obtain the following set of clauses:

```

Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)
UndergradStudent(?0) <- Bachelor(?0)

```



---

```

    Person(?0) <- Student(?0)
    Student(?0) <- UndergradStudent(?0)
    Student(?0) <- GradStudent(?0)
    Professor(?0) <- AssociateProfessor(?0)
    Syllabus(?0) <- CourseSyllabus(?0)
    Student(f0(?0)) <- ImpartedCourse(?0)
    hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
    AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
    Professor(f1(?0)) <- AUX$0(?0)
    hasProfessor(?0,f1(?0)) <- AUX$0(?0)
    Syllabus(?0) <- hasCourseSyllabus(?1,?0)
    Syllabus(?0) <- ApprovedSyllabus(?0)
    AUX$1(?0) <- Professor(?1), hasProfessor(?0,?1)
    Syllabus(f2(?0)) <- AUX$1(?0)
    hasCourseSyllabus(?0,f2(?0)) <- AUX$1(?0)
    Syllabus(?0) <- DraftSyllabus(?0)
    GradStudent(?0) <- PhDStudent(?0)
    Person(?0) <- Professor(?0)
    GradStudent(?0) <- MasterStudent(?0)
    Professor(?0) <- FullProfessor(?0)
    Syllabus(f3(?0)) <- PotentialCourse(?0)
    hasCourseSyllabus(?0,f3(?0)) <- PotentialCourse(?0)
    Professor(?0) <- AssistantProfessor(?0)

```

However, in this case, we first remove the strongly connected components that can be removed according to proposition 10. In the current example, the axiom  $Bachelor \sqsubseteq UndergradStudent$  can be removed. After removing this axiom, the resulting set of clauses is the following:

```

    Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)
    Student(?0) <- UndergradStudent(?0)

```

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

```
Person(?0) <- Student(?0)
Student(?0) <- GradStudent(?0)
Professor(?0) <- AssociateProfessor(?0)
Syllabus(?0) <- CourseSyllabus(?0)
Student(f0(?0)) <- ImpartedCourse(?0)
hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
AUX$0(?0) <- Student(?1), hasStudent(?0,?1)
Professor(f1(?0)) <- AUX$0(?0)
hasProfessor(?0,f1(?0)) <- AUX$0(?0)
Syllabus(?0) <- hasCourseSyllabus(?1,?0)
Syllabus(?0) <- ApprovedSyllabus(?0)
AUX$1(?0) <- Professor(?1), hasProfessor(?0,?1)
Syllabus(f2(?0)) <- AUX$1(?0)
hasCourseSyllabus(?0,f2(?0)) <- AUX$1(?0)
Syllabus(?0) <- DraftSyllabus(?0)
GradStudent(?0) <- PhDStudent(?0)
Person(?0) <- Professor(?0)
GradStudent(?0) <- MasterStudent(?0)
Professor(?0) <- FullProfessor(?0)
Syllabus(f3(?0)) <- PotentialCourse(?0)
hasCourseSyllabus(?0,f3(?0)) <- PotentialCourse(?0)
Professor(?0) <- AssistantProfessor(?0)
```

Analogously, the EBox is also converted to Datalog. In this case, the axioms containing some bottom entity generate no clauses, and will be considered separately. In particular, we will use a list with the predicates that are mapped, similarly to chapter 4. From the conversion to Datalog we obtain the following clauses:

```
UndergradStudent(?0) <- Bachelor(?0)
MasterStudent(?0) <- PhDStudent(?0)
AssociateProfessor(?0) <- FullProfessor(?0)
PhDStudent(?0) <- Student(?0)
Student(?0) <- hasStudent(?1,?0)
```

---

Analogously to previous chapters, we perform a first saturation step on the Datalog generated from the TBox.

$$\begin{array}{l} \text{Syllabus}(\text{?0}) \leftarrow \text{hasCourseSyllabus}(\text{?1}, \text{?0}) \\ \text{hasCourseSyllabus}(\text{?0}, \text{f3}(\text{?0})) \leftarrow \text{PotentialCourse}(\text{?0}) \\ \hline \therefore \text{Syllabus}(\text{f3}(\text{?0})) \leftarrow \text{PotentialCourse}(\text{?0}) \end{array}$$

$$\begin{array}{l} \text{Syllabus}(\text{?0}) \leftarrow \text{hasCourseSyllabus}(\text{?1}, \text{?0}) \\ \text{hasCourseSyllabus}(\text{?0}, \text{f2}(\text{?0})) \leftarrow \text{AUX\$1}(\text{?0}) \\ \hline \therefore \text{Syllabus}(\text{f2}(\text{?0})) \leftarrow \text{AUX\$1}(\text{?0}) \end{array}$$

$$\begin{array}{l} \text{Person}(\text{?0}) \leftarrow \text{Professor}(\text{?0}) \\ \text{Professor}(\text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0}) \\ \hline \therefore \text{Person}(\text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0}) \end{array}$$

$$\begin{array}{l} \text{Person}(\text{?0}) \leftarrow \text{Student}(\text{?0}) \\ \text{Student}(\text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0}) \\ \hline \therefore \text{Person}(\text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0}) \end{array}$$

$$\begin{array}{l} \text{AUX\$1}(\text{?0}) \leftarrow \text{Professor}(\text{?1}), \text{hasProfessor}(\text{?0}, \text{?1}) \\ \text{hasProfessor}(\text{?0}, \text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0}) \\ \hline \therefore \text{AUX\$1}(\text{?0}) \leftarrow \text{AUX\$0}(\text{?0}), \text{Professor}(\text{f1}(\text{?0})) \end{array}$$

$$\begin{array}{l} \text{AUX\$1}(\text{?0}) \leftarrow \text{AUX\$0}(\text{?0}), \text{Professor}(\text{f1}(\text{?0})) \\ \text{Professor}(\text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0}) \\ \hline \therefore \text{AUX\$1}(\text{?0}) \leftarrow \text{AUX\$0}(\text{?0}) \end{array}$$

$$\begin{array}{l} \text{AUX\$0}(\text{?0}) \leftarrow \text{Student}(\text{?1}), \text{hasStudent}(\text{?0}, \text{?1}) \\ \text{hasStudent}(\text{?0}, \text{f0}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0}) \\ \hline \therefore \text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0}), \text{Student}(\text{f0}(\text{?0})) \end{array}$$

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

```
AUX$0(?0) <- ImpartedCourse(?0), Student(f0(?0))  
Student(f0(?0)) <- ImpartedCourse(?0)  
-----  
∴ AUX$0(?0) <- ImpartedCourse(?0)
```

```
Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)  
hasCourseSyllabus(?0,f3(?0)) <- PotentialCourse(?0)  
-----  
∴ Course(?0) <- PotentialCourse(?0), Syllabus(f3(?0))
```

```
Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)  
hasCourseSyllabus(?0,f2(?0)) <- AUX$1(?0)  
-----  
∴ Course(?0) <- AUX$1(?0), Syllabus(f2(?0))
```

```
Course(?0) <- AUX$1(?0), Syllabus(f2(?0))  
Syllabus(f2(?0)) <- AUX$1(?0)  
-----  
∴ Course(?0) <- AUX$1(?0)
```

```
Course(?0) <- PotentialCourse(?0), Syllabus(f3(?0))  
Syllabus(f3(?0)) <- PotentialCourse(?0)  
-----  
∴ Course(?0) <- PotentialCourse(?0)
```

After that saturation step is finished, we can continue by removing non-mapped predicates. This is analogous to how auxiliary predicates were removed in chapter 5, but considering non-mapped predicates as in chapter 4.

```
AUX$1(?0) <- AUX$0(?0)  
AUX$0(?0) <- ImpartedCourse(?0)  
-----  
∴ AUX$1(?0) <- ImpartedCourse(?0)
```

```
Course(?0) <- AUX$1(?0)  
AUX$1(?0) <- ImpartedCourse(?0)  
-----
```

---



---

$\therefore \text{Course}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\text{Person}(\text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0})$ $\text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{Person}(\text{f1}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\text{Professor}(\text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0})$ $\text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{Professor}(\text{f1}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\text{hasProfessor}(\text{?0}, \text{f1}(\text{?0})) \leftarrow \text{AUX\$0}(\text{?0})$ $\text{AUX\$0}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{hasProfessor}(\text{?0}, \text{f1}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\text{Syllabus}(\text{f2}(\text{?0})) \leftarrow \text{AUX\$1}(\text{?0})$ $\text{AUX\$1}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{Syllabus}(\text{f2}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\text{hasCourseSyllabus}(\text{?0}, \text{f2}(\text{?0})) \leftarrow \text{AUX\$1}(\text{?0})$ $\text{AUX\$1}(\text{?0}) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\therefore \text{hasCourseSyllabus}(\text{?0}, \text{f2}(\text{?0})) \leftarrow \text{ImpartedCourse}(\text{?0})$
$\text{Student}(\text{?0}) \leftarrow \text{GradStudent}(\text{?0})$ $\text{GradStudent}(\text{?0}) \leftarrow \text{PhDStudent}(\text{?0})$
$\therefore \text{Student}(\text{?0}) \leftarrow \text{PhDStudent}(\text{?0})$
$\text{Student}(\text{?0}) \leftarrow \text{GradStudent}(\text{?0})$ $\text{GradStudent}(\text{?0}) \leftarrow \text{MasterStudent}(\text{?0})$
$\therefore \text{Student}(\text{?0}) \leftarrow \text{MasterStudent}(\text{?0})$
$\text{AUX\$1}(\text{?0}) \leftarrow \text{AUX\$0}(\text{?0})$ $\text{AUX\$0}(\text{?0}) \leftarrow \text{Student}(\text{?1}), \text{hasStudent}(\text{?0}, \text{?1})$

---

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

---

$\therefore \text{AUX\$1(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

$\text{Person(f1(?0))} \leftarrow \text{AUX\$0(?0)}$

$\text{AUX\$0(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

---

$\therefore \text{Person(f1(?0))} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

$\text{Professor(f1(?0))} \leftarrow \text{AUX\$0(?0)}$

$\text{AUX\$0(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

---

$\therefore \text{Professor(f1(?0))} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

$\text{hasProfessor(?0,f1(?0))} \leftarrow \text{AUX\$0(?0)}$

$\text{AUX\$0(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

---

$\therefore \text{hasProfessor(?0,f1(?0))} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

$\text{Course(?0)} \leftarrow \text{AUX\$1(?0)}$

$\text{AUX\$1(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

---

$\therefore \text{Course(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

$\text{Syllabus(f2(?0))} \leftarrow \text{AUX\$1(?0)}$

$\text{AUX\$1(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

---

$\therefore \text{Syllabus(f2(?0))} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

$\text{hasCourseSyllabus(?0,f2(?0))} \leftarrow \text{AUX\$1(?0)}$

$\text{AUX\$1(?0)} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

---

$\therefore \text{hasCourseSyllabus(?0,f2(?0))} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

$\text{AUX\$1(?0)} \leftarrow \text{Professor(?1), hasProfessor(?0,?1)}$

$\text{hasProfessor(?0,f1(?0))} \leftarrow \text{ImpartedCourse(?0)}$

---

$\therefore \text{AUX\$1(?0)} \leftarrow \text{ImpartedCourse(?0), Professor(f1(?0))}$

$\text{AUX\$1(?0)} \leftarrow \text{Professor(?1), hasProfessor(?0,?1)}$

$\text{hasProfessor(?0,f1(?0))} \leftarrow \text{Student(?1), hasStudent(?0,?1)}$

---

---



---

```

 $\therefore$  AUX$1(?0) <- Professor(f1(?0)), Student(?1),
      hasStudent(?0,?1)

```

The preprocessing removes some non-mapped predicates. Intuitively, the operation is composed of two stages. First, non-mapped predicates are unified and new clauses are generated. If there are cycles in the clauses then one predicate for each cycle must be excluded from the resolution to avoid generating an infinite number of clauses. Second, we eliminate the clauses that contain in the body the predicates that have participated in the resolution. These clauses are removed because they are redundant with respect to the clauses generated in the first stage. The correctness of this operation is proven in proposition 8. Those clauses that contain these predicates in the head are preserved, as we cannot check for reachability during the preprocessing (the query could specify any of these predicates). The result is the following set of clauses:

```

      Course(?0) <- ImpartedCourse(?0)
      Student(?0) <- UndergradStudent(?0)
      Person(f0(?0)) <- ImpartedCourse(?0)
      Person(?0) <- Student(?0)
      Professor(?0) <- AssociateProfessor(?0)
      Syllabus(?0) <- CourseSyllabus(?0)
      Student(f0(?0)) <- ImpartedCourse(?0)
      hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
      Person(f1(?0)) <- ImpartedCourse(?0)
      Professor(f1(?0)) <- ImpartedCourse(?0)
      hasProfessor(?0,f1(?0)) <- ImpartedCourse(?0)
      Syllabus(?0) <- hasCourseSyllabus(?1,?0)
      Syllabus(?0) <- ApprovedSyllabus(?0)
      Syllabus(f2(?0)) <- ImpartedCourse(?0)
      hasCourseSyllabus(?0,f2(?0)) <- ImpartedCourse(?0)
      Syllabus(?0) <- DraftSyllabus(?0)

```

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

```
GradStudent(?0) <- PhDStudent(?0)
Student(?0) <- PhDStudent(?0)
Person(?0) <- Professor(?0)
GradStudent(?0) <- MasterStudent(?0)
Student(?0) <- MasterStudent(?0)
Professor(?0) <- FullProfessor(?0)
Professor(?0) <- AssistantProfessor(?0)
Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)
hasProfessor(?0,f1(?0)) <- Student(?1), hasStudent(?0,?1)
Professor(f1(?0)) <- Student(?1), hasStudent(?0,?1)
Person(f1(?0)) <- Student(?1), hasStudent(?0,?1)
hasCourseSyllabus(?0,f2(?0)) <- Student(?1), hasStudent(?0,?1)
Syllabus(f2(?0)) <- Student(?1), hasStudent(?0,?1)
Course(?0) <- Student(?1), hasStudent(?0,?1)
```

The EBox is analogously preprocessed, obtaining the following set of clauses:

```
Student(?0) <- hasStudent(?1,?0)
PhDStudent(?0) <- Student(?0)
PhDStudent(?0) <- hasStudent(?1,?0)
AssociateProfessor(?0) <- FullProfessor(?0)
MasterStudent(?0) <- PhDStudent(?0)
MasterStudent(?0) <- hasStudent(?1,?0)
MasterStudent(?0) <- Student(?0)
UndergradStudent(?0) <- Bachelor(?0)
```

Once the query is available we can run a reachability test on the clauses, as in previous chapters. In this case we will use the query  $Q(?0) \leftarrow Course(?0)$ , obtaining the following set of clauses:



---

```

        Course(?0) <- ImpartedCourse(?0)
        Student(?0) <- UndergradStudent(?0)
        Syllabus(?0) <- CourseSyllabus(?0)
        Student(f0(?0)) <- ImpartedCourse(?0)
        hasStudent(?0,f0(?0)) <- ImpartedCourse(?0)
        Syllabus(?0) <- hasCourseSyllabus(?1,?0)
        Syllabus(?0) <- ApprovedSyllabus(?0)
        Syllabus(f2(?0)) <- ImpartedCourse(?0)
        hasCourseSyllabus(?0,f2(?0)) <- ImpartedCourse(?0)
        Syllabus(?0) <- DraftSyllabus(?0)
        Student(?0) <- PhDStudent(?0)
        Student(?0) <- MasterStudent(?0)
        Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)
        hasCourseSyllabus(?0,f2(?0)) <- Student(?1), hasStudent(?0,?1)
        Syllabus(f2(?0)) <- Student(?1), hasStudent(?0,?1)
        Course(?0) <- Student(?1), hasStudent(?0,?1)

```

The previous set of clauses is saturated with the query, as in previous chapters. Subsumption checks as explained in section 5.2.2 eliminate several clauses, obtaining the following set of clauses:

```

Q(?0) <- Course(?0)
Course(?0) <- ImpartedCourse(?0)
Student(?0) <- UndergradStudent(?0)
Syllabus(?0) <- CourseSyllabus(?0)
Syllabus(?0) <- hasCourseSyllabus(?1,?0)
Syllabus(?0) <- ApprovedSyllabus(?0)
Syllabus(?0) <- DraftSyllabus(?0)
Student(?0) <- PhDStudent(?0)
Student(?0) <- MasterStudent(?0)
Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)
Course(?0) <- Student(?1), hasStudent(?0,?1)

```

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

In previous algorithms this would be the final Datalog program, which could later be unfolded into a union of conjunctive queries. In this case we use the knowledge in the EBox to remove the clauses that provide redundant answers to the query. In particular, the inferences that are done are the following:

```
Student(?0) <- UndergradStudent(?0)
UndergradStudent(?0) <- Bachelor(?0)


---


∴ Student(?0) <- Bachelor(?0)
```

```
Student(?0) <- PhDStudent(?0)
PhDStudent(?0) <- hasStudent(?1,?0)


---


∴ Student(?0) <- hasStudent(?1,?0)
```

```
Student(?0) <- MasterStudent(?0)
MasterStudent(?0) <- PhDStudent(?0)


---


∴ Student(?0) <- PhDStudent(?0)
```

```
Student(?0) <- MasterStudent(?0)
MasterStudent(?0) <- hasStudent(?1,?0)


---


∴ Student(?0) <- hasStudent(?1,?0)
```

```
Student(?0) <- MasterStudent(?0)
MasterStudent(?0) <- PhDStudent(?0)


---


∴ Student(?0) <- PhDStudent(?0)
```

```
Student(?0) <- MasterStudent(?0)
MasterStudent(?0) <- hasStudent(?1,?0)


---


∴ Student(?0) <- hasStudent(?1,?0)
```

```
Student(?0) <- UndergradStudent(?0)
UndergradStudent(?0) <- Bachelor(?0)


---


∴ Student(?0) <- Bachelor(?0)
```

