UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI **MATEMATICA E INFORMATICA**

Dottorato di Ricerca in
MATEMATICA E INFORMATICA

CICLO XXXIII

Large-scale ontology-mediated query answering over OWL 2 RL ontologies

Settore Scientifico Disciplinare: INF/01 Informatica

Coordinatore: Ch.mo Prof. Gianluigi Greco

Supervisore: Prof. Ma[illegible]o Manna

Dottorando: Dott. Alessio Fiorentino

A mia madre, mio padre,

Gianluca e Andrea.

Acknowledgments

First of all, I would like to thank my supervisor Professor Marco Manna. His knowledge and his passion for research have inspired me throughout the course of this journey, and I just consider myself honored and lucky to have been one of his students. Among all of the people I've worked with, I would like to thank Professor Simona Perri and Doctor Jessica Zangari. Their patient guidance in some of my works has been of great help to me. I also thank Professor Gianluigi Greco, Coordinator of the PhD program in Mathematics and Computer Science, and Professor Nicola Leone, whose ideas and insights have contributed to the development of my research activity. Finally, I would like to thank the reviewers of this thesis work for the valuable suggestions that have certainly improved its contents.

Abstract

Ontology-mediated query answering (OMQA) is an emerging paradigm at the basis of many semantic-centric applications. In this setting, a conjunctive query has to be evaluated against a *logical theory* (*knowledge base*) consisting of an extensional *database* paired with an *ontology*, which provides a semantic conceptual view of the data. Among the formalisms that are capable to express such a conceptual layer, the *Web Ontology Language* OWL is certainly the most popular one.

Reasoning over OWL is a very expensive task, in general. For that reason, expressive yet decidable fragments of OWL have been identified. Among them, we focus on OWL 2 RL, which offers a rich variety of semantic constructors, apart from supporting all RDFS datatypes. Although popular Web resources—such as DBpedia—fall in OWL 2 RL, only a few systems have been designed and implemented for this fragment. None of them, however, fully satisfy all the following desiderata: (*i*) being freely available and regularly maintained; (*ii*) supporting SPARQL queries; (*iii*) properly applying the *sameAs* property without adopting the *unique name assumption*; (*iv*) dealing with concrete datatypes.

This thesis aims to provide a contribution in this setting. Primarily, we present *DaRLing:* an open-source Datalog rewriter for OWL 2 RL ontological reasoning under SPARQL queries. We describe its architecture, the rewriting strategies it implements, and the result of an experimental evaluation that demonstrates its practical applicability. Then, to reduce memory consumption and possibly optimize execution times of Datalog queries over large databases, we introduce novel techniques to determine an optimal indexing schema together with suitable body-orderings for Datalog rules, based on the concept of *optimal evaluation plan*. The ASP encoding of a planner for the computation of such plans is provided and explained in detail. The new approach is then compared with the standard execution plans implemented in stat-of-the-art Datalog systems over widely used ontological benchmarks.

Sommario

L'*ontology-mediated query answering* (OMQA) è un paradigma emergente alla base di molte applicazioni semantiche. In questo contesto, una query congiuntiva deve essere valutata su una *base di conoscenza*, i.e., un *database* estensionale insieme ad un'*ontologia*, che fornisce una vista concettuale semantica dei dati. Tra i formalismi in grado di esprimere tale livello concettuale, il *Web Ontology Language* OWL rappresenta di certo il più popolare.

In generale, il reasoning su OWL rappresenta un compito molto costoso e per tale motivo ne sono stati identificati alcuni frammenti (profili) decidibili. Tra questi, ci focalizziamo sul profilo OWL 2 RL che, oltre a supportare tutti gli RDFS datatypes, offre una ricca varietà di costruttori semantici. Nonostante diverse ontologie del web—come DBpedia—ricadono in OWL 2 RL, pochi sono i sistemi ideati ed implementati per questo profilo e nessuno tra questi possiede tutti i seguenti requisiti: (*i*) accessibilità gratuita e mantenimento regolare; (*ii*) supporto del query answering tramite SPARQL; (*iii*) applicazione della proprietà *sameAs* in assenza dell'*assunzione di nome unico* (UNA); (*iv*) trattazione dei *datatype*.