---

The EBox contains the axiom  $PhDStudent \sqsubseteq MasterStudent$ , which means that the individuals mapped for *MasterStudent* contain all the individuals mapped for the predicate *PhDStudent*. Therefore, the clause `Student(?0) <- MasterStudent(?0)` subsumes the clause `Student(?0) <- PhDStudent(?0)`, which can be removed. We may observe similar axioms that could trigger similar operations due to extensional containment. However, the predicates involved in these axioms have intensional definitions that prevent their removal at this moment. For example, the EBox contains the axiom,  $Student \sqsubseteq PhDStudent$ , however *Student* has an intensional definition which may imply individuals for *Student* that are not *PhDStudent*, hence it is not replaceable at this point in time.

```

Q(?0) <- Course(?0)
Course(?0) <- ImpartedCourse(?0)
Student(?0) <- UndergradStudent(?0)
Syllabus(?0) <- CourseSyllabus(?0)
Syllabus(?0) <- hasCourseSyllabus(?1,?0)
Syllabus(?0) <- ApprovedSyllabus(?0)
Syllabus(?0) <- DraftSyllabus(?0)
Student(?0) <- PhDStudent(?0)
Student(?0) <- MasterStudent(?0)
Course(?0) <- Syllabus(?1), hasCourseSyllabus(?0,?1)
Course(?0) <- Student(?1), hasStudent(?0,?1)

```

Once the Datalog program has been obtained, it is unfolded, as discussed in previous chapters, obtaining the following UCQ:

```

Q(?0) <- Course(?0)
Q(?0) <- ImpartedCourse(?0)
Q(?0) <- hasCourseSyllabus(?0,?1)
Q(?0) <- Student(?1), hasStudent(?0,?1)
Q(?0) <- UndergradStudent(?1), hasStudent(?0,?1)
Q(?0) <- MasterStudent(?1), hasStudent(?0,?1)

```

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

Again, we can consider the information available in the EBox to eliminate some redundant clauses in this UCQ. The inferences needed in this case are not many:

$$\begin{array}{l} Q(?0) \leftarrow \text{MasterStudent}(?1), \text{hasStudent}(?0,?1) \\ \text{MasterStudent}(?0) \leftarrow \text{PhDStudent}(?0) \\ \hline \therefore Q(?0) \leftarrow \text{PhDStudent}(?1), \text{hasStudent}(?0,?1) \end{array}$$
$$\begin{array}{l} Q(?0) \leftarrow \text{MasterStudent}(?1), \text{hasStudent}(?0,?1) \\ \text{MasterStudent}(?0) \leftarrow \text{hasStudent}(?1,?0) \\ \hline \therefore Q(?0) \leftarrow \text{hasStudent}(?0,?1), \text{hasStudent}(?2,?1) \end{array}$$
$$\begin{array}{l} Q(?0) \leftarrow \text{MasterStudent}(?1), \text{hasStudent}(?0,?1) \\ \text{MasterStudent}(?0) \leftarrow \text{Student}(?0) \\ \hline \therefore Q(?0) \leftarrow \text{Student}(?1), \text{hasStudent}(?0,?1) \end{array}$$

After using the EBox information with the original UCQ we obtain a significantly reduced UCQ, whose answers are equivalent (under set semantics) to the original:

$$\begin{array}{l} Q(?0) \leftarrow \text{hasCourseSyllabus}(?0,?1) \\ Q(?0) \leftarrow \text{ImpartedCourse}(?0) \\ Q(?0) \leftarrow \text{Course}(?0) \end{array}$$

The answers to this UCQ are equivalent (under set semantics) to the previous one. They are also equivalent to those that would have been obtained without considering an EBox:

$$\begin{array}{l} Q(?0) \leftarrow \text{Course}(?0) \\ Q(?0) \leftarrow \text{PotentialCourse}(?0) \\ Q(?0) \leftarrow \text{ImpartedCourse}(?0) \\ Q(?0) \leftarrow \text{hasCourseSyllabus}(?0,?1) \end{array}$$

---

```

Q(?0) <- Student(?1), hasStudent(?0,?1)
Q(?0) <- UndergradStudent(?1), hasStudent(?0,?1)
Q(?0) <- Bachelor(?1), hasStudent(?0,?1)
Q(?0) <- GradStudent(?1), hasStudent(?0,?1)
Q(?0) <- PhDStudent(?1), hasStudent(?0,?1)
Q(?0) <- MasterStudent(?1), hasStudent(?0,?1)
Q(?0) <- Professor(?1), hasProfessor(?0,?1)
Q(?0) <- AssociateProfessor(?1), hasProfessor(?0,?1)
Q(?0) <- FullProfessor(?1), hasProfessor(?0,?1)
Q(?0) <- AssistantProfessor(?1), hasProfessor(?0,?1)

```

### 6.2.3 Pseudocode of the algorithms

In this section we explain the *kyrie3* algorithm, based on previous algorithms. The chapter focuses on explaining the main novelty of the algorithm, which is the use of extensional constraints. Extensional constraints can be used both in the preprocessing stage, which is performed before queries are posed to the system, and in the main algorithm for the rewriting of these queries when they are available. We conclude this section with the algorithm that prunes a Datalog program (or a UCQ as a particular case of Datalog program) using the available extensional constraints.

The algorithm in *kyrie3* extends the earlier *kyrie2* algorithm to handle EBoxes. The original *kyrie2* algorithm obtains a set of clauses  $\Sigma$  from the TBox  $\mathcal{T}$  and a query  $q$  and performs resolution on this set of clauses.

The usual operations performed in these algorithms are equal to those in *kyrie2*, as seen in the previous chapter:

- *Saturate* performs a saturation of a set of clauses using a selection function to guide the atoms that should be unified in the resolution. The selection function may be:
  - *sfRQR* is the selection function used in REQUIEM, it avoids the unification of unary predicates with no function symbols to produce a Datalog program.

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

- *sfAux* selects auxiliary predicates to perform the inferences in which these predicates may participate and remove them if possible.
- *sfSel(p)* selects the predicate *p*.
- *sfNoRec(P)* selects all predicates except those that included in the set of predicates *P* (used to avoid infinite resolution on recursive predicates).

As explained in section 5.1.3, the saturation algorithm has a parameter (*p*, *s* or *u*) to specify the way in which the clauses should be treated.

- *Condensate* is used to condensate clauses, i.e. remove redundant atoms.
- *RemoveSubsumed* removes clauses that are subsumed in a set of clauses.

There are three main stages in which resolution is performed:

- *Preprocessing* is performed once for the  $\mathcal{ELHIJ}$  TBox ( $\mathcal{T}$ ), before any query is posed to the system. In this stage some inferences are materialised to save time later and the set of clauses  $\Sigma$  is generated according to the TBox.
- *Saturation* is performed when the query arrives: the query is added to  $\Sigma$ , then functional symbols are removed from  $\Sigma$ , reducing  $\Sigma$  to a Datalog program (i.e., a function-free set of Horn clauses).
- *Unfolding* is performed partially or completely, depending on the respective presence or absence of cycles in the Datalog rewriting.

In *kyrie3*, we add a further operation in each of these stages, highlighted in the corresponding algorithms, this is the operation that will be detailed in this chapter. The new operation makes use of the EBox to infer extensional subsumption between atoms and between clauses. Atom subsumption in a conjunction of atoms means that the values for one are a subset of the values for another (the most general atom is eliminated from the conjunction). Clause subsumption in a disjunction of clauses means that the values provided by a clause for a predicate are a subset of the values provided by some other clause (the most specific clause is eliminated from the disjunction). In other words, the new operation detects

---

extensional redundancy in the set of clauses, thus allowing for reducing the size of the initial set of clauses, the subsequent Datalog program, and the final UCQ. Since, for technical reasons, the EBox is represented in two different ways in the algorithm (both as a set of standard DL axioms and as a set of clauses), this operation is defined and executed on both representations.

Our algorithms will use both the DL and the clause representation of TBoxes and EBoxes (obtained from the DL syntax through the function *toHorn*). Therefore, from now on we will use the terms TBox, EBox and OBDA system for both kinds of representations, and will denote a TBox and an EBox as  $\Sigma$  and  $E$  (respectively  $\mathcal{T}_\emptyset$  and  $\mathcal{E}_\emptyset$ ) to denote a TBox and an EBox as Horn clauses (and respectively as DL axioms).

Algorithm 6.1 implements a preprocessing stage on the TBox and the EBox before any query is available. The algorithm removes, through the function *delEBoxSCC*, the strongly connected components (SCCs) of the TBox graph that are implied by the EBox and do not receive incoming connections, i.e. for all axioms  $\psi_b$  in the SCC there is no  $\psi_a$  in the TBox such that  $(\psi_a, \psi_b)$  is in the set of edges of *dlgraph*( $\mathcal{T}_\emptyset$ ) (definition 3). The correctness for the removal of SCCs is proven in proposition 10.

---

**Algorithm 6.1:** kyrie3 preprocess algorithm

---

**Input:**  $\mathcal{ELHIJ}$  TBox  $\mathcal{T}_\emptyset$ ,  $\mathcal{ELHIJ}$  EBox  $\mathcal{E}_\emptyset$

**Output:** TBox  $\Sigma$ , EBox  $E$ , minimal sets of recursive predicates  $R_\Sigma$  and  $R_E$

- 1  $\mathcal{T}'_\emptyset = \text{delEBoxSCC}(\mathcal{T}_\emptyset, \mathcal{E}_\emptyset)$
  - 2  $\Sigma = \text{toHorn}(\mathcal{T}'_\emptyset)$
  - 3  $E = \text{saturate}(\mathbf{p}, \text{sfNonRec}(R_E), \text{toHorn}(\mathcal{E}_\emptyset), \emptyset)$
  - 4  $\langle R_\Sigma, R_E \rangle = \text{reducedRecursiveSets}(\Sigma, E)$
  - 5  $\Sigma = \text{saturate}(\mathbf{p}, \text{sfRQR}, \Sigma, \emptyset)$
  - 6  $\Sigma = \text{saturate}(\mathbf{s}, \text{sfAux}, \Sigma, \emptyset)$
  - 7  $\Sigma = \text{removeSubsumed}(\text{condensate}(\Sigma))$
  - 8 **return**  $\langle \Sigma, E, R_\Sigma, R_E \rangle$
- 

Through the function *saturate*, algorithm 6.1 computes the deductive closure of the EBox, except for recursive predicates: a reduced set of recursive predicates is excluded from the inference to make the saturation process finite. This reduced

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---



---

**Algorithm 6.2:** Remove extensionally implied strongly connected components (SCCs) of axioms: **delEBoxSCC**

---

**Input:**  $\mathcal{ELHJO}$  TBox  $\mathcal{T}$ ,  $\mathcal{ELHJO}$  EBox  $\mathcal{E}$

**Output:**  $\mathcal{ELHJO}$  TBox  $\mathcal{T}$  without extensionally implied SCCs

---

```

1 repeat
2   forall the  $(C : component) \in SCCs(dlgraph(\mathcal{T}))$  do
3     if  $incomingConnections(C) = 0 \wedge \forall \psi \in C. \mathcal{E} \models \psi$  then
4        $\mathcal{T} = \mathcal{T} \setminus C$ 
5     end
6   end
7 until Fixpoint
8 return  $\mathcal{T}$ 

```

---

set is computed by algorithm 6.4, where  $count(p, \theta) = card(\{\theta \in \Theta \mid p \in \theta\})$  and  $LoopsIn(\Gamma)$  finds the cycles of clauses (section 2.1.3) in the given set of clauses.

Excluding some predicates from the inference limits the effect of the EBox in the reduction of the rewritten queries and the possible unfolding of these queries. This limited effect of the EBox means that some redundant answers may be produced, which has obviously no negative effect on the correctness of the answers.

The result of the preprocessing stage is then used by the general *kyrie3* algorithm (algorithm 6.3). This algorithm preserves the same stages and optimisations of *kyrie2* to obtain the Datalog program and the unfolding. The main difference with *kyrie2* is the call to the function **useEBox** (algorithm 6.5).

The **useEBox** function, defined by algorithm 6.5, uses the EBox to reduce a Datalog program. This can be done by removing clauses or by replacing some of the clauses with other shorter ones. This algorithm performs a set of stages iteratively to transform the Datalog program considered, until a fixpoint is reached. These stages are:

- Predicates with no extension ( $p \sqsubseteq \perp$  in the EBox) are removed, if possible, after saturating the inferences where they participate. A reduced set of recursive predicates (selected according to algorithm 6.4) needs to be kept. The correctness for this process is proved by proposition 8.
- Clauses whose answers are subsumed by other clauses are removed. The



---

**Algorithm 6.3:** General kyrie3 algorithm

---

**Input:** Horn TBox  $\Sigma$ , Horn EBox  $E$ , recursive predicates in  $\Sigma$  ( $R_\Sigma$ ),  
recursive predicates in  $E$  ( $R_E$ ), UCQ  $q$ , working mode  
 $mode \in \{\text{Datalog}, \text{UCQ}\}$

**Output:** Rewritten query  $q_\Sigma$

```
1  $q = \text{removeSubsumed}(\text{condensate}(q))$ 
2  $\Sigma_r = \text{reachable}(\Sigma, q)$ 
3  $\Sigma_q = \text{saturate}(\mathbf{s}, \text{sfRQR}, q, \Sigma_r)$ 
4  $\Sigma_q = \text{useEBox}(\Sigma_q, E, R_\Sigma, R_E)$ 
5 if  $mode = \text{Datalog}$  then return  $\Sigma_q$ 
6  $\Sigma_q = \{q_i \in \Sigma_q \mid \text{head}(q_i) \neq \text{head}(q)\}$ 
7  $\Sigma_q = \{q_i \in \Sigma_q \mid \text{head}(q_i) = \text{head}(q)\}$ 
8  $\Sigma_q = \text{saturate}(\mathbf{u}, \text{sfNonRec}(R_\Sigma), \Sigma_q, \Sigma_q)$ 
9  $\Sigma_q = \text{useEBox}(\Sigma_q, E, R_\Sigma, R_E)$ 
10 return  $\Sigma_q$ 
```

---

subsumption of the answers according to the algorithm is proven in proposition 5 and therefore they are redundant, as proposition 6 states.

- Clauses where an atom subsumes another atom are condensed. We use resolution to find the condensed version of the clause, which subsumes the original clause. Due to propositions 4 and 7 we know that we can keep any of both clauses. We keep the condensed version, the subsuming clause, since this clause will more likely subsume some other clauses, which in turn we will be able to remove.
- SCCs that are implied by the EBox and receive no connections are removed. This is done according to proposition 10 for the preprocessing and proposition 9 for the Datalog and UCQ rewritings.

## 6.3 Optimisations

The algorithms described in the previous section group a series of operations that are described into more detail in this section. These optimisations are formalised in the next section and their correctness is formally proven.

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

**Algorithm 6.4:** Find reduced sets of recursive predicates:

**reducedRecursiveSet**

---

**Input:** Datalog program  $\Sigma_q$ , Datalog EBox  $E$

**Output:** Recursive predicates in  $\Sigma_q$  ( $R_{\Sigma_q}$ ), recursive predicates in  $E$  ( $R_E$ )

```

1  $\Lambda_{\Sigma_q} = \text{loopsIn}(\Sigma_q)$ 
2  $\Lambda_E = \text{loopsIn}(E)$ 
3  $R_{\Sigma_q} = \emptyset$ 
4  $R_E = \emptyset$ 
5 while  $\Lambda_{\Sigma_q} \neq \emptyset \wedge \Lambda_E \neq \emptyset$  do
6   if  $\Lambda_{\Sigma_q} \cap \Lambda_E \neq \emptyset$  then
7      $p = p_i \in \Lambda_{\Sigma_q} \cap \Lambda_E / \text{count}(p_i, \Lambda_{\Sigma_q}) + \text{count}(p_i, \Lambda_E) =$ 
        $\text{max}(\text{count}(p_j, \Lambda_{\Sigma_q}) + \text{count}(p_j, \Lambda_E)) \forall p_j \in \Lambda_{\Sigma_q} \cap \Lambda_E$ 
8      $R_{\Sigma_q} = R_{\Sigma_q} \cup \{p\}$ 
9      $R_E = R_E \cup \{p\}$ 
10     $\Lambda_{\Sigma_q} = \Lambda_{\Sigma_q} \setminus \{\lambda \in \Lambda_{\Sigma_q} \mid p \in \lambda\}$ 
11     $\Lambda_E = \Lambda_E \setminus \{\lambda \in \Lambda_E \mid p \in \lambda\}$ 
12  else
13    if  $\Lambda_{\Sigma_q} \neq \emptyset$  then
14       $p = p_i \in \Lambda_{\Sigma_q} / \text{count}(p_i, \Lambda_{\Sigma_q}) = \text{max}(\text{count}(p_j, \Lambda_{\Sigma_q})) \forall p_j \in$ 
         $\Lambda_{\Sigma_q}$ 
15       $R_{\Sigma_q} = R_{\Sigma_q} \cup \{p\}$ 
16       $\Lambda_{\Sigma_q} = \Lambda_{\Sigma_q} \setminus \{\lambda \in \Lambda_{\Sigma_q} \mid p \in \lambda\}$ 
17    end
18    if  $\Lambda_E \neq \emptyset$  then
19       $p = p_i \in \Lambda_E / \text{count}(p_i, \Lambda_E) = \text{max}(\text{count}(p_j, \Lambda_E)) \forall p_j \in$ 
         $\Lambda_E$ 
20       $R_E = R_E \cup \{p\}$ 
21       $\Lambda_E = \Lambda_E \setminus \{\lambda \in \Lambda_E \mid p \in \lambda\}$ 
22    end
23  end
24 end
25 return  $\langle R_{\Sigma_q}, R_E \rangle$ 

```

---

### 6.3.1 Deletion of strongly connected components

The strongly connected components (SCCs) that are entailed by the EBox and are not accessible from any other SCC can be deleted. This operation is done in two different contexts.

---

**Algorithm 6.5:** Prune Datalog program  $\Sigma_q$ : **useEBox**

---

**Input:** EBox  $E$ , TBox  $\Sigma_q$ , recursive predicates in  $E$  ( $R_E$ ), recursive predicates in  $\Sigma_q$  ( $R_{\Sigma_q}$ )  
**Output:** Pruned TBox program  $\Sigma_q$

```
1 repeat
2    $P_e = \{p_i \mid p_i \in \text{predicates}(\Sigma_q) \wedge (p_i \sqsubseteq \perp) \in E \wedge p_i \notin R_{\Sigma_q}\}$ 
3    $\Sigma_q = \text{saturate}(\mathbf{u}, \mathbf{sfSel}(P_e), \Sigma_q, \emptyset)$ 
4    $E_e = \{\gamma_i \in E \mid \forall p \in \gamma_i. \forall \gamma_j \in \Sigma_q. p \notin \text{head}(\gamma_j)\}$ 
5    $E_{\Sigma_q} = \emptyset$ 
6   forall the clauses  $\gamma_1 \in \Sigma_q \cup E_{\Sigma_q}$  do
7     forall the clauses  $\gamma_2 \in E_e$  do
8        $\Gamma = \text{resolve}(\gamma_1, \gamma_2, \mathbf{sfNoRec}(R_E))$ 
9       forall the  $\gamma_i \in \Gamma, \gamma_i \neq \gamma_1$  do
10        forall the  $\gamma_j \in \Sigma_q$  do
11          if  $\text{subsumes}(\gamma_i, \gamma_j)$  then
12             $\Sigma_q = \Sigma_q \setminus \{\gamma_j\}$ 
13            if  $\neg \text{subsumes}(\gamma_j, \gamma_i)$  then
14               $\Sigma_q = \Sigma_q \cup \{\gamma_i\}$ 
15            end
16          end
17        end
18      end
19       $E_{\Sigma_q} = E_{\Sigma_q} \cup \Gamma$ 
20    end
21  end
22  forall the  $C \in \text{stronglyConnectedComponents}(\text{cgraph}(\Sigma_q))$  do
23    if  $(\text{incomingConnections}(C) = 0 \wedge \forall \gamma \in C. \gamma \in E_e)$  then
24       $\Sigma_q = \Sigma_q \setminus C$ 
25    end
26  end
27   $\Sigma_q = \text{reachable}(\Sigma_q)$ 
28 until Fixpoint
29 return  $\Sigma_q$ 
```

---

In the first case, this deletion is the first operation that is performed in the process. The deletion is performed directly on DL axioms, modeling them as a DL graph (definition 3). In the second case, this deletion is performed on the clauses in the Datalog program, which are modeled as a clause graph (definition 4). This

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

second context is considered in two different stages: the first one is considered after the Datalog rewriting has been performed, and the second one is considered after the unfolding to the UCQ has been attempted. In both cases, the deletion of SCCs is done recursively, as deleting some SCC may enable the deletion of some other SCC.

For an intuitive explanation let us consider the deletion of a single clause. If the clause cannot be accessed from other clauses, then the predicates in the body are purely extensional (they have no intensional definition). If the clause is entailed by the EBox, then the extensional values for the head atom contain the extensional values for the body of the clause. Considering the values for the atoms in the body of such a clause, none has an impact on the values for the head atom. There are no intensional values (the SCC cannot be accessed) and the extensional ones are redundant with respect to the head atom (as implied by the EBox). Therefore, the clause can be deleted safely, i.e. without losing answers to the queries.

We can extend the previous intuitive idea by induction. If the considered clause  $\gamma_a$  is entailed by the EBox, but it can be accessed from some second clause  $\gamma_b$ , then this second clause  $\gamma_b$  may imply some intensional values for the body atoms of  $\gamma_a$ , and therefore also for the head atom of  $\gamma_a$ . However, if  $\gamma_b$  could be removed and no other clause accessed to  $\gamma_a$ , then  $\gamma_a$  would not be accessed by any clause and could be removed as in the previous paragraph. Consequently, we can apply this principle to sequences of clauses deleting removing finite sequences of them. In the case of cycles (or strongly connected components), we can consider them as an infinite sequence of clauses at the intensional level. In such a case, we have to remember that we are considering the extensional properties of these clauses. In fact, cycles do not translate to infinite sequences of extensional implications, they translate to indefinitely long but necessarily finite sequences of implications at the extensional level. Therefore, we can remove the clauses in such SCCs safely.

Note that theoretically the deletion of SCCs does not need to be recursive for the same reasons. All SCCs with such characteristics can be removed in one step. The reason to perform this deletion recursively is to keep the process simple, for implementation, execution, validation and explanation purposes.

---

The explanation for the case of DL axioms is analogous, considering the clause graph in such a case.

### 6.3.2 Removal of predicates with empty extension

We have already seen in chapter 4 that the predicates that have an empty extension can be removed. We have also seen that these optimisations have greater benefits the sooner they are applied, especially as seen in chapter 5. In this case we adapt the optimisation detailed in chapter 4 for the removal of non-mapped predicates to apply it in the preprocessing stage added in chapter 5.

In the preprocessing step the same resolution and optimisation that we have already seen for the Datalog programs in chapter 4 can be applied to the FOL clauses. The difference is that some clauses must be preserved. More precisely, we can use these predicates that have an empty extension for the resolution. After that, the clauses that contain in the body the predicates with an empty extension are subsumed by the clauses obtained through resolution, therefore they can be removed. On the contrary, the clauses that contain these predicates in the head cannot be removed and must be preserved. In chapter 4 these clauses were removed after a reachability test. However, it is not possible to perform a reachability test during the preprocessing, and therefore these clauses must remain. For example, the user query may contain some of these predicates, in such a case these clauses would be used in the query rewriting.

The last difference is the possibility of cycles involving function symbols. After the cycles have been detected, the same algorithms are applied over FOL as they would be applied in the case of Datalog. In this case, we consider potentially cyclic recursion the same cases as in chapter 4 and additionally all those cases in which the head of some clause involved in the recursion contains some function symbol.

### 6.3.3 Deletion of extensionally subsumed clauses

We have seen that removing subsumed clauses has a strong impact on the efficiency of the query rewriting process and its results (section 5.2.2). With the EBox we can detect clauses that are subsumed at an extensional level and re-

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

move them. For this task, we will use the part of the EBox that only contains predicates with no intensional definition in the TBox. This means that we will use the clauses in the EBox that do not contain any predicates in the head of any clause in the TBox. Given some EBox  $E$ , we will refer to this set of clauses as  $ext(E)$ .

If we can derive some clause  $\gamma_r$  using the clauses in the  $ext(E)$  and some other clause  $\gamma$  in the TBox, then the clause  $\gamma_r$  is extensionally subsumed by  $\gamma$ . This extensional subsumption has the same practical implications as the regular subsumption, i.e. the values provided by this clause are contained in the values that its head has without it, either due to other clauses (as in regular subsumption) or due to the extensional values of the head predicate (in this case of extensional subsumption). Therefore, the extensionally subsumed clause can be removed from the Datalog program as any other subsumed clause. This operation is formalised in proposition 7, and applied in algorithm 6.5 at line 14.

The entailment is checked by applying resolution. The clauses in the purely extensional part of the EBox  $\mathcal{E}_p$  are used as side premises. The main premise for the resolution is either in the Datalog program or has been derived from a previous stage of the resolution. In practice this is equivalent to first obtaining the closure from the  $\mathcal{E}_p$  and then checking subsumption as especificied in proposition 7. The operations described here are more efficient than producing the closure of  $\mathcal{E}_p$  due to the optimisations explained in chapter 5.

Intuitively, we can see that for this kind of subsumption check we need to consider exclusively  $ext(E)$  and not the full EBox. The resolution derives clauses following containment relations that are only satisfied in the extensional level when using the EBox for resolution. By imposing an empty intension, the properties that hold in the extensional level do also hold in the union of extensional and intensional values, i.e. absolutely. In particular, this derivation will start from a clause and its head will never be unified, as it is an intensional predicate, not present in  $ext(E)$ . The replacements that happen in the body due to resolution will lead to more specific clauses, thus if we derive some other clause  $\gamma_r$  that is in the Datalog program, then this clause  $\gamma_r$  can be removed.

---

### 6.3.4 Extensional condensation of clauses

Similarly to removing the clauses that are subsumed, we can remove the atoms that are subsumed. This operation is similar to the clause condensation that we have seen in 2.1.1, but in this case we will use the information in the EBox to remove the atoms that are redundant. As in the previous optimisation, and for the same reasons, we will use the part of the EBox  $E$  that refers only to predicates that are exclusively extensional in the TBox, i.e.  $ext(E)$ .

In this case, we derive a clause  $\gamma_r$  from another clause  $\gamma$  and the extensional part of the EBox  $E$ , with the property that  $\gamma_r$  subsumes some other clause  $\gamma_s$  in the Datalog program, where  $\gamma_s$  may be  $\gamma$  or not. In such a case, the derived clause  $\gamma_r$  is extensionally implied, as in the previous case, which means that the values provided by this clause are redundant. Therefore, the clause  $\gamma_r$  can be added to the Datalog program. After the clause has been added,  $\gamma_s$  is subsumed by  $\gamma_r$ , therefore, we can remove  $\gamma_s$ .

This replacement of a clause  $\gamma_s$  with another clause  $\gamma_r$  that subsumes  $\gamma_s$  means replacing some clauses with more general ones, as for example those that have one less atom in the body. This replacement is analogous to clause condensation as described in section 2.1.1, using relations of subsumption between atoms derived from the EBox.

This operation is formalised in proposition 6, and applied in algorithm 6.5 at line 12. The operation is performed similarly to the previous, inside the same loop. The only difference is that in this case we are not checking subsumption but equivalence. To check equivalence, we check subsumption in both directions.

## 6.4 Proofs

In this section we include the formalisations and proofs relevant for the optimisations presented in the chapter.

**Proposition 5.** *Let  $\mathcal{J} = \langle \Sigma, \mathcal{A} \rangle$  be an OBDA system, let  $E$  be an EBox such that  $\mathcal{A}$  satisfies  $E$  and let  $E_p$  the part of the EBox  $E$  that contains only predicates with no intensional definition in  $\Sigma$ . For every pair of clauses  $\gamma, \gamma_r$  such that  $\gamma \in \Sigma$  and  $E_p \cup \gamma \models \gamma_r$  then  $\varphi_{\mathcal{J}}(\gamma_r) \subseteq \varphi_{\mathcal{J}}(\gamma)$ .*

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

*Proof.* To proof this we only need to consider how resolution works and the meaning of the EBox. We can consider the base case with one single resolution step, for  $n$  resolution steps the extension is immediate due to the transitive property of containment ( $\subseteq$ ). When we resolve  $\gamma$  with  $\gamma_e$ ,  $\gamma$  is the main premise, the predicates in  $E_p$  do not have an intensional definition in  $\Gamma$ , therefore it cannot be the head predicate in  $\gamma$ . Therefore,  $\gamma_r$  is created by replacing an atom in  $\gamma$  (the head of  $\gamma_e$ ) with the body of  $\gamma_e$ , plus some unification of variables. Due to the EBox we know that  $\forall \gamma_e \in E_p. v_\Gamma^e(\text{body}(\gamma_e)) \subseteq v_\Gamma^e(\text{body}(\gamma_e))$ . Therefore the values for the body of  $\gamma_r$  are a subset or equal to the values for the body of  $\gamma$  and  $\varphi_\Gamma(\gamma_r) \subseteq \varphi_\Gamma(\gamma)$ .  $\square$

**Proposition 6.** *Let  $\mathcal{J} = \langle \Sigma, \mathcal{A} \rangle$ ,  $E$  and  $E_p$  be defined as in proposition 5. Then, for every pair of clauses  $\gamma, \gamma_r$  such that  $\gamma \in \Sigma$  and  $E_p \cup \gamma \models \gamma_r$ , and for every query  $q$ ,  $\Phi_{\mathcal{J}}^q = \Phi_{\mathcal{J}'}^q$ , where  $\mathcal{J}' = \mathcal{J} \setminus \{\gamma_r\}$ .*

*Proof.* From proposition 5 we know that given the pair of clauses  $\gamma, \gamma_r$  such that  $\gamma \in \Sigma$  and  $E_p \cup \gamma \models \gamma_r$  then  $\varphi_{\mathcal{J}}(\gamma_r) \subseteq \varphi_{\mathcal{J}}(\gamma)$ .

For  $\mathcal{J}' = \langle \Sigma \setminus \{\gamma_r\}, \mathcal{A} \rangle$ , with  $p$  as the head predicate in  $\gamma_r$ , from proposition 3, if  $\varphi_{\mathcal{J}}(\gamma_r) \subseteq \varphi_{\mathcal{J}}(\gamma)$  (as previously proven) then  $\varphi_{\mathcal{J}}(\gamma_r) \subseteq v_{\mathcal{J}'}(p)$ .

Considering that  $\varphi_{\mathcal{J}}(\gamma_r) \subseteq v_{\mathcal{J}'}(p)$ , due to proposition 1, all the values for the predicates in the head of the clauses in  $\Sigma$  are the same for  $\mathcal{J}$  and  $\mathcal{J}'$ .

And finally, from corollary 1 it follows that for  $\gamma, \gamma_r$ , as previously described and due to the previous properties,  $\Phi_{\mathcal{J}}^q = \Phi_{\mathcal{J}'}^q$ .  $\square$

**Proposition 7.** *Let  $\mathcal{J} = \langle \Sigma, \mathcal{A} \rangle$ ,  $E$  and  $E_p$  be defined as in proposition 5. Let  $\gamma, \gamma_r$ , and  $\gamma_s$  be three clauses such that  $(\gamma, \gamma_s \in \Sigma) \wedge (E_p \cup \gamma \models \gamma_r) \wedge (\gamma_r \succeq_s \gamma_s)$ . And with  $\mathcal{J}' = \langle \{\gamma_r\} \cup \Sigma \setminus \{\gamma_s\}, \mathcal{A} \rangle$ . Then for every query  $q$ ,  $\Phi_{\mathcal{J}}^q = \Phi_{\mathcal{J}'}^q$ .*

*Proof.* Given the pair of clauses  $\gamma, \gamma_r$  such that  $\gamma \in \Sigma$  and  $E_p \cup \gamma \models \gamma_r$  due to proposition 5  $\varphi_{\mathcal{J}}(\gamma_r) \subseteq \varphi_{\mathcal{J}}(\gamma)$ . From proposition 3 we know that for  $\gamma, \gamma_r$  such that  $\varphi_{\mathcal{J}}(\gamma_r) \subseteq \varphi_{\mathcal{J}}(\gamma)$  then  $\Phi_{\mathcal{J}}^q = \Phi_{\langle \{\gamma_r\} \cup \Sigma, \mathcal{A} \rangle}^q$  for any possible query  $q$ , therefore  $\gamma_r$  can be added to  $\Sigma$  without altering the certain answers for  $\mathcal{J}$ . If  $\exists \gamma_s \in \mathcal{J}$  such that  $\gamma_r \succeq_s \gamma_s$ , as stated in this proposition, then from proposition 4 follows that  $\Phi_{\langle \{\gamma_r\} \cup \Sigma, \mathcal{A} \rangle}^q = \Phi_{\mathcal{J}'}^q$ , for  $\mathcal{J}' = \langle \{\gamma_r\} \cup \Sigma \setminus \{\gamma_s\}, \mathcal{A} \rangle$ . Summarising the proof:  $\Phi_{\mathcal{J}}^q = \Phi_{\langle \{\gamma_r\} \cup \Sigma, \mathcal{A} \rangle}^q = \Phi_{\mathcal{J}'}^q$  for any possible query  $q$ .  $\square$



---

Note that in this proposition,  $\gamma$  and  $\gamma_s$  may be the same clause without altering the result and due to the characteristics of the rewriting algorithm this will often be the case.

The following proposition is very similar to proposition 2, both refer to the deletion of clauses with non-mapped predicates. However, the context is different, proposition 2 refers to Datalog programs, which includes a specific query, and proposition 8 refers to OBDA systems, before any query is posed to the system. Intuitively, we can see that proposition 8 is more general, and it has different consequences.

**Proposition 8.** *Let  $\mathcal{J} = \langle \Sigma, \mathcal{M}, D \rangle$  and  $\mathcal{J}' = \langle \Sigma', \mathcal{M}, D \rangle$  two OBDA systems, let  $p$  be a predicate in  $\Sigma$  and  $\Sigma'$ , and let  $\Gamma$  be a set of clauses, such that:*

1.  $\Gamma \subseteq \Sigma$ ,
2.  $\Sigma' = \Sigma \setminus \Gamma$ ,
3. *for every clause  $\gamma$  in  $\Sigma$  such that  $p$  appears in its body,  $\gamma \in \Gamma$ ,*
4.  *$p$  does not appear in any mapping assertion in  $\mathcal{M}$ , and*
5.  *$\Sigma$  is closed with respect to  $p$ , i.e. no resolution can be performed in  $\Sigma$  by unifying atoms with  $p$ .*

*In this situation, the values for every predicate  $p_i$  in  $\mathcal{J}'$  are equal to the values for that predicate  $p_i$  in  $\mathcal{J}$ , i.e. the set of clauses  $\Gamma$  can be removed from  $\Sigma$  without altering the values for the predicates in it, nor the certain answers for any possible query  $q$  on these OBDA systems.*

*Proof.* The proof is analogous to the proof of proposition 2. □

The operation in proposition 8 is performed during the preprocessing. This implies a difference with proposition 2, in this case a reachability test cannot be performed after the removing the specified set of clauses, as there is no query to consider the reachability. Additionally to that, during the preprocessing the TBox  $\Sigma$  may contain function symbols, i.e. it may not be Datalog but FOL. This expressiveness simply causes a richer casuistry for cycles, which

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

may prevent a finite closure of the resolution with respect to some predicates and therefore their possible removal.