Lo scopo di questa tesi è quello di fornire un contributo in questo contesto. In primis presentiamo *DaRLing*: un riscrittore Datalog open-source per il reasoning su ontologie OWL 2 RL e query SPARQL. Ne descriviamo l'architettura, le strategie di riscrittura e i risultati di una valutazione sperimentale che ne dimostrano l'applicabilità. Poi, allo scopo di ridurre il consumo di memoria ed ottimizzare i tempi di esecuzione di queries Datalog su databases di grandi dimensioni, introduciamo una nuova tecnica per determinare uno schema di indicizzazione insieme ad un ordinamento ottimale per regole Datalog, basata sul concetto di *piano di valutazione ottimale*. Forniamo inoltre il codice ASP di un planner architettato per il calcolo di tali piani. Questo nuovo approccio è infine comparato con i piani di esecuzione standard implementati nei più moderni sistemi Datalog su benchmark ontologici di uso comune.

Contents

Abstract v

Sommario vii

Contents ix

List of Figures xiii

List of Tables xv

1 Introduction 17

1.1 Context and state-of-the-art 17

1.2 Motivation and objectives 18

1.3 Challenges and contribution 20

1.4 Structure of the Thesis 22

I Preliminary notions and notation 24

2 Answer Set Programming 25

2.1 Syntax 25

2.2 Answer set semantics 30

2.3 GCO programming paradigm 36

2.4 Datalog 39

3 Description Logics and OWL **45**

3.1 OWL 2 profiles 45

3.2 OWL 2 RL 46

3.3 Ontology-mediated query answering 50

II *DaRLing* rewriter **53**

4 Rewriting techniques **55**

4.1 System overview 56

4.2 From OWL 2 RL to Datalog 57

4.3 Handling `owl:sameAs` via Datalog 64

5 Experimental evaluation **75**

5.1 Set-up 75

5.2 Quality 77

5.3 Scalability 78

5.4 Discussion 80

III Evaluation planner **82**

6 Planning techniques **83**

6.1 Admissible plans 84

6.2 Preferences 91

7 ASP-based implementation **95**

7.1 Data model 95

7.2 Guessing part 98

7.3 Checking part 98
7.4 Optimization part 100

8 Experimental evaluation **103**
8.1 Setting 103
8.2 Planner customization 105
8.3 Discussion 107

IV Conclusion **112**

9 Related work **113**

10 Combining the *DaRLing* rewriter and the planner **117**

11 Discussion and future work **123**

Bibliography **127**

List of Figures

2-1 Dependency graph of a non stratified program 40

2-2 Dependency graph of program P from Example 2.4.2 42

3-1 The OWL 2 profiles 46

3-2 OMQA via Datalog rewriting 52

4-1 *DaRLing*'s Architecture 56

4-2 Dependency graph of program $P^N_{\sim}$ 68

4-3 `owl:sameAs` graph from Example 4.3.3 71

6-1 Hypergraph associated to a Datalog rule 87

8-1 Plots of the average running time and memory usage of I-DLV with and without planner on LUBM 108

8-2 Plots of the average running time and memory usage of I-DLV with and without planner on LUBM-LUTZ 109

10-1 Plots of the average running time and memory usage of I-DLV with and without planner on *DaRLing*'s outputs over LUBM 120

List of Tables

1.1 Main tools supporting OMQA over OWL 2 RL ontologies 19

3.1 Axioms of the DLs underlying OWL 2 47

4.1 Datalog translation of concept inclusions in normalized form 60

5.1 Average running times of I-DLV executions on the rewritings generated by *DaRLing* and Clipper on LUBM, Adolena, Stock Exchange and Vicodì . . . 77

5.2 Costs of the `owl:sameAs`-cliques materialization on DBpedia 79

5.3 Performance of I-DLV executions on *DaRLing* rewritings with `owl:sameAs` management mode on DBpedia 80

8.1 Planner performance on LUBM and LUBM-LUTZ 106

8.2 Planner performance on Stock Exchange and Vicodì 107

8.3 Memory peaks and sum of execution times of I-DLV execution (with and without planner) on LUBM, LUBM-LUTZ, Stock Exchange and Vicodì . . 110

10.1 Planner performance over *DaRLing* rewritings on LUBM and Stock Exchange 119

10.2 Memory peaks and sum of execution times of I-DLV execution (with and without planner) over *DaRLing* rewritings on LUBM and Stock Exchange . 121

Chapter 1

Introduction

In this chapter we introduce the context and describe the main motivations and challenges that inspired this work. We provide a summary of the contributions and finally outline the structure of the thesis.