**Proposition 9.** *Let  $\mathcal{J} = \langle \Sigma, \mathcal{A} \rangle$  be an OBDA system, let  $E$  be an EBox satisfied by  $\mathcal{A}$  and  $cgraph(\Sigma) = \langle \mathcal{V}(\Sigma), \mathcal{W}(\Sigma) \rangle$ . Let  $\Gamma$  be a set of clauses in  $\Sigma$  such that  $cgraph(\Gamma) = \langle \mathcal{V}(\Gamma), \mathcal{W}(\Gamma) \rangle$  is a SCC in  $cgraph(\Sigma)$  that receives no connections from other SCCs, i.e.  $\forall (\gamma_a, \gamma_b) \in \mathcal{W}(\Sigma). \gamma_b \in \mathcal{V}(\Gamma) \rightarrow \gamma_a \in \mathcal{V}(\Gamma)$ , and all clauses in this SCC are entailed by the EBox, i.e.  $\forall \gamma \in \mathcal{V}(\Gamma). E \models \gamma$ . Let  $\mathcal{J}' = \langle \Sigma', \mathcal{A} \rangle$  be an OBDA system where  $\Sigma' = \Sigma \setminus \Gamma$ . Then, for every query  $q$ , it obtains the same certain answers from  $\mathcal{J}$  and  $\mathcal{J}'$ , i.e.  $\Phi_{\mathcal{J}}^q = \Phi_{\mathcal{J}'}^q$ .*

*Proof.* To prove this proposition we will prove that all the predicates in  $\Sigma$  have the same values in  $\mathcal{J}'$  have the same values, i.e. for every predicate  $p$  in any clause of  $\Sigma$ ,  $v_{\mathcal{J}}(p) = v_{\mathcal{J}'}(p)$ . This suffices to prove that the certain answers are also the same, as in corollary 1. The values for each predicate are extensional or intensional.

By definition 2, the extensional values of a predicate depend exclusively on the ABox. Both  $\mathcal{J}$  and  $\mathcal{J}'$  share the same ABox,  $\mathcal{A}$ . Therefore, the extensional values for every predicate  $p$  in  $\mathcal{J}$  are the same for both systems, i.e. for every predicate  $p$ ,  $v_{\mathcal{J}}^e(p) = v_{\mathcal{J}'}^e(p)$ .

The intensional values for a predicate in an OBDA system depend on the clauses that have that predicate in the head and their contributions, again by definition 2. All the clauses are equal in both systems, except for the clauses in  $\Gamma$ , which is a SCC that does not receive any connections. This means that the intensional values of the clauses in  $\Gamma$  depend exclusively on other clauses that are also in  $\Gamma$ . Therefore, if the predicates in the head of clauses in  $\Gamma$  have the same intensional values for  $\mathcal{J}$  and  $\mathcal{J}'$  then all the predicates in  $\Sigma$  have the same intensional values for  $\mathcal{J}$  and  $\mathcal{J}'$  as well.

Finally, for the intensional values of the predicates in  $\Gamma$ , they all have the same property: For every clause  $\gamma$  in  $\Gamma$ , with  $p = pred(head(\gamma))$ , the contributions of  $\gamma$  are contained in the extensional values of the predicate in its head, i.e.  $\varphi_{\mathcal{J}}(\gamma) \subseteq v_{\mathcal{J}}^e(p)$ . We have already proved that the extensional values for these predicates are the same in  $\mathcal{J}$  and  $\mathcal{J}'$ , therefore by this property, the presence of these clauses would have no impact on the values for the predicates in  $\Sigma$ ,

---

analogously to proposition 1. The remainder of the proof consists on proving this property.

A simple way to prove this property is *reductio ad absurdum*. If there is some value in the contributions of  $\gamma$  that is not an extensional value for  $p$  then it must be due to some values for the predicates in the body. The EBox  $E$  entails  $\gamma$ , therefore, these values have again to be intensional for the predicates in the body. The provenance of these intensional values should be traced until some clause that is not entailed by the EBox  $E$ . However, this is not possible, because the SCC  $\Gamma$  does not receive any connections and all clauses in  $\Gamma$  are entailed by  $E$ .

Therefore, we must conclude that all the predicates in clauses of  $\Gamma$  have the same values (both extensional and intensional) in  $\mathcal{J}$  and  $\mathcal{J}'$ . The same holds for all the other predicates in the clauses of  $\Sigma$ . Therefore the certain answers for  $\mathcal{J}$  and  $\mathcal{J}'$  are the same for any query.

As a summary of the proof:

1. The intensional values for the predicates in the clauses in  $\Gamma$  are the same for  $\mathcal{J}$  and  $\mathcal{J}'$ .
2. The intensional values for the predicates in the clauses in  $\Gamma$  are the same for  $\mathcal{J}$  and  $\mathcal{J}'$ .
3. Therefore all the intensional values for the predicates in  $\Sigma$  are the same for  $\mathcal{J}$  and  $\mathcal{J}'$ .
4. The ABox in  $\mathcal{J}$  and  $\mathcal{J}'$  is the same.
5. Therefore the extensional values for the predicates in  $\Sigma$  are the same for  $\mathcal{J}$  and  $\mathcal{J}'$ .
6. The certain answers for any query  $q$  posed to a OBDA system  $\mathcal{J}$  depend directly on the values for the predicates in the body of  $q$  according to  $\mathcal{J}$ , i.e.  $v_{\mathcal{J}}(p)$ .
7. Both intensional and extensional values for the predicates in clauses of  $\Sigma$  are the same for  $\mathcal{J}$  and  $\mathcal{J}'$ .

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

8. Therefore the certain answers for any query  $q$  are the same for  $\mathcal{J}$  and  $\mathcal{J}'$ , i.e.  
 $\forall q. \Phi_{\mathcal{J}}^q = \Phi_{\mathcal{J}'}^q$ .

□

After the removal of one SCC there may be other SCCs that satisfy the conditions in the proposition. The removal of such SCCs can be repeated iteratively until the SCCs that do not receive any connections are not entailed by the EBox.

The removal of SCCs for  $dlgraph(\mathcal{T}_{\mathcal{O}})$  is analogous to the previously described removal of SCCs for  $cgraph(\Sigma)$ .

**Proposition 10.** *Let  $\mathcal{O} = \langle \mathcal{T}_{\mathcal{O}}, \mathcal{A}_{\mathcal{O}} \rangle$  be an  $\mathcal{ELHIJ}$  ontology and let  $\mathcal{E}_{\mathcal{O}}$  be an  $\mathcal{ELHIJ}$  EBox such that  $\mathcal{A}_{\mathcal{O}}$  satisfies  $\mathcal{E}_{\mathcal{O}}$ . Let  $\mathcal{T}_{\mathcal{O}}'$  be the TBox obtained by recursively removing the axioms in  $\mathcal{T}_{\mathcal{O}}$  that belong to SCCs in  $dlgraph(\mathcal{T}_{\mathcal{O}})$  that receive no connections from any other SCC and whose axioms are all entailed by the EBox  $\mathcal{E}_{\mathcal{O}}$ . The ontology  $\mathcal{O}' = \langle \mathcal{T}_{\mathcal{O}}', \mathcal{A}_{\mathcal{O}} \rangle$  and the ontology  $\mathcal{O}$  entail the same set of ground atoms.*

*Proof.* The conversion of DL axioms to Horn clauses for  $\mathcal{ELHIJ}$  preserves the satisfiability of the OBDA system. Therefore the removal of SCCs in  $dlgraph(\mathcal{T})$  is equivalent to performing the operation on  $\Sigma = toHorn(\mathcal{T}_{\mathcal{O}})$  and obtaining a new  $\Sigma' = toHorn(\mathcal{T}_{\mathcal{O}}')$ .  $\langle \Sigma, \mathcal{A} \rangle$  and  $\langle \Sigma', \mathcal{A} \rangle$  are equisatisfiable due to proposition 9. Due to the conversion to Horn clauses  $\langle \Sigma, \mathcal{A} \rangle$  is equisatisfiable to  $\langle \mathcal{T}_{\mathcal{O}}, \mathcal{A}_{\mathcal{O}} \rangle$ . Similarly for  $\langle \Sigma', \mathcal{A} \rangle$  and  $\langle \mathcal{T}_{\mathcal{O}}', \mathcal{A}_{\mathcal{O}} \rangle$ . Finally, for all of the previous, we have equisatisfiability for  $\mathcal{J}$  and  $\mathcal{J}'$ . □

## 6.5 Conclusions

In this chapter we have seen that knowledge about the completeness of the mappings can be used at different stages of the query rewriting process for optimisation purposes, and in different ways.

The characteristics of this knowledge are similar to those seen in chapter 4. However, this knowledge is captured as  $\mathcal{ELHIJ}$  axioms, which together form an EBox. Therefore, the optimisations to use this knowledge during query rewriting resemble more closely those in chapter 5.

---

Similarly to previous chapters, the impact of these optimisations will depend on the characteristics of the queries and the ontologies. Additionally in this case, the impact of the optimisations will depend on the characteristics of the EBox. In particular, using the knowledge in the EBox involves additional resolution saturation processes, which are costly operations. This means that in some cases the query rewriting time may increase. However, the size of the rewritten queries should decrease in the case of UCQs, i.e. using an EBox should lead to more simple rewritten queries. Normally, the time required for query rewriting is a small fraction of the total time in query answering. Therefore, the optimisation of the query rewriting results may produce a greater improvement in the times for query answering than the potential increase of the query rewriting time.

Additionally, further optimisations may be feasible when addressing some given logic in particular, for example with more specific resolution processes. The techniques presented in this chapter are based on the operations of resolution and subsumption check, which are general and well known. Overall, the techniques presented in this chapter should be generalisable to other logics and other algorithms for both query rewriting and query answering.

As in previous chapters, we focus on an improvement of the query rewriting process and its results (especially the latter in this case) quantitatively considering the average situation, i.e. the optimisations may not be effective in some particular cases. This quantitative evaluation is detailed in the next chapter. In any case, the optimisations presented are proven to be correct with respect to query answering. In this regard, soundness means that the rewritten queries produce no additional answers when the optimisations are applied, while completeness means that the rewritten queries produced with the optimisations retrieve all the certain answers of the original query.

## 6. OPTIMISATIONS IN THE PRESENCE OF AN EBOX

---

# Chapter 7

## Evaluation

In this chapter we analyse the dimensions and assets used in the evaluation of the systems in the state of the art. Then we evaluate the hypotheses presented in section 3.2.

### 7.1 Dimensions in query rewriting evaluation

We will first analyse the main differences among different query rewriting approaches, so as to be able to obtain the main dimensions and challenges that should be considered when comparing among them:

#### 7.1.1 Determine the ontology language expressiveness

We have already seen that query rewriting in an OBDA context uses at least a TBox  $\mathcal{T}$  to transform a query  $q$  into a rewritten query  $q_{\mathcal{T}}$ . This operation normally uses Horn clauses to model both the TBox as a Datalog TBox  $\Sigma$  and the query  $q$  as a conjunctive clause or a UCQ of clauses, with the usual rewritten query  $q_{\Sigma}$ . The main characteristics of this process are the computational cost of the rewriting and the complexity of the rewritten query, both of which depend on the expressiveness of the ontology. We have also briefly described the main logics used in the state of the art in section 2.3. Each logic has a different expressiveness, which determines the coverage of the ontology that can be performed when converting it from a description logic language into a set of clauses in Datalog, Datalog $\pm$  or FOL.

## 7. EVALUATION

---

The conversion to FOL happens in Rapid, REQUIEM and kyrie. Given the languages that they give support to, these systems consider expressions of the form  $\exists A_1 \sqsubseteq R.A_2$ . These expressions generate Skolem functions, what implies the production of FOL clauses that are later processed and converted to Datalog. Nyaya considers as well the existential quantifier in Datalog $\pm$ .

### 7.1.2 Characterise the impact of reduced expressiveness in each approach

As discussed above, one of the main differences among systems is the expressiveness of the TBox  $\mathcal{T}$  (or ontology  $\mathcal{O}$ ) that can be handled by the approach in the conversion into a set of clauses<sup>1</sup>  $\Sigma$ . In fact, it normally happens that some axioms of the original ontology may be even lost or discarded in this transformation process. This can lead to unnecessarily complex queries, incomplete results and even incorrect results:

- Some systems may not be able to handle the additional expressiveness, what means the loss of completeness with respect to that expressiveness and the loss of some answers.
- Non-recursive Datalog is a precondition for some systems. These systems may enter infinite loops or produce incomplete answers when the axioms in some ontology lead to the production of recursive Datalog. For example, for certain ontologies, the rewriting process in Nyaya times out before producing some result for all the queries tested. Other systems like Presto target non-recursive SQL rewritings, and therefore some answers may be lost in the process. Finally, the remainder of the systems choose different strategies to preserve the recursive predicates, and in such cases the completeness of the answers depends on the capabilities of the underlying systems to process recursive queries.
- The subsumption of some clauses may be implied by some axioms out of the handled expressiveness. Thus clauses that could have been eliminated

---

<sup>1</sup>Note that other forms of knowledge representation could potentially be used but the state of the art so far focuses on Horn fragments in DL.



---

upon checking that subsumption will be preserved in the rewritten query. This does not impact the correctness of the results, but it may imply longer and more redundant rewritings that may increase the load in underlying modules in the OBDA system.

- Negations (or disjunctions) are not used for query rewriting by any of the systems analysed. This means that the rewritten query may contain clauses that are contradictory and will not obtain answers (they could be eliminated). If the data source is not consistent with these negations then incorrect answers may be obtained, according to the semantics of the TBox.

The impact of the lost expressiveness has not been considered in previous evaluations. And to the best of our knowledge, there is no evaluation about which is the usual expressiveness in the ontologies that can be found and reused to enable OBDA.

### 7.1.3 Determine the complexity of rewritten queries

Evaluation of query rewriting systems has mainly focused so far on the most comparable cases: conjunctive queries and ontologies whose expressiveness is at the intersection of the expressiveness of all the systems to be compared. Most systems allow choosing whether the output of the query rewriting process should be a Datalog program or a UCQ. If Datalog is produced then this Datalog is always non-recursive for FOL-reducible logics. If the expressiveness is in the aforementioned intersection of all systems, then rewritten queries are ensured to be complete and correct with respect to this expressiveness. In this case the only difference that can be found among different UCQs, for the same ontology and query, is in terms of subsumed clauses or atoms that are not removed from the final result. Datalog rewritings may vary more due to different ways in which subqueries can be arranged. This heterogeneity makes the evaluation process significantly complex.

Furthermore, there is neither standard nor a set of tools to convert the Datalog or the UCQs obtained with these systems to actual queries to perform on a data source. As we have seen in section 2.4 some systems are integrated in OBDA

## 7. EVALUATION

---

systems and some are not, but the modular design and implementation does not come with a modular evaluation that can compare them properly. A workaround for this limitation has been proposed in the evaluation of Nyaya by measuring more carefully the UCQs generated (Datalog is not considered in this evaluation). More precisely, they propose considering the number of clauses, the number of atoms and the number of joins. However, it is not specified whether the number of atoms is the total number of atoms or the number of distinct atoms. Atom repetition may have a big impact on the execution of the query depending on whether 1) the query translation uses some kind of temporary tables to store intermediate results or 2) repeated atoms are translated to as many queries as times they are related. However, the differences among systems are small in the resulting UCQs due to the lack of freedom in this structure. UCQs are flat and (unlike Datalog) the results are basically equal in all systems, unless some subsumed atoms or clauses are kept in the UCQ or there is some loss of completeness or correctness. This should not be the case for any of these systems if the ontologies used do not exceed the expressiveness that they can handle.

### 7.1.4 Determine the impact of the complexity of original queries

The behaviour of query rewriting systems may also vary greatly depending on the original queries that are posed to the systems. Among the main characteristics to be considered for queries we can cite:

- length of the queries, i.e. predicates in the body of the query.
- types of variables and number of variables of each type. Variables may be present in queries in different roles:
  - distinguished variables (those in the head of the query).
  - existential variables (non distinguished variables that appear only in one predicate).
  - join variables (non distinguished variables that appear in at least two predicates).

- 
- length of the property paths in the queries.
  - hanging or closed property paths. Hanging property paths leads to existential variables, closed property paths lead to distinguished variables.
  - separability of some parts of the query as independent subqueries.

A set of queries has traditionally been used for evaluations in the state of the art. However, this set of queries has not been adequately characterised in terms of their realism or the coverage of the casuistry.

### 7.1.5 Usage of additional information for the query rewriting process

Besides ontologies and queries, some systems add the possibility to use additional information for the query rewriting process (e.g., the use of an EBox in Prexto and kyrie3). Such additional information may provide potentially further optimisations of the process and results. However, as such additional information may be very ad-hoc for each system, it is difficult to compare results among systems that use and do not use it. Furthermore, in the case of EBoxes, for instance, their application in realistic scenarios is still largely unknown, what makes it harder to evaluate the impact of EBoxes in the rewriting (process and results).

The factors to consider for this matter are the *characteristics of the rewritten queries* and how they relate with the *characteristics of the system* that should process them and the actual data that is stored and queried. There is neither a standard nor a set of tools to convert the Datalog or the UCQs obtained with these systems to actual queries to perform on a data source.

## 7.2 Assets used for evaluations

Despite the limitations mentioned in the previous section, the need for a proper evaluation has been shaping a common framework for evaluation. This framework provides a set of ontologies and queries, but lacks other elements as an ABox to consider how different optimisations or differences in the queries could translate

## 7. EVALUATION

---

to answers and results. Moreover, to the best of our knowledge there are no estimations about the representativeness of the set of ontologies and queries with respect to the characteristics of actual OBDA systems as implemented in practice.

Despite the difficulties and challenges presented above, several approaches have been formulated for the comparison of query rewriting approaches. However, there is a large heterogeneity in these approaches, what claims for the need to come up with a common evaluation framework to allow better comparisons and to provide sufficient information as well for system developers to improve their systems, as in any general benchmarking process.

Some of the seminal work can be attributed to the perfect reformulation proposal from Calvanese [Calvanese et al., 2007a], which is evaluated only in theoretical terms with respect to complexity, completeness and correctness.

Pérez-Urbina compared his approach with perfect reformulation by using a set of ontologies that have become common across evaluations in this area:

- Adolena (A). An ontology developed to allow OBDA for the South African National Accessibility Portal [Keet et al., 2008]. This is the largest ontology in this set of ontologies.
- path1 (P1) and path5 (P5). Two synthetic ontologies to help understand (and show) the “impact of the reduction step” in REQUIEM. Despite their apparent simplicity, rewriting times for Datalog in these ontologies are significant.
- StockExchange (S). A ontology that captures information about European Union financial institutions. This ontology has been later used [Rodríguez-Muro et al., 2008] as a driving example to explain OBDA and how users may benefit from it.
- University (U). A DL-Lite<sub>R</sub> version of LUBM [Guo et al., 2005]. LUBM focuses on the ABox more than the TBox. On the one hand this allows systems like Clipper to use the ABox for further evaluation of rewritten queries. On the other hand it has a rather flat TBox [Rodríguez-Muro and Calvanese, 2012], what means that the rewritings are simple. Simple

---

rewritings are produced in short times and the rewritten queries are also short, as we will see in the next section.

- Vicodi (V). An ontology about European history developed in the EU-funded VICODI project [Nagypal, 2005].

This set of ontologies was expanded with AX, P5X, and UX, where some of the previous ontologies included auxiliary predicates. These auxiliary predicates replace the existential predicates by applying the encoding required by the previous approach [Calvanese et al., 2007a]. This encoding significantly increases the size of the ontologies and the time required for the rewriting, as the data in the evaluation reflects.

All previous ontologies are accompanied by a set of five queries for each of them.

For the evaluation of Presto these sets of queries are incremented up to seven queries for each of the ontologies. The ontologies are also incremented with the ontologies from Kontchakov [Kontchakov et al., 2009], more examples from the LUBM benchmark (besides of U and UX) and a newly created ontology. The ontologies from Kontchakov are:

- Galen-Lite. The *DL-Lite<sub>core</sub>* approximation of the well-known medical ontology Galen [Rogers and Rector, 1996]. The interest in this ontology is mainly in its taxonomy, where few axioms involve roles.
- Core. A *DL-Lite<sub>core</sub>* representation of (a fragment of) a supply-chain management system used by the bookstore chain Ottakar's, now rebranded as Waterstone's. Contrary to Galen, the taxonomy in Core is small but it contains a rich set of axioms about roles.

Later approaches like Rapid, Nyaya, Clipper and kyrie use the aforementioned ontologies and queries, and keep on using them for evaluation purposes. Prexto is only compared with Presto by means of a small unnamed ontology to show the impact that the optimisations in Prexto may have, which is done more as an example than as an empirical evaluation. In the case of kyrie some newly

## 7. EVALUATION

---

created ontologies named AXE, AXEb, P5XE and UXE are used. These ontologies expand AX, P5X and UXE by including additional axioms that fall in the expressiveness of  $\mathcal{ELHJO}$ , which is not covered in less expressive systems. Again the purpose of this evaluation is showing how these axioms may impact results, regardless of realism or empirical significance, specially by comparing the performance with REQUIEM which can handle the  $\mathcal{ELHJO}$  expressiveness.

Upon a closer analysis of the ontologies and the queries we can see that they are fairly heterogeneous. For example rewritings may vary greatly in time and size, depending on the characteristics of the ontology and the query as we have mentioned in the previous section. Finally, there is not a well-founded analysis of whether these ontologies are representative enough to cover all the challenges and characteristics that we have discussed in the previous section.

### 7.3 Evaluation for the optimisations in the presence of mappings

For the evaluation of the system two geographical ontologies have been used, both developed and used in the context of independent projects. The first ontology is *hydrOntology* [Vilches-Blázquez et al., 2007], with 155 concepts and expressiveness  $\mathcal{SHJN}(\mathcal{D})$ . *HydrOntology* has been used in several projects and mapped with several geographical databases of the Instituto Geográfico Nacional (IGN)<sup>1</sup>, generating several RDB2RDF mapping files, which, being available, have been used for the tests. More precisely, the databases mapped here are three:

- The National Atlas data source, with 32 mappings that provide 1,100 hydrographical instances for an overview of Spain’s human and physical environment.
- The Numerical Cartographic Database (BCN200) with 57 mappings that provide 60,000 toponyms related to hydrographical instances.
- EuroGlobalMap (EGM), produced in cooperation with the National Mapping Agencies of Europe, with 32 mappings that provide 3,500 Spanish

---

<sup>1</sup><http://www.ign.es>

hydrographical toponyms.

The second ontology is *PhenomenOntology* [Gómez-Pérez et al., 2008]. In this case among the phenomena that could be covered only the module regarding transportation networks has been used, which provides 66 concepts, having a common ancestor in “Red” (“Network”) which is the concept used for the test queries whose results are shown in table 7.1. In the case of *PhenomenOntology*, only one file with 18 mappings is used.

Both of these two ontologies are expressed in OWL, without imposing any kind of restriction, thus some of the axioms have to be discarded in the beginning of the process, as described in section 4.3.1, as they do not fall into the  $\mathcal{ELHJ}$  expressiveness. The mapping files are R2O mappings [Barrasa et al., 2004].

Ontologies		HydrOntology								Phenomenontology		
Information		686 clauses, 405 ignored statements								66 c., 114 i.s.		
Mappings		None	BCN200		Atlas		EGM		None	Unique		
Mappings #		0	57		32		32		0	18		
Prune method		A	H	L	H	L	H	L	A	H	L	
Mode	Phase	Number of clauses generated after each phase										
N	Prune	535	323	445	287	415	336	429	66	66	66	
	Saturation	412	25	50	17	24	19	31	66	60	60	
	Unfolding	2333	25	46	17	22	19	28	64	14	14	
	Prune	2333	25	46	17	22	19	28	64	14	14	
G	Prune	535	323	445	287	415	336	429	66	66	66	
	Saturation	407	25	49	15	24	19	30	66	60	60	
	Unfolding	894	25	45	15	22	19	27	64	14	14	
	Prune	688	25	45	15	22	19	27	64	14	14	
F	Prune	535	323	445	287	415	336	429	66	66	66	
	Saturation	407	25	49	15	24	19	30	66	60	60	
	Unfolding	2245	25	45	15	22	19	27	64	14	14	
	Prune	911	25	45	15	22	19	27	64	14	14	
N	Total time (ms)	2735	172	625	156	359	250	625	17	15	15	
G	Total time (ms)	2250	172	512	156	360	218	532	31	16	16	
F	Total time (ms)	3719	172	531	156	344	235	531	31	16	16	

Table 7.1: Results of the evaluation for the kyrie algorithm

## 7. EVALUATION

---

The results<sup>1</sup> are summarised in table 7.1. They vary depending on the method used. The first letter is for the methods in the original REQUIEM, ‘N’ for “naive”, ‘F’ for “full forwarding” and ‘G’ for “greedy”, the second letter stands for the modification applied as described in this paper, in this case ‘A’ stands for “all”, since all predicates are kept independently of the mappings. ‘H’ will stop the prune on the first predicate that is mapped (the highest), as all the values that are correct for that predicate are assumed to be retrievable from the RDB2RDF mappings, as described in section 4.3.1. Finally ‘L’ will stop the prune at the “lowest” mapped predicate. This last approach will keep all the mapped predicates in the ontology after pruning it, since they could be complementary to some extent, again as described in section 4.3.1.

The query posed to the system for comparison purposes is a general query covering a good part of the ontology. In the case of *hydrOntology* the query is simply  $Q(x) \leftarrow \text{Aguas}(x)$ , i.e. “Water”, which covers as a superclass most of the taxonomy present in *hydrOntology*. Similarly, in the case of *PhenomenOntology* the query used is  $Q(x) \leftarrow \text{Red}(x)$ . Being a module about transportation networks, the concept “Red” (“Network”) is the most general one. Both queries cover most of the mappings and ontologies used for the tests, providing a good approximation of what could be expected in a different context.

As we can see in the table, the reduction in the number of classes (and hence in the number of queries submitted) is very important, especially in the case of Atlas, which has the same number of mappings than EGM but these allow less combinations in the rewritings. It is also noteworthy that the number of clauses is unaltered after the last prune phase, which has less clauses to prune due to the prunes performed previously.

### 7.4 Evaluation for the engineering optimisations

We have performed an empirical evaluation using the ontologies and queries normally used for this purpose in the state of the art and we have expanded this set with four additional ontologies to show the potential impact of the axioms that are not covered with less expressive logics and are covered in our approach. In

---

<sup>1</sup>Available at [purl.org/net/jmora/kyrie/mappings/analysis](http://purl.org/net/jmora/kyrie/mappings/analysis)



---

the most critical example we obtained a reduction of roughly 90% of the number of clauses generated for the union of conjunctive queries (UCQ) with the addition of one single axiom. We have also added mapping files specifying the mappings for these ontologies providing different degrees of coverage to measure the impact of the mapping coverage on the process and results of the rewriting. The full results for this evaluation are available online<sup>1</sup>.

The evaluation has been performed by running the main systems in the state of the art. REQUIEM, Presto, Rapid, Nyaya, Clipper and our system, kyrie. The evaluation has been performed on cold run, by restarting the application after every query. Each query has been run a minimum of five times per system and the results averaged. The hardware used is a Intel®Core™2 6300 @1.86GHz with 2GB of RAM, Windows® XP and Java™ version 1.6.0\_33. We have measured the times and number of clauses generated for the usual ontologies, which were proposed in [Pérez-Urbina et al., 2009] along with sets of five queries for each of the ontologies provided.

These results can be seen in tables 7.2 and 7.3, for Datalog rewritings and UCQ rewritings respectively. Presto and Clipper do not perform rewritings to UCQ, at the same time Nyaya and Prexto only rewrite to UCQs, therefore each pair is present only in its respective table. It must be noted that times for these systems have been measured in the same way they were measured for the evaluations in their papers (with the code included in the systems). This means that the times have been obtained by using the Java method `System.currentTimeMillis()`. It can be seen from the results in previous papers that this method updates the time in milliseconds in intervals of more than 15ms and less than 16ms, for the current infrastructure. In this case we compensate for the lack of precision in the measure making more measurements and then the average, for instance if a stage takes an average of 7.75ms then the likelihood of obtaining a measurement of 0ms and a measurement of 15 or 16 ms are about the same, and the average will tend to 7.75 as the number of measurements increases.

We have also extended some of the ontologies, the most complex ones and interesting in this evaluation, with additional axioms, as aforementioned. This allows to check the impact that a little difference in expressiveness and only a

---

<sup>1</sup><http://purl.org/net/jmora/kyrie/optimisations/evaluation>

## 7. EVALUATION

---

$\mathcal{O}$	$q$	REQUIEM(F)		Presto		Rapid		Clipper		kyrie	
		size	time	size	time	size	time	size	time	size	time
U	1	19	12	4	7	4	3	2	21	2	0
	2	47	16	2	9	2	9	49	19	47	3
	3	20	9	8	16	8	12	21	24	20	3
	4	64	15	3	12	3	3	63	18	64	3
	5	53	12	8	15	8	12	53	15	53	0
	6	20	12	19	6	21	15	16	18	16	9
	7	49	25	22	15	22	18	44	18	45	12
	8	10	9	13	6	13	9	10	20	10	3
	9	29	15	24	12	24	17	19	20	21	7
UX	1	22	6	7	10	7	3	5	27	5	6
	2	52	15	2	15	2	15	54	27	52	3
	3	24	15	10	28	10	9	25	28	24	3
	4	70	15	6	19	6	12	69	22	70	0
	5	56	15	11	15	11	15	56	21	56	12
	6	24	18	28	15	27	22	20	19	20	3
	7	55	28	29	21	27	21	50	28	51	12
	8	11	6	14	12	14	13	11	26	11	1
	9	32	15	30	21	30	15	22	46	24	4

Table 7.2: Results of the execution to obtain datalog (time in ms)

$\mathcal{O}$	$q$	REQUIEM(F)		Rapid		Prexto		Nyaya		kyrie	
		size	time	size	time	size	time	size	time	size	time
U	1	2	15	2	3	2	9	2	5	2	0
	2	1	103	1	15	1	15	1	1	1	34
	3	4	212	4	9	4	18	4	34	4	18
	4	2	3762	2	12	2	15	2	4	2	50
	5	10	13034	10	18	10	15	10	33	10	37
	6	29	47	29	28	28	12	40	1595	29	28
	7	42	797	42	37	70	18	54	670	42	43
	8	10	15	10	18	10	6	10	63	10	3
	9	960	1893	960	209	960	928	960	75135	960	1107
UX	1	5	15	5	12	5	12	5	22	5	9
	2	1	172	1	12	1	16	1	3	1	37
	3	12	2062	12	15	12	28	12	55	12	21
	4	5	31422	5	15	5	23	5	6	5	47
	5	25	91878	25	27	25	23	25	39	25	46
	6	323	468	323	106	448	178	348	2685	323	187
	7	1456	37212	224	81	280	75	264	852	224	121
	8	20	21	20	23	20	15	20	61	20	9
	9	4200	30506	4200	739	4200	21181	4200	366673	4200	16875

Table 7.3: Results of the execution to obtain UCQ (time in ms)

---

few axioms can have on the results of query rewriting.

For example we have expanded AX into AXE by adding a single axiom.  $\exists AUX0^-. \exists AUX1 \sqsubseteq Quadriplegia$ , even though  $AUX0$  and  $AUX1$  are not very descriptive names, this is an axiom that could fit in the semantics of the ontology, considering  $AUX0$  is a subproperty of *isAffectedBy* and  $AUX1$  is a subproperty of *isAssistedBy*. The impact of this single axiom is specially noticeable in the 5th query where we can see the rewriting time and clauses generated in the UCQ decrease significantly. The results for Rapid remain the same since this axiom falls out of the expressiveness that Rapid can handle. Similar axioms have been added to the rest of ontologies that have been expanded into their 'E' version.

Finally, we include the results for the preprocessing stage, which is not shared by any of the systems in the state of the art. As can be seen in table 7.4, the time needed for preprocessing is fairly short considering that this stage is performed only once every time the ontology changes. The size of the preprocessed ontologies does also remain at reasonable sizes, being P5XE the worst case with a factor of  $5.8\overline{51}$  and three cases with a factor of 1.

O	Number of clauses		time
	before	after	
A	128	205	65
AX	154	283	103
AXE	156	312	112
AXEb	156	312	118
P1	1	1	9
P5	9	9	3
P5X	13	21	15
P5XE	27	158	84
S	52	69	31
U	82	96	27
UX	87	111	31
UXE	88	112	34
V	222	222	50

Table 7.4: Results of ontology preprocessing in kyrie2, number of clauses and milliseconds.

The evaluation provided interesting results. Times are in many cases comparable with Rapid, which is the fastest algorithm. Times are even shorter than

## 7. EVALUATION

---

Rapid in some cases for the generation of the Datalog rewriting. The unfolding takes longer times than Rapid in many cases, which is natural considering that Rapid handles a lesser expressiveness. More precisely, the Datalog that Rapid uses has always one atom in the body and this allows grouping several inference steps into one. The results in the number of clauses generated for the UCQ are the same for the ontologies that fall in the intersection of the different expressivenesses that the different systems handle and the main difference relies on the time that is needed to generate these rewritings.

However, with the more expressive ontologies (names including a capital “E”) we can see the difference in the size of the rewritings. When comparing REQUIEM and kyrie we can see they generate a different number of clauses in the UCQ when the Datalog generated is recursive and the unfolding can only be partial. In this case kyrie aims at reducing the number of different predicates that can be found in the head of some clause, which can be considered as “unfolding as much as possible”, REQUIEM chooses to unfold partially the Datalog available. The Datalog generated in this way is linear Datalog, where linear means that each clause contains at most one intensional database predicate. When comparing Rapid and kyrie we can see that kyrie generates a different number of clauses in more cases than when compared with REQUIEM, this is due to the axioms included in the extended ontologies, which are ignored by Rapid since they fall out of the expressiveness that it can handle. The difference in query five with ontology “AXE” is especially remarkable. In this case kyrie generates a UCQ that is about 10% the size of the Rapid UCQ by including one single axiom, as explained before.

The optimisations done are especially evident in the time that the generation of the UCQ takes when compared to REQUIEM. However, they are more significant when comparing the Datalog results. REQUIEM performs an optimisation stage that removes subsumed clauses after the UCQ has been generated, but in the case of kyrie this is done along with resolution (section 5.2.2), therefore subsumed clauses have already been removed from this Datalog program. The time needed to generate the Datalog rewriting for some queries is on par with Rapid, being lesser or greater depending on the case. The same can be said about the size of this Datalog rewriting.

---

## 7.5 Evaluation for the optimisations in the presence of an EBox

We have performed an empirical evaluation to check our query rewriting optimisations. There is no benchmark in the state of the art to test query rewriting with EBoxes, therefore we have decided to use some of the most widely used ontologies for the evaluation of query rewriting systems [Mora and Corcho, 2013a], in particular we used: Several real world ontologies used in independent projects like Adolena (A), Vicodi (V) and StockExchange (S). Benchmark ontologies, like a DL-Lite<sub>R</sub> version of LUBM (U). Artificial ontologies to test the impact of property paths, path1 and path5 (P1, P5). Previous ontologies with auxiliary predicates for DL-Lite compliance [Calvanese et al., 2007a] (UX, P5X). Additionally, we consider the extension of previous ontologies with axioms in  $\mathcal{ELHI}$  and beyond DL-Lite (UXE, P5XE).

We have expanded previous assets with a set of synthetically-generated EBoxes, using a randomized and parameterised algorithm, with parameters:

**size:** the size of the EBox relative to the size of the TBox: zero is an empty EBox, one is an EBox with as many axioms as the TBox, two means an EBox with twice as many axioms as the TBox, etc.

**cover:** how much of the TBox is covered by the EBox: zero means that all the axioms in the EBox will be randomly generated, one means that all the axioms in the EBox will come from the TBox.

**reverse:** how many of the axioms obtained from the TBox (cover) are reversed in the EBox (the reverse of  $A \sqsubseteq B$  being  $B \sqsubseteq A$ ) with respect to the original form in the TBox: zero means that no axioms are reversed, one means that all axioms are reversed. The reversed axioms belong to the cover, i.e. if the cover is zero then this number has no effect.