1.1 Context and state-of-the-art

Ontology-mediated query answering (OMQA) is an emerging paradigm at the basis of many semantic-centric applications [17, 18, 60]. In this setting, a classical data source is reinterpreted via an ontology, which provides a semantic conceptual view of the data. As a direct and positive effect, the knowledge provided by the ontology can be used to improve query answering.

Among the formalisms that are capable to express such a conceptual layer, the *Web Ontology Language* (OWL) is certainly the most popular one [67, 70]. But an ontology-mediated query (OMQ) has also a second component other than the ontology: the actual query that specifies, in a semantic way and via the ontological vocabulary, which part of the data one is interested in. The most suitable formalism used to specify a query that complements an OWL ontology is definitely the *SPARQL Protocol and RDF Query*

Language (SPARQL), representing—as for OWL—a W3C standard [42].

A number of effective practical approaches proposed in the literature perform OMQA via rewriting the ontology and the query into an equivalent Datalog program. Formally, given a dataset (ABox) $\mathcal{A}$, an OWL ontology (TBox) $\mathcal{T}$ and a SPARQL query $q(\mathbf{x})$, a Datalog program P is constructed with an output predicate *ans* of arity $|\mathbf{x}|$ such that, for each $|\mathbf{x}|$-tuple $\mathbf{t}$ of domain constants, $\mathcal{A} \cup \mathcal{T} \models q(\mathbf{t})$ *if and only if* the atom $ans(\mathbf{t})$ can be derived via P.

Over the past decade, several systems implementing rewriting algorithms have been designed, even for the same fragment of OWL. They may differ, in addition to the techniques, also in the size and time spent to process the rewritings. Among these, concerning the QL profile, Presto [65] produces Datalog rewritings of polynomial size, whereas QuOnto [2] and Requiem [62] produce a union of conjunctive queries (i.e., a set of Datalog rules having predicate *ans* in the head) of exponential size in the worst case. For both the RL and EL fragments, we recall Orel [48] and DReW [72]; whereas OwlOntDB [32] and RDFox [59] are OMQA systems ad-hoc for the RL profile. Concerning more expressive ontologies, Clipper [30] and OWL2DLV [5] are reasoners for conjunctive query answering over Horn-$\mathcal{SHIQ}$ description logics; Graal [15] and VLog [27] support features such as the *skolem* and the *restricted (standard) chase* for reasoning over existential rules. (A deeper comparison and discussion about the state-of-the-art tools is reported in Chapter 9.)

1.2 Motivation and objectives

OWL is a very powerful formalism. But its unrestricted usage makes reasoning undecidable already in case of very simple tasks such as fact entailment. Hence, expressive yet decidable fragments have been identified. Among them, we focus here on the one called OWL 2 RL [57]. From the knowledge representation point of view, OWL 2 RL enables scalable reasoning without sacrificing too much the expressiveness. Indeed, it supports all RDFS datatypes and provides a rich variety of semantic constructors, such as: *inverseOf*,

Tool	License	Latest release	Query language	*sameAs*	Datatypes
Clipper	Free	Dec 2015	SPARQL-BGP	under UNA	No
DReW	Free	Mar 2013	SPARQL-BGP	No	No
Orel	Free	Feb 2010	ground queries	No	No
OWL2DLV	Commercial	Jun 2019	SPARQL-BGP	under UNA	Yes
OwlOntDB	-	-	SPARQL-DL$_E$	under UNA	No
RDFox	Commercial	Jun 2021	SPARQL 1.1	Yes	Yes
DaRLing	Free	Oct 2021	SPARQL-BGP	Yes	Yes

Table 1.1: Main tools supporting OMQA over OWL 2 RL ontologies.

transitiveProperty, *reflexiveProperty*, *equivalentClass*, *disjointWith*, *unionOf*, *minCardinality*, *allValuesFrom*, *someValuesFrom*, and *sameAs*—among others. But the simple fact of allowing *someValuesFrom* only in the left-hand-side of an axiom guarantees that conjunctive query answering can be performed in polynomial time in data complexity (when the OMQ is considered fixed) and in nondeterministic polynomial time in the general case (the latter being exactly the same computational complexity of evaluating a single conjunctive query over a relational database).