All these parameters can be any real number between zero and one, except for size, which can be greater than one (e.g. a size of two means the EBox is twice the size of the TBox). Of course, the latter parameters are only significant when previous ones are greater than zero. If the cover is less than one then axioms

## 7. EVALUATION

Query independent information					query	Datalog time(ms)				Datalog size				UCQ time(ms)				UCQ size			
EBox	I	II	III	IV		I	II	III	IV	I	II	III	IV	I	II	III	IV	I	II	III	IV
PT	109	2047	24266	2859	1	0	0	157	235	15	13	14	9	0	0	516	672	15	13	14	9
PS	222	195	171	111	2	16	16	157	234	10	10	10	10	16	16	500	656	10	10	10	10
size	0.0	0.2	0.8	0.8	3	0	0	125	235	35	30	28	15	31	15	485	813	72	57	54	15
cover	0.0	0.8	0.2	0.8	4	0	16	219	188	41	38	22	16	63	94	735	719	185	170	3	42
rev	0.0	0.0	0.0	0.0	5	16	16	172	250	8	5	7	1	32	31	609	719	30	9	15	1
PT: preprocess time (ms)					6	0	15	234	188	18	14	14	11	0	15	578	641	18	14	14	11
PS: preprocessed size					7	0	0	125	172	27	23	23	20	94	125	1359	1359	180	140	140	110

Table 7.5: Results for ontology V (original size 222 clauses) with EBoxes I, II, III and IV.

up to one are generated by selecting randomly the LHS or RHS of other rules, adding an axiom  $A_1 \sqsubseteq A_2$  for each pair  $A_1, A_2$  found this way. For the  $A_i$  that are classes, nothing else is done. If some of them is a property  $P$  then the axiom  $\exists P$  is added and with a probability of  $1/2$  some other LHS or RHS  $A_3$  of some other rule is selected. If  $A_3$  is selected then  $\exists P$  is expanded into  $\exists P.A_3$ . This is repeated recursively until (1) a class is selected or (2) a property is selected and not expanded (expansion probability:  $1/2$ ).

The “size” is the main parameter. It is meant to help to evaluate the impact of the EBox on the results. The “cover” specifies how much of the EBox is related to the TBox and how much is random. It is meant to help to evaluate how the relation between EBox and TBox impacts on the results. The “reverse” specifies how many of the EBox axioms that are obtained from the TBox remain unaltered and how many are reversed. If the axiom is unaltered then the subconcepts are redundant, otherwise the superconcept is redundant. This parameter helps to evaluate the impact of redundancy in these cases.

We have run the tests with a set of EBoxes, selecting the values 0.0, 0.2, 0.4, 0.6, 0.8 and 1.0 for each of the parameters described above. When the size is zero then all the other parameters are zero, and when the cover is zero then reverse is zero as well. This involves a total of  $1 + 5 * (1 + 5 * 6) = 156$  EBoxes, for each ontology, used for all queries. Due to space limitations, we present a small excerpt of the results for a single ontology in Table 7.5; but the full results for all the ontologies and the EBoxes can be found online<sup>1</sup>.

<sup>1</sup><http://purl.org/net/jmora/extensionalqueryrewriting>

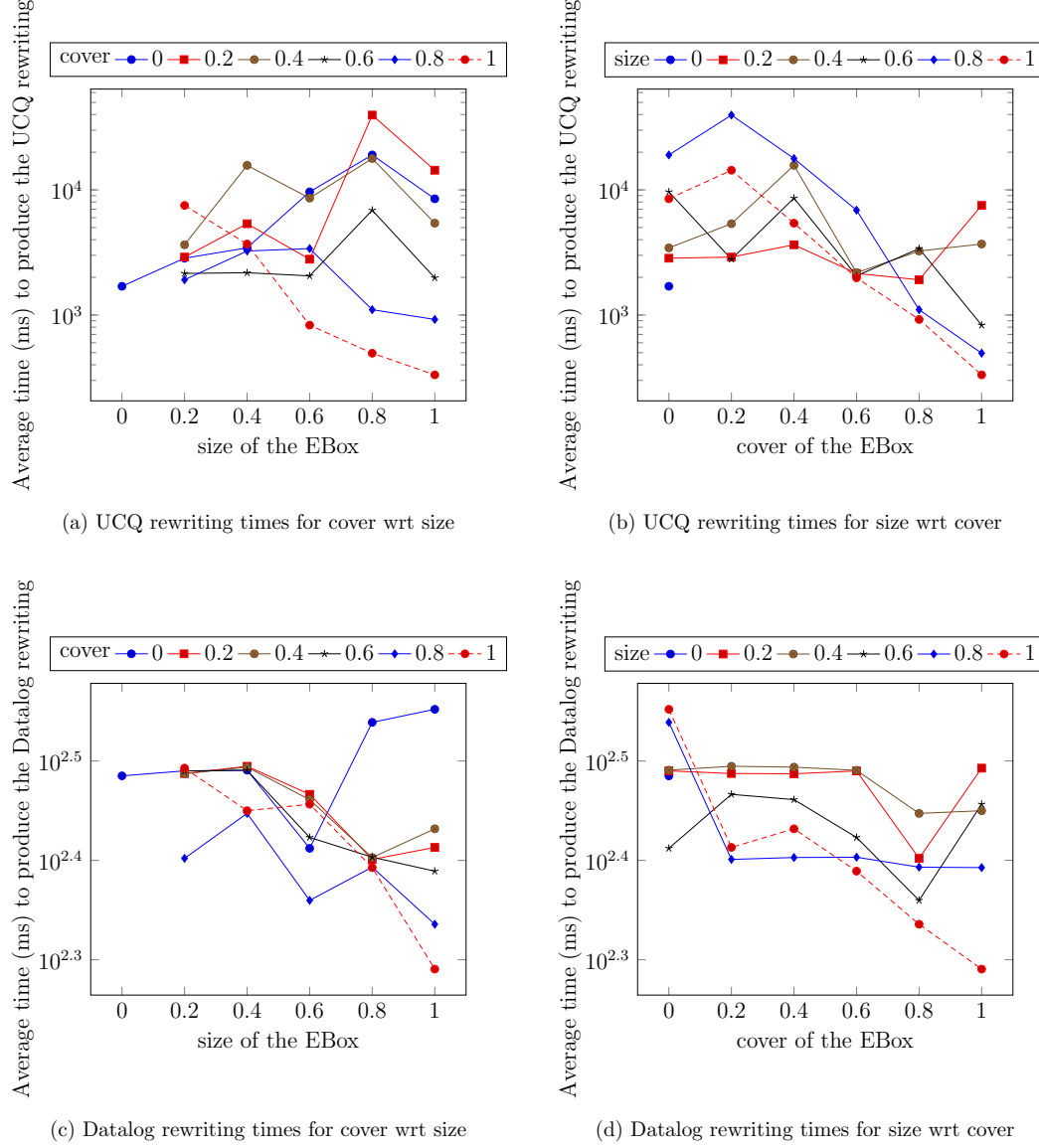
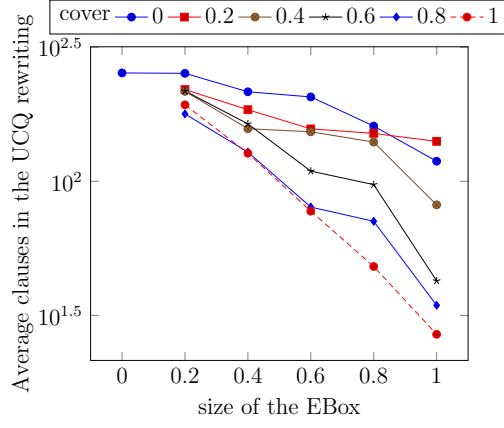


Figure 7.1: Average rewriting times relating size and cover. All queries and ontologies considered. “Reverse” is zero in all cases.

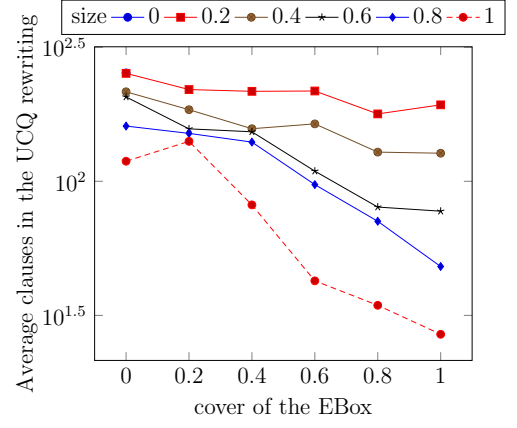
The results have been obtained on cold runs, by restarting the application after every query (passed in the application invocation parameters), again except for Presto2 (online). The consistency of the results regardless of how the system is run has been ensured by measuring the query rewriting time and discarding operations done before and after it. The hardware used in the evaluation is a Intel®Core™2 6300 @1.86GHz with 2GB of RAM, Windows® XP and

## 7. EVALUATION

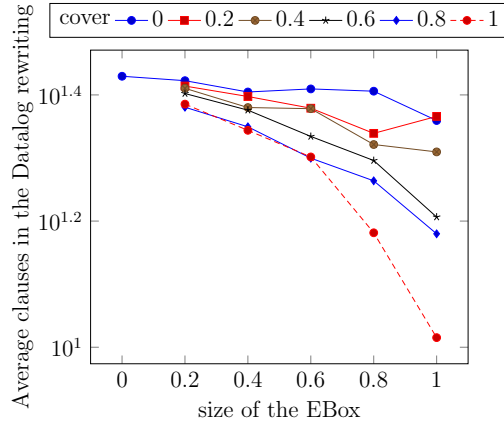
Java<sup>TM</sup> version 1.6.0\_33, with default settings for the Java Virtual Machine.



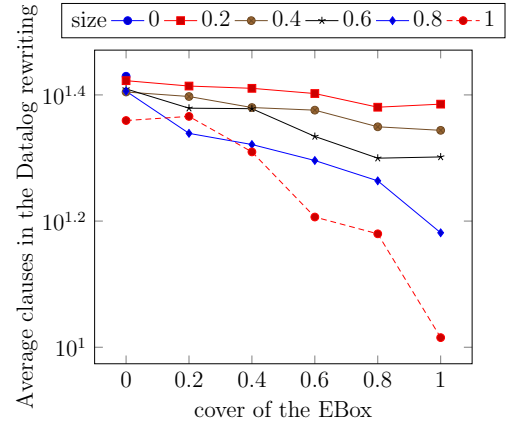
(a) number of clauses in the UCQ rewriting for cover wrt size



(b) number of clauses in the UCQ rewriting for size wrt cover



(c) number of clauses in the Datalog rewriting for cover wrt size



(d) number of clauses in the Datalog rewriting for size wrt cover

Figure 7.2: Average rewriting sizes relating size and cover. All queries and ontologies considered. “Reverse” is zero in all cases.



# Chapter 8

## Conclusions

This section contains the main conclusions that can be derived from the work presented in this thesis. These conclusions are grouped in different sections. There is one section for each chapter explaining the approach and there is a last section with general conclusions from the thesis as a whole.

### 8.1 Conclusions on Mapping Optimisations

We have presented an extension of the REQUIEM algorithm, used for query rewriting, where we propose to reduce the number of queries finally generated by adding two additional pruning steps. As a first consequence of the work performed, the number of queries can be reduced when considering the information provided by RDB2RDF mappings. There is a gain in efficiency due to this reduction, taking less time to rewrite the query and, most importantly, to execute it. Therefore, an algorithm that takes into account the capabilities of the source of information can make a better use of this queried source.

There is still room for improvement. The intersection among the individuals from several concepts may not be null and some may not be required to obtain the same result, specially among subclasses. This redundancy of information in related concepts is not considered. The user is responsible for specifying whether the individuals provided by some classes contain those provided by their respective subclasses. This is specified in a global manner, what means that if this is

## 8. CONCLUSIONS

---

the case for only some predicates then that information cannot be used. There is no possibility to specify more complex relations between the sets individuals provided by mapped predicates either. For example these relations (ABox dependencies [Rodríguez-Muro and Calvanese, 2012]) could be specified in an EBox [Rosati, 2012]. Thus, the description of the information provided by the source, with respect to the concepts provided, can be improved in this sense.

Similarly to the EBox, additional meta-information regarding the data sources and their information could be considered. When the same concept is provided by several data sources the individuals that these contain may overlap in many different ways. These concepts could have information about their provenance in a similar way in which concept inclusion can be considered, for example at the level of sources or mapping assertions. This would allow the algorithm to differentiate among the concepts that are provided by different sources and again the overlap that may exist among them. This additional information would enable algorithms that discriminate based on provenance, to avoid parts (e.g. clauses) in rewritten queries that are redundant with respect to another part in the same query. The consequence would be shorter rewritten queries that the underlying software can optimise and execute more easily. Additional considerations could be made if different sources of information had different characteristics, like latency, accuracy, prices, etc.

It must be noted that the improvement presented also depends on the mappings that are provided. In an extreme case, if all the concepts in the ontology were mapped to the database then the application of the algorithm would be pointless. We can see this by considering the consequences of applying both working modes. 1) The results provided on the H (prune to highest) mode would simply be the query provided with no inference on it. 2) The results provided on the L (prune to lowest) mode of this modification would be those provided by the original REQUIEM. In both working modes the optimisations imply a computational overhead that could be prevented. 1) No rewriting needs to be done to obtain the results in the first case. 2) No optimisation in terms of mappings needs to be done in the second case, that can be achieved by using the A (keep all) mode, corresponding to the original unmodified REQUIEM algorithm. We have seen, as shown on table 4.1, that the cases we can usually find are very far

---

from this extreme case.

Finally, the partial unfolding process performed after saturation preserves all the possible inferences by anticipating some of them. More precisely, those inferences related with the clauses that contain non-mapped predicates, which are the clauses that will be removed. This partial unfolding may result in an increase of the number of clauses despite of the removal of some of these clauses. For instance, if 1) several non-mapped predicates are present in the body of a clause and 2) these predicates are at the head of several clauses, then the possible combinations of unifications of clauses in the latter group with clauses in the former group may exceed the original number of clauses. We can see this more easily with an example. Consider the following Datalog program:

$$\begin{array}{ll} Q(x, y, z) \leftarrow B(x), R(x, y), B(y), R(y, z) & B(x) \leftarrow B1(x) \\ R(x, y) \leftarrow R1(x, y) & B(x) \leftarrow B2(x) \\ R(x, y) \leftarrow R2(x, y) & B(x) \leftarrow B3(x) \end{array}$$

If the mapped predicates are B1, B2, B3, R1 and R2 then the (partial) unfolding that is required in this case to remove non-mapped predicates is complete, producing the following clauses.

$$\begin{array}{l} Q(x, y, z) \leftarrow B1(x), R1(x, y), B1(y), R1(y, z) \\ Q(x, y, z) \leftarrow B2(x), R1(x, y), B1(y), R1(y, z) \\ Q(x, y, z) \leftarrow B3(x), R1(x, y), B1(y), R1(y, z) \\ Q(x, y, z) \leftarrow B1(x), R2(x, y), B1(y), R1(y, z) \\ Q(x, y, z) \leftarrow B2(x), R2(x, y), B1(y), R1(y, z) \\ Q(x, y, z) \leftarrow B3(x), R2(x, y), B1(y), R1(y, z) \\ Q(x, y, z) \leftarrow B1(x), R1(x, y), B2(y), R1(y, z) \\ Q(x, y, z) \leftarrow B2(x), R1(x, y), B2(y), R1(y, z) \\ Q(x, y, z) \leftarrow B3(x), R1(x, y), B2(y), R1(y, z) \\ Q(x, y, z) \leftarrow B1(x), R2(x, y), B2(y), R1(y, z) \\ Q(x, y, z) \leftarrow B2(x), R2(x, y), B2(y), R1(y, z) \\ Q(x, y, z) \leftarrow B3(x), R2(x, y), B2(y), R1(y, z) \\ Q(x, y, z) \leftarrow B1(x), R1(x, y), B3(y), R1(y, z) \end{array}$$

## 8. CONCLUSIONS

---

$Q(x,y,z) \leftarrow B2(x), R1(x,y), B3(y), R1(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R1(x,y), B3(y), R1(y,z)$   
 $Q(x,y,z) \leftarrow B1(x), R2(x,y), B3(y), R1(y,z)$   
 $Q(x,y,z) \leftarrow B2(x), R2(x,y), B3(y), R1(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R2(x,y), B3(y), R1(y,z)$   
 $Q(x,y,z) \leftarrow B1(x), R1(x,y), B1(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B2(x), R1(x,y), B1(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R1(x,y), B1(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B1(x), R2(x,y), B1(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B2(x), R2(x,y), B1(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R2(x,y), B1(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B1(x), R1(x,y), B2(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B2(x), R1(x,y), B2(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R1(x,y), B2(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B1(x), R2(x,y), B2(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B2(x), R2(x,y), B2(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R2(x,y), B2(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B1(x), R1(x,y), B3(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B2(x), R1(x,y), B3(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R1(x,y), B3(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B1(x), R2(x,y), B3(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B2(x), R2(x,y), B3(y), R2(y,z)$   
 $Q(x,y,z) \leftarrow B3(x), R2(x,y), B3(y), R2(y,z)$

We can see here a combinatorial explosion and as we can see, these combinatorial explosions are theoretically possible. Without the mappings optimisation, non-mapped predicates would be preserved, and the Datalog program would not be unfolded when rewriting to Datalog. If the rewriting targets to a UCQ then the UCQ would look exactly equal to the previously shown Datalog program if we consider the mapping optimisation. If the mapping optimisation is not applied then the UCQ would contain more clauses, as it would include non-mapped predicates ( $B$  and  $R$ ).

---

## 8.2 Conclusions on Engineering Optimisations

The process to obtain the optimisations has been tightly coupled with the evaluation of the results. The nature of these optimisations is purely quantitative and its evaluation is experimental. Optimisations that were plausible from a theoretical perspective needed to be implemented and evaluated to verify empirically whether the process would perform better or not. In some cases, the improvement in the process would not pay off for the additional cost of the computation, and so those candidates were discarded.

This can be compared with test driven development (TDD) [Astels, 2003]. Shortly we can say that in TDD the tests are designed and implemented first and then the remainder of the development of the software is done to pass these tests. In our case, the benchmark used for the evaluation of the optimisations was an important asset to guide the analysis, design and implementation of the optimisations. This is indeed different from TDD where test are qualitative and are meant to guarantee that certain functionalities are covered by the software. In our case, the correctness (completeness and soundness) in the results had to be maintained, but the results had to be obtained faster (and if possible better results should be obtained). We may consider that this diverges from TDD and may be more similar to what could be considered as “benchmark driven development” or “experiment driven research”. This is a topic that has not received much attention [Gavras, 2010; Prange, 1996] and the state of the art seems not to be clearly defined. Properly defining these concepts would require extensive work beyond OBDA and query rewriting thus, we can simply say that we have performed TDD with one main difference: the objective is not a software that simply passes the tests, but a software that:

- passes the tests (in the benchmark), which means that it is verified for correctness according to them (and in addition formal proofs in our case).
- behaves in a qualitatively better way in the tests with respect to performance (time to obtain the results)
- produces qualitatively better results (size of the rewritten queries in our case).

## 8. CONCLUSIONS

---

### 8.3 Conclusions on EBox Optimisations

From the analysis of the evaluation results we can conclude that:

- In the computation of Datalog rewritings, using the EBox allows for obtaining equal or smaller Datalog programs for all queries, with negligible effects on the query rewriting time.
- For UCQ rewritings, our results show that, in general, the number of clauses is diminished as the EBox increases in size and similarity ("cover") with the TBox. This reduction in the number of clauses usually implies a reduction in the time required for the query rewriting process, as we can see in figure 7.1. This figure shows the average time to produce a UCQ or a Datalog rewriting and how EBoxes with different sizes and TBox coverage influence this time. More precisely:
  - We can notice that EBoxes with both a high value for cover and size tend to reduce the query rewriting time when compared with other EBoxes or no EBox.
  - When the EBox involves more random axioms (e.g. EBox III in Table 2 with 0.8 for "size" and 0.2 for "cover") the results are less predictable. More specifically, we can see that for UCQ size the EBox III behaves better than EBox IV in query 4 and it behaves worse than EBox II in query 5. This variation depends on the query, the EBox may contain axioms that imply the subsumption of atoms in the query. Notice that if these axioms were in the TBox, then their presence in the EBox would have no impact in the results (EBox IV). However, when these axioms are not in the TBox, then this can lead to further clause condensation, with the elimination of subsumed clauses and the ones generated from these. This elimination of clauses means a potentially strong reduction in the size of the results and the time required to produce them (as seen in chapter 5).
- Even in the cases where the query rewriting times are higher, we can see an important reduction in the number of clauses generated, in the Datalog

---

and UCQ rewritings, as figure 7.2 shows. This reduction in the number of clauses implies a reduction in the redundancy of the queries that are generated, with the consequent simplification of the computation required for answering the query by the other layers of the OBDA system.

## 8.4 Conclusions from the evaluation

We can group the shortcomings we have seen in section 7.1 into two main groups, one related with the input of the systems and the other one related with the output.

On the side of the **input** we need to know how well the tests represent reality. This is in terms of (1) queries with respect to (1a) syntax, (1b) expressiveness, (1c) shape and (1d) size, (2) ontologies in terms of (2a) shape, (2b) size and (2c) expressiveness and whether it is possible to consider (3) additional information such as (3a) mappings availability, (3b) ABox dependencies or EBoxes or (3c) others. We can see this in figure 8.1.

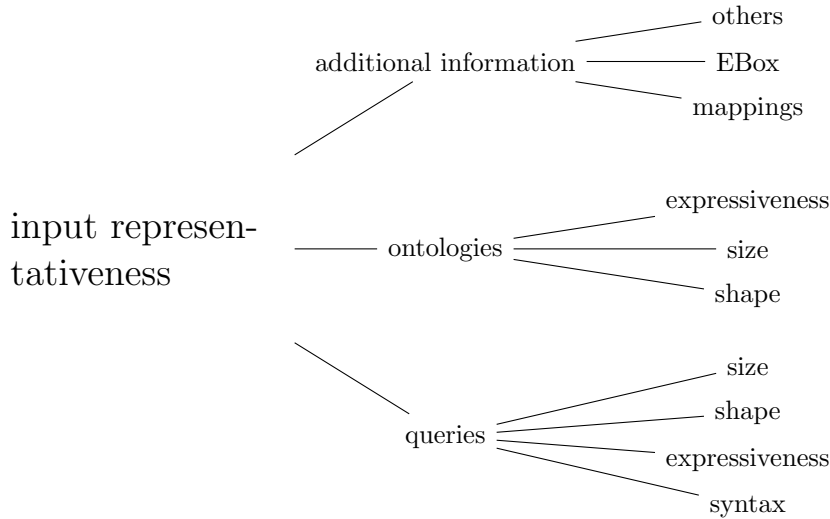


Figure 8.1: Input dimensions that may affect benchmark representativeness.

On the side of the **output** we need to know how well rewritten queries may perform when posed to some other system, and again we should focus on the same details like the (1) shape in terms of (1a) expressiveness (Property paths

## 8. CONCLUSIONS

---

in SPARQL, subqueries in Datalog, UCQs), (1b) types of clauses (non-recursive Datalog, linear Datalog), (1c) syntax with special characteristics (SPARQL, SQL, Datalog), and (2) size in terms of (2a) number of clauses, (2b) number of atoms and distinct atoms, (2c) number of joins. We can see this in figure 8.2.

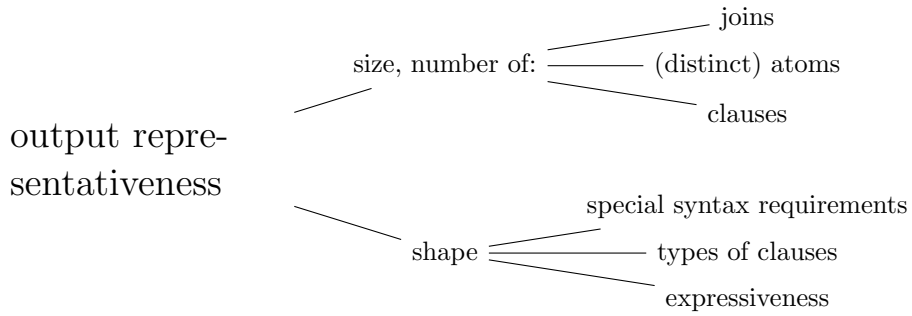


Figure 8.2: Output dimensions that may affect the representativeness of the evaluations done with benchmarks.

In the experimental results we have seen these characteristics of the input of these systems are relevant and may cause the output to vary greatly. Clearly the same principle may apply to underlying systems, which means that the same care should be put on the evaluation of the output.

We have given a first step to the solution of these shortcomings. Nowadays we can make better comparisons thanks to the additional approaches and how they behave in different testcases. We can see that:

- **REQUIEM** was an interesting addition to the state of the art and being among the most expressive systems allows comparisons with systems that handle less expressive logics.
- **Rapid** has improved REQUIEM in every aspect except expressiveness, which is more reduced. If the ontology can be handled by Rapid then this is the most competitive approach on the light of the data.
- **Prexto** is unique among the evaluated approaches in the handling of the EBox. The EBox can produce dramatic improvements by reducing the size of the rewritten query and the time to obtain it.



- 
- **Nyaya** is the only system that paid attention to the size of the queries in terms of atoms and joins. As the area matures and standardization continues we will be able to evaluate the impact of these factors on the underlying systems.
  - **Clipper** improves previous approaches in terms of expressiveness, it handles the most expressive logic among the evaluated approaches (Horn- $\mathcal{SHJQ}$ ) but its output is also the most expressive, only Datalog meant to be evaluated by Datalog engines. It offers the most and it also requires the most.
  - **kyrie** has improved REQUIEM in every aspect except expressiveness, which is exactly the same. If the ontology cannot be handled by Rapid then kyrie allows handling this expressiveness more efficiently than REQUIEM.

As more systems are developed and evaluated, we will be able to ascertain with more detail the impact of their differences by analysing how they behave. As more test data is available, we will be able to determine with better accuracy the significance of each test, i.e. how representative they are among all the tests. Finally, as the area goes through standardization and the underlying systems are standardized in the area, we will be able to compare how well the output of rewriting systems can be handled by these underlying systems.

We have shown that it is already possible to do this evaluation despite of current limitations and we have pointed at current limitations and possibilities for the future.

## 8.5 General Conclusions

We have considered the problem of query rewriting in OBDA and made proposals on its three main aspects.

**Input** We have considered additional elements that could be used as part of the input. This requires looking at the problem from a *broader* perspective. The mappings are required for OBDA, they add no additional requirements or constraints to the whole problem. However the mappings are not used in the query rewriting process. By including the mappings as part of the

## 8. CONCLUSIONS

---

input we obtain additional benefits, as we have seen. Additionally, an EBox may provide additional information about the extensional containment of predicates and projections of predicates in the queries, with consequent optimisations.

**Process** The process is dependent on the logics that are used. In our case we have chosen the very expressive logic  $\mathcal{ELHIQ}$ . With this logic fixed as a requirement we have performed optimisations in the query rewriting process by looking at the process in a *detailed* way. The impact of these optimisations is comparable to the impact that can be obtained by modifying the expressiveness. This shows the relevance of engineering optimisations and research in algorithms for automated resolution, in this case applied to the specific context of query rewriting.

**Output** The output is improved by both previous elements. A shorter output in equal complexity conditions means less work load for the systems processing the query. The output has been improved and has been analysed. Special attention has been paid to the evaluation that can be done of these systems and how it can be done, making contributions in this area as well.

These tasks have been done mostly with a pragmatical and applied approach in mind. At the same time, query rewriting approaches in the state of the art are mostly focused on theoretical considerations. Due to these facts we consider to have a balanced proposal with respect to theoretical and pragmatical considerations. A theoretical evaluation (completeness and soundness) is required in these cases and it has been included in the corresponding sections. Additionally, we have given the first steps towards a qualitative evaluation. We have collected assets and used them to derive these qualitative conclusions that allow for a more pragmatical evaluation. We have also proposed some guidelines for the extension and improvement of this qualitative evaluation. As we have seen and as we would like to remark and summarise, in this thesis attention has been paid and contributions have been made to the specific details, to the broader picture, to the theoretical aspects and to the pragmatical aspects of the problem.

Through several chapters, we have seen several points that advise the estimation of the capabilities and characteristics of the whole OBDA system to optimise

---

the process and results of query rewriting further. We have seen that the lack of mappings for some predicates can help to produce shorter rewritings in shorter times. In a modular system, one particular point to consider is the module responsible of processing the rewritten query, its efficiency for different types of queries and its capabilities, e.g. the possibility of processing UCQs, Datalog or recursive Datalog. In addition, having information about the extensional containment of some predicates may help to produce much shorter rewritings in shorter times reducing the redundancy in the answers.

While this further optimisation could be performed in one single way, it is also possible that addressing specific characteristics means adapting to different domains for each cluster of characteristics. For this reason, we also point the convenience of further analysis of the domains in which OBDA is used and the characteristics they may present.

## 8.6 Future work

In this thesis we have focused on optimising query rewriting for ontology based data access. The challenges to overcome in the future will depend on the use that is given to this technology. Query rewriting is an approach that allows abstracting from the data sources, while considering some of their characteristics (e.g. the predicates that are mapped). For example, other approaches for OBDA rely on the possibility to modify the schema of the database (e.g. Ultrawrap [Sequeda and Miranker, 2013]) or the mappings (e.g. OnTop [Rodríguez-Muro and Calvanese, 2012]). In the case of query rewriting some of the most interesting lines to explore in the future are those where the modification of the schema of the data source or the mappings is not possible. One example in this case is querying web services [Arpinar et al., 2005; Gray et al., 2011; Thakkar et al., 2002], the so-called Deep Web [He et al., 2007]. Another example is querying data streams [Calbimonte et al., 2010] and the so-called Internet of Things [Atzori et al., 2010]. Basically, not requiring modifications on the data sources and abstracting from them as much as possible eases a future integration with a wider range of data source types.

If we want to keep a good generality and applicability with this abstraction

## 8. CONCLUSIONS

---

one aspect that requires special focus are the interfaces. A good way to ensure interoperability is complying with well established standards. While Datalog is a general language that has been and is currently widely used, a standard with particular interest for queries in the semantic web is SPARQL. SPARQL presents new challenges and opportunities to explore when used as the target or source language for the process of query rewriting.

SPARQL has an expressiveness comparable to Datalog [Angles and Gutierrez, 2008; Pérez et al., 2009], but SPARQL 1.1 has some additional features beyond those offered by SPARQL. Due to the expressiveness in SPARQL 1.1 and more specifically the property paths, some recursion is allowed, which is powerful and complex at the same time [Arenas et al., 2012; Losemann and Martens, 2012]. Additionally, subqueries are possible, with similar consequences [Angles and Gutierrez, 2011]. Both features allow to rewrite a linearly recursive Datalog query into a SPARQL 1.1 query, with subqueries for the intensional predicates and property paths for some recursive predicates. Queries can also contain operators for comparisons, filtering and aggregation among others, which are not supported by conventional Datalog. This opens several challenges and possibilities to explore that are enabled by the expressiveness in SPARQL 1.1. For example the type of recursion in  $\mathcal{ELHJO}$  seems to be translatable to property paths. This means that possibly more expressive logics could also be handled in the rewriting generating SPARQL 1.1 queries. Possibly some features in the Horn- $\mathcal{SHJQ}$  that Clipper handles could be supported by rewriting to SPARQL 1.1. The expressiveness of SPARQL could extend further beyond Datalog, since SPARQL offers an additional possibility beyond first-order rewritability, the predicates in the triples can be variables and can be unified.

Besides the language of the rewritten queries, we can consider as well the language for the original queries. If the rewritten queries are in SPARQL, one perfect match for the original queries would be SPARQL as well. In the present thesis we have rewritten unions of conjunctive queries, but SPARQL queries pose new challenges and opportunities. We can clearly see that the structure in SPARQL queries is different and includes additional operators (e.g. aggregation). In the case of SPARQL, the semantics of SPARQL entailment are expected, but they can be extended with richer semantics like OWL [Glimm, 2011; Glimm et al.,

---

2012; Kollia et al., 2011; Polleres, 2007]. However this work and the work done on OBDA have progressed separately. Bridging this gap should mean an improvement for both communities, increase usability, applicability and adoption.

Additionally, due to the abstraction from the mappings, these mappings could map several data sources, allowing data integration and federated queries [Buil-Aranda et al., 2013]. In such a context, further optimisations could be considered, for example join operations between predicates in a data source would probably be less expensive (in computational terms) than join operations between predicates in different data sources. When considering several data sources additional considerations could be done with respect to provenance and data quality. These considerations would definitively require additional metadata and would impose strong modifications in the algorithms. In such a scenario, the correctness of the result and the efficiency of the process and the rewritten query would remain relevant. However, the focus would be on the quality of the query and its answers, which would certainly require new assets for the evaluation.

We have also seen from the optimisations and the evaluation that a quantitative and empirical approach may be beneficial for query rewriting. We have given a few steps in that direction, but there is still much work to be done in that line. Better benchmarks could be developed and better tools for benchmarking as well. On the side of the benchmarks, they could be improved with a good compilation of use cases including ontologies, mappings, queries and even datasets if possible. In this sense initiatives like Mappingpedia<sup>1</sup> could be useful. On the side of the tools, we can consider the creation of an automated suite of tests to be run on a platform like the one provided by SEALS [García-Castro et al., 2010]. There are two aspects that are particularly interesting in such a context. On the one hand, different systems for query rewriting could be compared with each other. Having a full OBDA system or a set of OBDA systems where query rewriting would be integrated would allow to determine the characteristics of the rewritten queries that work better with other modules of the OBDA system, and at the same time the characteristics of those modules that are relevant for query rewriting. On the other hand, OBDA systems that rely on query rewriting could be compared with OBDA systems that use different techniques (e.g. mapping saturation).

---

<sup>1</sup><http://mappingpedia.linkeddata.es/>

## 8. CONCLUSIONS

---

Certainly, not all OBDA systems can be applied in the same context, as some have requirements that other systems do not have (e.g. possibility of materialising some facts, modifying the database schema, etc.), however, this comparison would allow to check details as for example the impact on performance caused by a different set of requirements.

Finally, in a much more general perspective, OBDA allows accessing data sources using ontologies. For this, some data is calculated, mainly performing operations on strings to compose the URIs for the individuals in the ontology. This calculation could be taken much further, specially when considering datatype properties, numbers and vectors. As the amount of data and their quality increase, the value of data analysis increases as well, and there is an important trend in the analysis of this data and data science, some authors do even talk about a “Fourth Paradigm” [Hey et al., 2009] when considering data-intensive science. The combination of data analysis techniques with OBDA could allow to access datatype properties, concepts and values that are calculated numerically from data with a different granularity. A data scientist in this case would not *calculate* a mean, an average or search for outliers, but would *query* for them. There are many problems in the use of OBDA for big data [Calvanese et al., 2013] that might need to be solved in satisfactory ways to allow the addition of a more complex analysis in an OBDA system. Therefore, this is a line for a far future where much exploration needs to be done yet.

# References

- Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.
- Jose Luis Ambite, Craig A. Knoblock, and Steven Minton. Learning plan rewriting rules. *Artificial Intelligence Planning Systems*, 10:7Y, 2000.
- Renzo Angles and Claudio Gutierrez. The Expressive Power of SPARQL. In *Proceedings of the 7th International Conference on The Semantic Web, ISWC '08*, pages 114–129, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88563-4. doi: [http://dx.doi.org/10.1007/978-3-540-88564-1\\_8](http://dx.doi.org/10.1007/978-3-540-88564-1_8). ACM ID: 1483165.
- Renzo Angles and Claudio Gutierrez. Subqueries in SPARQL. In *AMW*, volume 749 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.
- Krzysztof R. Apt. Introduction to Logic Programming. Technical report, University of Texas at Austin, Austin, TX, USA, 1988.
- Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting Beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard. In *Proceedings of the 21st International Conference on World Wide Web, WWW '12*, pages 629–638, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1229-5. doi: 10.1145/2187836.2187922.
- Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

## REFERENCES

---

- Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems*, 6(2):99–130, June 1996. doi: 10.1007/BF00122124.
- Yigal Arens, Chun-Nan Hsu, and Craig A. Knoblock. Query processing in the SIMS information mediator. In *Readings in agents*, pages 82–90. Morgan Kaufmann Publishers Inc., 1998. ISBN 1-55860-495-2.
- I. Budak Arpinar, Ruoyan Zhang, Boanerges Aleman-Meza, and Angela Maduko. Ontology-driven Web services composition platform. *Information Systems and e-Business Management*, 3(2):175–199, July 2005. ISSN 1617-9846, 1617-9854. doi: 10.1007/s10257-005-0055-9.
- Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyashev. The DL-Lite family and relations. *J. Artif. Int. Res.*, 36(1): 1–69, 2009.
- Dave Astels. *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. ISBN 0-13-101649-0.
- Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010. ISSN 1389-1286. doi: 10.1016/j.comnet.2010.05.010.
- Leo Bachmair and Harald Ganzinger. Resolution theorem proving. *Handbook of automated reasoning*, 1:19–99, 2001.
- Jesús Barrasa, Oscar Corcho, and Asunción Gómez-Pérez. R2o, an Extensible and Semantically based Database-to-Ontology Mapping Language. *2nd Workshop on Semantic Web and Databases (SWDB2004)*, 3372, 2004.
- Domenico Beneventano, Sonia Bergamaschi, Silvana Castano, Alberto Corni, R. Guidetti, G. Malvezzi, Michele Melchiori, and Maurizio Vincini. Information Integration: The MOMIS Project Demonstration. In *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00*, pages 611–614, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3.