Although a number of important Web semantic resources—such as DBpedia[1] and FOAF[2]—trivially fall in OWL 2 RL, only a few systems have been designed and implemented in this setting. None of them, however, fully satisfy all the following desiderata:

(i) being freely available and regularly maintained;

(ii) supporting ontology-mediated query answering;

(iii) properly applying the *sameAs* OWL property without adopting the *unique name assumption* (UNA);

(iv) dealing with concrete *datatypes*.

[1]See https://wiki.dbpedia.org/
[2]See http://www.foaf-project.org/

Table 1.1 reports the main tools supporting or natively implementing ontology-mediated query answering over knowledge bases that fall in the RL profile of OWL 2, or beyond. Concerning the query language, apart from Orel, all the tools support SPARQL patterns: SPARQL 1.1, SPARQL-BGP [42], and SPARQL-DL$_E$ [66]. Finally, the row of OwlOntDB contains some missing value because the system is currently not available. Hence, none of the existing systems fully meet conditions (i)-(iv) above.

1.3 Challenges and contribution

Driven by the lack of effective tools in this domain, we conceived: (i) the open-source system[3] *DaRLing* [34] for preforming OMQA via Datalog over the RL profile of OWL; and (ii) an ASP-based *planner* [7, 33] for optimizing the evaluation of Datalog queries in large-scale scenarios, also available online.[4]

Among the most arduous challenges in the design of *DaRLing* there is certainly the management of the *sameAs*—a property used by many OWL 2 ontologies to declare equalities between resources. As said, the rewriting techniques aim to express inference tasks for OWL in terms of inference tasks for Datalog. However, as well as most logic programming approaches, Datalog works under the *Unique Name Assumption* (UNA), i.e., presumes that different names represent different objects of the world. Preserving the *sameAs* semantics in Datalog programs deriving from rewriting is therefore equivalent to enabling matches between equivalent but syntactically distinct individuals. In this setting, an expensive work (both in terms of time and memory consumption) is represented by the materialization of the *sameAs* transitive closure, due to the enormous extension size that the latter typically assume. To accomplish this task, we devised a Datalog encoding that allows to compute the *sameAs*-cliques in a "non-explicit" way, that is, avoiding the transitive rule directly over the *sameAs* predicate.

[3] See https://demacs-unical.github.io/DaRLing/.
[4] See https://www.mat.unical.it/perri/iclp2019.zip.

Another challenge that motivated this thesis work is performing OMQA via Datalog in large-scale contexts. Indeed, with the growing availability of large databases, an efficient yet memory-saving evaluation of queries is arousing a renewed interest among researchers and industry experts [11]. Typically, classical Datalog reasoners adopt sophisticated internal policies to speed-up the computation trying to limit the memory consumption. However, when the amount of data exceeds a certain size, these policies may result inadequate. In this scenario, to reduce memory consumption and possibly optimize execution times, we propose novel techniques to determine an optimal indexing schema for the underlying database together with suitable body-orderings for the Datalog rules.

Our contribution can be therefore summarized as follows:

- We design *DaRLing*, a freely available Datalog rewriter for OWL 2 RL ontological reasoning under SPARQL queries.
- We face the problem of rewriting OWL 2 RL ontologies by readjusting and optimizing a well-known technique [44] used for the normalization of Horn-$\mathcal{SHIQ}$ description logics.
- We implement the problem of managing the *sameAs* property by efficiently computing its transitive closure via Datalog, and enabling the matches of elements belonging to the same equivalence class (*sameAs*-clique).
- We conduct an experimental evaluation over popular ontological benchmarks widely used for testing both capabilities and performance of OMQA systems. The results confirm that: (*i*) over synthetic OWL 2 RL benchmarks, *DaRLing*'s output is comparable with the one produced by existing tools in terms of both number of produced rules and quality of the rewriting; and (*ii*) over real-world OWL 2 RL knowledge bases, *DaRLing*'s rewriting strategy enables scalable query answering even in case the UNA is not a viable option.