## REFERENCES

---

- Christian Bizer. D2r MAP - A Database to RDF Mapping Language. In *The Twelfth International World Wide Web Conference*, Budapest, Hungary, May 2003.
- Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. CLASSIC: a structural data model for objects. *SIGMOD Rec.*, 18(2):58–67, June 1989. ISSN 0163-5808. doi: 10.1145/66926.66932.
- Carlos Buil-Aranda, Marcelo Arenas, Oscar Corcho, and Axel Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *Web Semantics: Science, Services and Agents on the World Wide Web*, 18(1):1–17, January 2013. ISSN 1570-8268. doi: 10.1016/j.websem.2012.10.001.
- Marco Cadoli, Luigi Palopoli, and Maurizio Lenzerini. Datalog and description logics: Expressive power. In *Database Programming Languages*, number 1369 in Lecture Notes in Computer Science, pages 281–298. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-64823-9 978-3-540-68534-0.
- Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC 2010*, pages 96–111, 2010. ISBN 3-642-17745-X.
- Jean-Paul Calbimonte, Hoyoung Jeung, Oscar Corcho, and Karl Aberer. Enabling Query Technologies for the Semantic Sensor Web. *International Journal On Semantic Web and Information Systems*, 8, 2012.
- Andrea Cali, Georg Gottlob, and Andreas Pieris. Query Answering under Non-guarded Rules in Datalog+/- . In *Web Reasoning and Rule Systems*, volume 6333, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-15917-6 978-3-642-15918-3.
- Andrea Cali, Georg Gottlob, and Andreas Pieris. New Expressive Languages for Ontological Query Answering. In *AAAI*. AAAI Press, 2011.
- Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Vardi. What Is Query Rewriting? In *Cooperative Information Agents IV - The Future of Information Agents in Cyberspace*, volume 1860 of *Lecture Notes in*

## REFERENCES

---

- Computer Science*, pages 439–457. Springer Berlin / Heidelberg, 2000. ISBN 978-3-540-67703-1.
- Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated Reasoning*, 39(3):385–429, October 2007a. doi: 10.1007/s10817-007-9078-x.
- Diego Calvanese, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Ontology-Based Database Access. Technical report, CiteSeerX, 2007b.
- Diego Calvanese, Ian Horrocks, Ernesto Jiménez-Ruiz, Evgeny Kharlamov, Michael Meier, Mariano Rodríguez-Muro, and Dmitriy Zheleznyakov. On Rewriting and Answering Queries in OBDA Systems for Big Data (Short Paper). In *OWL Experiences and Directions Workshop (OWLED)*, 2013.
- Caterina Caracciolo, Juan Heguiabehere, Aldo Gangemi, Wim Peters, and Armando Stellato. Second Network of Fisheries Ontologies. deliverable D7.2.4, NeOn project, 2010.
- Tiziana Catarci and Maurizio Lenzerini. Representing and Using Interschema Knowledge in Cooperative Information Systems. *Int. J. Cooperative Inf. Syst.*, 2(4):375–398, 1993.
- Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989. ISSN 1041-4347. doi: 10.1109/69.43410.
- Alexandros Chortaras, Despoina Trivela, and Giorgos Stamou. Optimized Query Rewriting for OWL 2 QL. In *Automated Deduction – CADE-23*, volume 6803, pages 192–206. Springer, 2011. ISBN 978-3-642-22437-9.
- Marco Console, Maurizio Lenzerini, Riccardo Mancini, Riccardo Rosati, and Marco Ruzzi. Synthesizing Extensional Constraints in Ontology-Based Data Access. In *Description Logics*, pages 628–639, 2013.

## REFERENCES

---

- Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):309–322, November 2008. ISSN 1570-8268. doi: 10.1016/j.websem.2008.05.001.
- Souripriya Das, Seema Sundara, and Richard Cyganiak. R2rml: RDB to RDF Mapping Language. *W3C RDB2RDF Working Group*, W3C recommendation, September 2012.
- Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. A hybrid system with datalog and concept languages. In *Trends in Artificial Intelligence*, number 549 in Lecture Notes in Computer Science, pages 88–97. Springer Berlin Heidelberg, January 1991. ISBN 978-3-540-54712-9 978-3-540-46443-3.
- Thomas Eiter, Magdalena Ortiz, Mantas Simkus, Trung-Kien Tran, and Guohui Xiao. Query rewriting for Horn-SHIQ plus rules. In *26th AAAI Conference on Artificial Intelligence*, 2012.
- Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, 1994.
- Marc Friedman and Daniel S. Weld. Efficiently Executing Information-Gathering Plans. In *In Proc. of the Int. Joint Conf. of AI (IJCAI)*, pages 785–791, 1997.
- Marc Friedman, Alon Levy, and Todd Millstein. Navigational plans for data integration. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference*, pages 67–73, Orlando, Florida, United States, 1999. ISBN 0-262-51106-1.
- Raúl García-Castro, Miguel Esteban-Gutiérrez, and Asunción Gómez-Pérez. Towards an infrastructure for the evaluation of semantic technologies. In *eChallenges, 2010*, pages 1–7, October 2010.
- Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom.

## REFERENCES

---

- The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997. ISSN 0925-9902, 1573-7675. doi: 10.1023/A:1008683107812.
- Anastasius Gavras. *Experimentally driven research white paper*. Technical report, ICT Fireworks (April 2010), 2010.
- Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, April 2011. ISSN 0921-7126.
- Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: an information integration system. *SIGMOD Rec.*, 26(2):539–542, June 1997. ISSN 0163-5808. doi: 10.1145/253262.253400.
- Birte Glimm. Using SPARQL with RDFS and OWL Entailment. In *Reasoning Web. Semantic Technologies for the Web of Data*, number 6848 in Lecture Notes in Computer Science, pages 137–201. Springer Berlin Heidelberg, January 2011. ISBN 978-3-642-23031-8 978-3-642-23032-5.
- Birte Glimm, Aidan Hogan, Markus Krötzsch, and Axel Polleres. OWL: Yet to arrive on the Web of Data? *arXiv:1202.0984*, February 2012.
- François Goasdoué and Chantal Reynaud. Modeling Information Sources for Information Integration. In *Knowledge Acquisition, Modeling and Management*, number 1621 in Lecture Notes in Computer Science, pages 121–138. Springer Berlin Heidelberg, January 1999. ISBN 978-3-540-66044-6 978-3-540-48775-3.
- Asunción Gómez-Pérez, José Ángel Ramos, Antonio Federico Rodríguez-Pascual, and Luis Manuel Vilches-Blázquez. The IGN-E Case: Integrating Through a Hidden Ontology. In *Headway in Spatial Data Handling*, Lecture Notes in Geoinformation and Cartography, pages 417–435. Springer, 2008.
- Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological Queries: Rewriting and Optimization (Extended Version). *arXiv:1112.0343*, December 2011.

## REFERENCES

---

- Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity Conditions and their Application to Query Answering in Description Logics. In *KR*. AAAI Press, 2012. ISBN 978-1-57735-560-1.
- Alasdair J. G. Gray, Raúl García-Castro, Kostis Kyzirakos, Manos Karpapothakis, Jean-Paul Calbimonte, Kevin Page, Jason Sadler, Alex Frazer, Ixent Galpin, Alvaro A. A. Fernandes, Norman W. Paton, Oscar Corcho, Manolis Koubarakis, David De Roure, Kirk Martinez, and Asunción Gómez-Pérez. A Semantically Enabled Service Architecture for Mashups over Streaming and Stored Data. In *The Semantic Web: Research and Applications*, number 6644 in Lecture Notes in Computer Science, pages 300–314. Springer Berlin Heidelberg, January 2011. ISBN 978-3-642-21063-1 978-3-642-21064-8.
- Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2–3):158–182, October 2005. ISSN 1570-8268. doi: 10.1016/j.websem.2005.06.005.
- Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the Deep Web. *Commun. ACM*, 50(5):94–101, May 2007. ISSN 0001-0782. doi: 10.1145/1230819.1241670.
- Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft, 2009.
- Martha Imprialou, Giorgos Stoilos, and Bernardo Cuenca Grau. Benchmarking ontology-based query rewriting systems. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, AAAI*, 2012.
- Donald B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1):77, 1975. ISSN 00975397. doi: 10.1137/0204007.
- C. Maria Keet, Ronell Alberts, AURONA Gerber, and Gibson Chimamiwa. *Enhancing web portals with Ontology-Based Data Access: the case study of South Africa’s Accessibility Portal for people with disabilities*. OWLED, 2008.

## REFERENCES

---

- Ilianna Kollia, Birte Glimm, and Ian Horrocks. SPARQL Query Answering over OWL Ontologies. In *The Semantic Web: Research and Applications*, number 6643 in Lecture Notes in Computer Science, pages 382–396. Springer Berlin Heidelberg, January 2011. ISBN 978-3-642-21033-4 978-3-642-21034-1.
- Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyashev. Combined FO Rewritability for Conjunctive Query Answering in DL-Lite. In *Description Logics*, volume 477 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.
- Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyashev. The Combined Approach to Query Answering in DL-Lite. *PROCEEDINGS OF KR 2010*, 2010.
- Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, Madison, Wisconsin, 2002. ACM. ISBN 1-58113-507-6. doi: 10.1145/543613.543644.
- Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, July 2006. ISSN 1529-3785. doi: 10.1145/1149114.1149117.
- Alon Y. Levy. Logic-based techniques in data integration. In *Logic-based artificial intelligence*, pages 575–595. Kluwer Academic Publishers, 2000. ISBN 0-7923-7224-7.
- Alon Y. Levy and Marie-Christine Rousset. Combining Horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1–2):165–209, September 1998. ISSN 0004-3702. doi: 10.1016/S0004-3702(98)00048-4.
- Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22th International Conference on Very Large Data Bases, VLDB '96*, pages 251–262, San Francisco, CA, USA, 1996a. Morgan Kaufmann Publishers Inc. ISBN 1-55860-382-4.

## REFERENCES

---

- Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-Answering Algorithms for Information Agents. In *AAAI/IAAI, Vol. 1*, pages 40–47. AAAI Press / The MIT Press, 1996b. ISBN ISBN 0-262-51091-X.
- Katja Losemann and Wim Martens. The Complexity of Evaluating Path Expressions in SPARQL. In *Proceedings of the 31st Symposium on Principles of Database Systems, PODS '12*, pages 101–112, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1248-6. doi: 10.1145/2213556.2213573.
- Christopher Lynch. Oriented Equational Logic Programming is Complete. *Journal of Symbolic Computation*, 23(1):23–45, January 1997. ISSN 0747-7171. doi: 10.1006/jsc.1996.0075.
- Robert MacGregor and Raymond Bates. The Loom Knowledge Representation Language. Technical report, DTIC Document, May 1987.
- Robert M. MacGregor. A description classifier for the predicate calculus. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, AAAI '94, pages 213–220, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. ISBN 0-262-61102-3.
- David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, December 1979. ISSN 0362-5915. doi: 10.1145/320107.320115.
- Eduardo Mena, Arantza Illarramendi, Vipul Kashyap, and Amit P. Sheth. OB-SERVER: An Approach for Query Processing in Global Information Systems Based on Interoperation Across Pre-Existing Ontologies. *Distributed and Parallel Databases*, 8(2):223–271, April 2000. doi: 10.1023/A:1008741824956.
- Jose Mora and Oscar Corcho. Towards a systematic benchmarking of ontology-based query rewriting systems. In *The 12th International Semantic Web Conference (ISWC2013)*, pages 369–384, Sydney, Australia, 2013a.
- Jose Mora and Oscar Corcho. Engineering optimisations in query rewriting for OBDA. In *Proceedings of the 9th International Conference on Semantic*

## REFERENCES

---

- Systems (I-SEMANTICS '13)*, pages 41–48, Graz, Austria, September 2013b. ACM. ISBN 978-1-4503-1972-0. doi: 10.1145/2506182.2506188.
- Jose Mora, Riccardo Rosati, and Oscar Corcho. kyrie2: Query Rewriting under Extensional Constraints in  $\mathcal{ELHIO}$ . In *The Semantic Web – ISWC 2014*, number 8796 in Lecture Notes in Computer Science, pages 568–583. Springer International Publishing, October 2014. ISBN 978-3-319-11963-2 978-3-319-11964-9.
- Gabor Nagypal. History ontology building: The technical view. In *Proceedings of the XVI international conference of the Association for History and Computing*, pages 207–214. Royal Netherlands Academy of Arts and Sciences, Amsterdam, 2005.
- Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering, 1995*, pages 251–260, 1995. doi: 10.1109/ICDE.1995.380386.
- Norman W. Paton, Carole Anne Goble, and Sean Bechhofer. Knowledge based information integration systems. *Information and Software Technology*, 42: 299–312, April 2000. doi: 10.1016/S0950-5849(99)00090-7.
- Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):1–45, 2009. doi: 10.1145/1567274.1567278.
- Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. Efficient Query Answering for OWL 2. In *Lecture Notes in Computer Science*, volume 5823, pages 489–504. Springer, 2009.
- Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Tractable query answering and rewriting under description logic constraints. *Journal of Applied Logic*, 8 (2):186–209, June 2010. ISSN 1570-8683. doi: 10.1016/j.jal.2009.09.004.
- Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19–20, Sup-



## REFERENCES

---

- plement 1:261–320, May 1994. ISSN 0743-1066. doi: 10.1016/0743-1066(94)90028-0.
- Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking Data to Ontologies. In *Journal on Data Semantics*, volume 10 of *Lecture Notes in Computer Science*, pages 133–173. Springer, April 2008.
- Axel Polleres. From SPARQL to Rules (and Back). In *Proceedings of the 16th International Conference on World Wide Web, WWW '07*, pages 787–796, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242679.
- John D. Prange. Evaluation Driven Research: The Foundation of the TIPSTER Text Program. In *Proceedings of a Workshop on Held at Vienna, Virginia: May 6-8, 1996*, TIPSTER '96, pages 13–22, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics. doi: 10.3115/1119018.1119022.
- Freddy Priyatna, Carlos Buil-Aranda, and Oscar Corcho. Applying SPARQL-DQP for federated SPARQL querying over Google Fusion Tables. *ESWC2013 Demo*, 2013.
- Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. In *The Semantic Web: Research and Applications*, number 5021 in *Lecture Notes in Computer Science*, pages 524–538. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-68233-2 978-3-540-68234-9.
- John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, January 1965. ISSN 0004-5411. doi: 10.1145/321250.321253.
- Mariano Rodríguez-Muro and Diego Calvanese. Dependencies: Making ontology based data access work in practice. *Proc. of the 5th Alberto Mendelzon Int. Workshop on Foundations of Data Management*, 2011.
- Mariano Rodríguez-Muro and Diego Calvanese. Quest, an OWL 2 QL reasoner for ontology-based data access. *OWLED 2012*, 2012.

## REFERENCES

---

- Mariano Rodríguez-Muro, Lina Lubyte, and Diego Calvanese. Realizing Ontology Based Data Access: A plug-in for Protégé. In *IEEE 24th International Conference on Data Engineering Workshop, 2008. ICDEW 2008*, pages 286–289, 2008. doi: 10.1109/ICDEW.2008.4498333.
- Jeremy Rogers and Alan Rector. The GALEN ontology. *Medical Informatics Europe (MIE 96)*, pages 174–178, 1996.
- Riccardo Rosati. Prexto: Query Rewriting under Extensional Constraints in DL-Lite. In *The Semantic Web: Research and Applications*, volume 7295 of *Lecture Notes in Computer Science*, pages 360–374. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-30283-1.
- Riccardo Rosati and Alessandro Almatelli. Improving Query Answering over DL-Lite Ontologies. In *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*. AAAI Press, 2010.
- Marie-Christine Rousset and Chantal Reynaud. Pictel and Xyleme: Two Illustrative Information Integration Agents. In *Intelligent Information Agents*, pages 50–78. Springer-Verlag, 2003.
- Juan F. Sequeda and Daniel P. Miranker. Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 22:19–39, October 2013. ISSN 1570-8268. doi: 10.1016/j.websem.2013.08.002.
- Giorgos Stamou, Despoina Trivela, and Alexandros Chortaras. Progressive Semantic Query Answering. In *The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, page 112, 2010.
- Snehal Thakkar, Craig A. Knoblock, Jose Luis Ambite, and Cyrus Shahabi. Dynamically Composing Web Services from On-line Sources. In *In Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, 2002.
- Tassos Venetis, Giorgos Stoilos, and Giorgos Stamou. Query rewriting under query extensions for OWL 2 QL ontologies. In *The 7th International Workshop*

## REFERENCES

---

*on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, page 59, 2011.

Luis Manuel Vilches-Blázquez, Miguel Ángel Bernabé-Poveda, Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Antonio F. Rodríguez-Pascual. Towntology & hydrOntology: Relationship between Urban and Hydrographic Features in the Geographic Information Domain. *Ontologies for Urban Development*, 61:73–84, 2007.

Holger Wache, Thomas Vögele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hubner. Ontology-based integration of information-a survey of existing approaches. In *Workshop: Ontologies and Information Sharing*, volume 2001, pages 108–117, 2001.