- Given a Datalog program P, a database D and some domain properties, we define the notion of *evaluation plan*, which consists of an indexing schema for D together with a suitable body-ordering for each rule of P. Moreover, to target "optimal" plans among all admissible ones, we identify a number of additional options, the combination of which induces different preference orderings among all plans.
- We encode the problem of finding an optimal evaluation plan in ASP, by making use of choice-rules, strong constraints, weak constraints, aggregates and negation.
- We implement optimal plans by adding annotations [23] to the original Datalog program P. The annotated program will be the actual input for I-DLV. In this way, I-DLV execution is forced to follow the plan without the need for any internal change to the system. Nonetheless, optimal plans are sufficiently general to be implemented natively also in different Datalog engines that do not benefit from features like annotations.
- We design a setting in the context of ontological reasoning to test our evaluation planner with the aim of minimizing the memory consumption without paying in efficiency.
- We compare performance in terms of time and memory usage of DLV when the classical computation is performed, and when the computation is driven by the planner. The results confirm that our plans improve the computation with a general gain in both time and space.

1.4 Structure of the Thesis

This document is structured in 4 parts.

- In Part I we provide an overview of all the background needed to understand the next parts. In Chapter 2 we describe the ASP language and the *Guess-Check-Optimize*

(GCO) programming paradigm. We introduce the Datalog language and some of its extensions that are used as target languages in the translation of OWL ontologies. In Chapter 3 we present the OWL Web Ontology Language together with the syntax and semantics of the description logic underpinning its RL profile. Finally, we introduce the OMQA problem and formally define the problem of performing OMQA via Datalog rewriting.

- Part II represents the main part of the thesis. Here we introduce *DaRLing* [34], a freely available Datalog rewriter for OWL 2 RL ontological reasoning under SPARQL queries. In particular, in Chapter 4 we introduce its architecture and describe the rewriting strategies it implements together with some implementation details, whereas in Chapter 5 we show the result of an experimental evaluation that demonstrates its practical applicability.

- In Part III we formalize the concept of *optimal evaluation plan* for an efficient evaluation of Datalog queries over large databases (Chapter 6). In Chapter 7, the ASP encoding of a planner for the computation of such plans is provided and explained in detail. The new approach is then compared with the standard execution plans implemented in DLV over widely used ontological benchmarks (Chapter 8).

- Part IV is divided into three chapter. In detail, in Chapter 9 we address an in-depth discussion on related work; in Chapter 10 we include the results of an experimental evaluation aimed at testing the behavior of the planner when it takes *DaRLing* rewritings as input; finally, in Chapter 11 we draw some conclusions and highlight future work.

Part of the work presented in this thesis appears in [6], [7], [33] and [34].

Part I

Preliminary notions and notation

Chapter 2

Answer Set Programming

Answer Set Programming (ASP) [19, 31, 54] is a declarative programming paradigm oriented towards "difficult" (primarily NP-hard) search problems, based on the *answer set semantics* of logic programming [37]. ASP supports the use of nonmonotonic reasoning in knowledge-representation and is particularly useful in knowledge-intensive applications. The standard input language for ASP systems is referred to as ASP-Core-2 [21]. The syntax and semantics of the ASP language are introduced below.

2.1 Syntax

Terms are either *constants*, *variables* or *arithmetic terms*. Constants can be either integers, strings starting with some lowercase letter or quoted strings. Variables are denoted by strings starting with some uppercase letter. Arithmetic terms have form $-(t)$, $(t + u)$, $(t - u)$, $(t * u)$ or (t/u) for terms t and u.

Atoms are either *predicate atoms*, *built-in atoms* or *aggregate atoms*. If $t_1, \ldots, t_k$ are terms and p is a *predicate symbol* of arity $k \geq 0$, then $p(t_1, \ldots, t_k)$ is a predicate atom. If a is a predicate atom, then a and *not* a are called *classical literals*. A built-in atom has form $t \gtrless u$, where t and u are terms, and $\gtrless \in \{\text{“}<\text{”}, \text{“}\leq\text{”}, \text{“}>\text{”}, \text{“}\geq\text{”}, \text{“}=\text{”}, \text{“}\neq\text{”}\}$. If

l is a classical literal or a built-in atom, then l is called *naf-literal*.

An *aggregate element* has form

$$t_1, \ldots, t_m : l_1, \ldots, l_n$$

where $t_1, \ldots, t_m$ are terms, $l_1, \ldots, l_n$ are naf-literals and $n, m \geq 0$. An aggregate atom has form

$$\#\text{aggr}\{E\} \gtrless t$$

where $\#\text{aggr} \in \{\text{“}\#\min\text{”}, \text{“}\#\max\text{”}, \text{“\#count”}, \text{“\#sum”}\}$ is an *aggregate function name*, E is a (possibly infinite) collection of aggregate elements syntactically separated by “;”, t is a term and $\gtrless$ is one of the (in)equality symbols as above. For an aggregate atom a, expressions a and *not* a are *aggregate literals*.

Example 2.1.1. *To give an idea of the meaning of the aggregated elements and their use, consider a predicate name of arity 3 whose arguments represent the identifier, the salary and the amount of the monthly working hours of the* employees *of a certain company. Then consider the following aggregates:*

$$\#max\{\, WorkingHours : employee(Id,Salary,WorkingHours)\},$$
$$\#count\{Id : employee(Id,Salary,WorkingHours), Salary > 2000\}.$$

It is easy to guess what these elements are meant to describe. The first one represents the maximum between the number of working hours on all employees, whereas the second one counts the number of distinct employees whose salary is greater than $2000. □

An *ASP rule* r is of the form

$$h_1 \mid \cdots \mid h_m \leftarrow b_1, \ldots, b_n. \tag{2.1}$$

where $m \geq 0$, $n \geq 0$; $h_1, \ldots, h_m$ are predicate atoms and $b_1, \ldots, b_m$ are literals. The set $H(r) = \{h_1, \ldots, h_m\}$ is the *head* of r, whereas $B(r) = \{b_1, \ldots, b_n\}$ is called *body* of r. If $H(r) = \emptyset$ then r is a *(strong) constraint*; if $B(r) = \emptyset$ and $|H(r)| = 1$ then r is a *fact*.

A predicate is defined by a rule r if it occurs in $H(r)$. Predicates defined only by facts are *EDB (Extensional DataBase) predicates*, the remaining are *IDB (Intensional DataBase) predicates* [1]. The set of all facts in P is denoted by $Facts(P)$; the set of instances of all *EDB* predicates in P is denoted by $EDB(P)$.

A *weak constraint* has form

$$:\sim b_1, \ldots, b_n.\ [w@l,\ t_1, \ldots, t_m] \tag{2.2}$$

where $b_1, \ldots, b_n$ $(n \geq 0)$ are literals, $t_1, \ldots, t_m$ $(m \geq 0)$ are terms, w and l are terms standing for a *weight* and a *level*.

A *query* has form $q?$, where q is a predicate atom.

Definition 2.1.1 (ASP program)**.** *An* ASP program *is a finite set of rules and weak constraints, possibly complemented by a single query.*

Example 2.1.2. *The rules*

$$\begin{aligned}
r_1 &: \quad inClique(V) \mid outClique(V) \leftarrow vertex(V).\\
r_2 &: \quad \leftarrow inClique(V_1),\ inClique(V_2),\ V_1 \neq V_2,\ not\ edge(V_1, V_2).\\
r_3 &: \quad :\sim outClique(V).\ [1@1,\ V]
\end{aligned}$$

represent an ASP program P consisting of a disjunctive rule (r_1), a strong constraint (r_2) and a weak constraint (r_3). Given as input an undirected graph whose vertex set and edges are encoded by unary and binary predicates respectively, P detects the maximum cliques (i.e., sets of vertices of maximum cardinality such that, for each pair of vertices, there is an edge connecting them). Intuitively r_1 guesses if a given vertex belongs or not to the

clique, r_2 prunes the solutions in which there exist distinct vertices not connected by any edge, whereas r_3 filters out the cliques with the maximum number of elements ("penalizing" a clique for each external vertex). □

A program (a rule, a weak constraint, a literal, a term) is said to be *ground* if it contains no variables. A variable is *global* in a rule, weak constraint or query r if it appears outside of aggregate elements in r.

Definition 2.1.2 (Safety)**.** *Let V be a set of variables, a variable $v \in V$ is said to be* bound *by a set of literals $\{b_1, \ldots, b_n\}$ if v occurs outside of arithmetic terms in some b_i for $1 \leq i \leq n$ such that b_i is*

- *a (non-negated) predicate atom, or*
- *a built-in atom $t_1 = t_2$ such that each variable of V occurring in one of the terms t_1 and t_2 is bound by $\{b_1, \ldots, b_n\} \setminus \{b_i\}$, or*
- *an aggregate atom $\#aggr\{E\} = t$ such that each global variable of V occurring in E is bound by $\{b_1, \ldots, b_n\} \setminus \{b_i\}$.*

We say that the entire set V is bound by $\{b_1, \ldots, b_n\}$ if each v in V is bound by $\{b_1, \ldots, b_n\}$. A rule r of the form 2.1 (or a weak constraint of the form 2.2) is safe *if the following conditions are satisfied:*

(i) the set V of global variables in r is bound by $\{b_1, \ldots, b_n\}$;

(ii) for each aggregate element E of the form $t_1, \ldots, t_k : l_1, \ldots, l_m$ in r, being W the set of variables occurring in E, the set $W \setminus V$ of local variables is bound by $\{l_1, \ldots, l_m\}$.

Example 2.1.3. *Consider the rule r specified below:*

$$p(X, Y) \leftarrow q(X),\ \#sum\{X, Z : s(X, W),\ Z = X + W\} = Y.$$

It is not difficult to see that r is safe. Indeed, the set $\{X, Y\}$ of global variables is bound by the body $B(r)$ of r, and the set $\{W, Z\}$ of local variables of the aggregate atom of r is bound by $\{s(X, W),\ Z = X + W\}$. Instead the rule

$$p(X, Y) \leftarrow q(X),\ \#sum\{X, Z : s(X, W),\ Z - X = W\} = Y.$$

is not safe because the local variable Z is not bound by $\{s(X, W),\ Z - X = W\}$. □

From now on we refer to an ASP program as a program whose all rules and weak constraints are safe.

2.1.1 Syntactic shortcuts

In the following we illustrate some syntactic shortcuts that will be used in this document.

Anonymous Variables. An *anonymous variable* in a rule, weak constraint or query is denoted by "_" (character underscore) and stands for a fresh variable in the respective context. Different occurrences of anonymous variables represent distinct variables.

Example 2.1.4. *The rule*

$$a(X) \leftarrow b(Y, X),\ c(X, Z).$$

can be equivalently written

$$a(X) \leftarrow b(\_, X),\ c(X, \_).$$

using anonymous variables. □

Choice Rules. A *choice rule* has form

$$\{a_1 : l_{1,1}, \ldots, l_{1,k_1}; \ldots; a_m : l_{m,1}, \ldots, l_{m,k_m}\} \geqslant u \leftarrow b_1, \ldots, b_n. \tag{2.3}$$

where, for each $i = 1, \ldots, m$, a_i is a predicate atom, $k_i \geq 0$ and $l_{i,j}$ is a naf-literal for each $1 \leq j \leq k_i$; moreover, $\gtrless \in \{\text{“}<\text{”}, \text{“}\leq\text{”}, \text{“}>\text{”}, \text{“}\geq\text{”}, \text{“}=\text{”}, \text{“}\neq\text{”}\}$, u is a term and $b_1, \ldots, b_n$ are literals for $n \geq 0$. The part $\gtrless u$ can optionally be omitted if $\gtrless$ stands for “$\geq$” and $u = 0$.

For each predicate atom $a_i = p_i(t_{i,1}, \ldots, t_{i,h_i})$, let $\widehat{a_i} = \widehat{p_i}(t_{i,1}, \ldots, t_{i,h_i})$, where $\widehat{p_i}$ is a fresh predicate symbol that is uniquely associated with p_i. A choice rule of the form 2.3 is a shortcut to the rules:

$$a_i \mid \widehat{a_i} \leftarrow b_1, \ldots, b_n, l_{i,1}, \ldots, l_{i,k_i}.$$

for each $1 \leq i \leq m$ along with the single constraint

$$\leftarrow b_1, \ldots, b_n, \; not \; \#\text{count}\{a_1 : l_{1,1}, \ldots, l_{1,k_1}; \ldots; a_m : l_{m,1}, \ldots, l_{m,k_m}\} \gtrless u.$$

Example 2.1.5. *The choice rule*

$$\{a(X) : b(X)\} \leq 1 \leftarrow c(X, Y), d(Y).$$

stands for:

$$a(X) \mid \hat{a}(X) \leftarrow c(X, Y), d(Y), b(X).$$
$$\leftarrow c(X, Y), d(Y), \; not \; \#count\{a(X) : b(X)\} \leq 1.$$

where $\hat{a}$ is uniquely associated with a. □

2.2 Answer set semantics

Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all integers and ground terms that can be built combining constants and function symbols appearing in P. The *Herbrand base* of P, denoted by B_P, is the set of all ground predicate atoms

obtainable from the predicates appearing in P by replacing variables with elements from U_P. An *interpretation* I for P is a subset of B_P.

Satisfaction of literals

Classical. Given an interpretation $I \subseteq B_P$, a predicate atom $a \in B_P$ is *true* (resp., *false*) w.r.t. I if $a \in I$ (resp., $a \notin I$); *not* a is *true* (resp., *false*) w.r.t. I if a is false (resp., true) w.r.t. I.

Example 2.2.1. *Let* $I = \{a(2), b(1,2), c(2,1)\}$ *be an interpretation. Literals* $a(2)$ *and not* $a(1)$ *are true w.r.t.* I*, whereas not* $b(1,2)$ *and* $c(2,2)$ *are false w.r.t.* I*.* □

Built-in. Let $\preceq$ be a total order relationship defined on U_P as follows:

- $t \preceq u$ for integers t and u if $t \leq u$;
- $t \preceq u$ for any integer t and any string u;
- $t \preceq u$ for strings t and u if t is lexicographically smaller than or equal to u;

A built-in atom $t \gtrless u$, with $t, u \in U_P$, is *true* w.r.t. an interpretation $I \subseteq B_P$ if one of the following conditions is satisfied: (i) $\gtrless$ is "$\leq$" and $t \preceq u$; (ii) $\gtrless$ is "$\geq$" and $u \preceq t$; (iii) $\gtrless$ is "$<$" and $(t \preceq u) \wedge (u \not\preceq t)$; (iv) $\gtrless$ is "$>$" and $(u \preceq t) \wedge (t \not\preceq u)$; (v) $\gtrless$ is "$=$" and $(t \preceq u) \wedge (u \preceq t)$; (vi) $\gtrless$ is "$\neq$" and $(t \not\preceq u) \vee (u \not\preceq t)$). Otherwise, $t \gtrless u$ is *false* w.r.t. I.

Aggregates. For the satisfaction of aggregate literals, we introduce an *aggregate function* associated with each aggregate function name, which maps a set T of tuples of terms to a term, $+\infty$ or $-\infty$ as follows:

$$\#\min(T) \quad = \begin{cases} -\infty & \text{if } |T| = +\infty \\ \min\{t_1 \mid (t_1, \ldots, t_m) \in T\} & \text{if } 0 < |T| < +\infty \\ +\infty & \text{if } T = \emptyset; \end{cases}$$

$$
\begin{aligned}
\#\max(T) \quad &= \begin{cases} -\infty & \text{if } T = \emptyset \\ \max\{t_1 \mid (t_1, \ldots, t_m) \in T\} & \text{if } 0 < |T| < +\infty \\ +\infty & \text{if } |T| = +\infty; \end{cases} \\
\#\text{count}(T) \quad &= |T|; \\
\#\text{sum}(T) \quad &= \begin{cases} \sum\limits_{\substack{(t_1,\ldots,t_m)\in T; \\ t_1 \text{ is an integer}}} t_1 & \text{if } |\{(t_1, \ldots, t_m) \in T \mid t_1 \in \mathbb{Z} \setminus \{0\}\}| < +\infty \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}
$$

Notice that aggregate functions #min and #max rely on the total order $\preceq$ on U_P for finite non-empty sets. Moreover, in the case of $\#\text{aggr}(T) = \pm\infty$, we adopt the convention that $-\infty \preceq u$ and $u \preceq +\infty$ for each $u \in U_P$. An interpretation $I \subseteq B_P$ maps any collection E of aggregate elements to the following set, denoted $\texttt{eval}(E, I)$, of tuples of ground terms:

$$\texttt{eval}(E, I) = \{(t_1, ..., t_m) \mid t_1, ..., t_m : l_1, ..., l_n \text{ occurs in } E \text{ and } l_1, ..., l_n \text{ are true w.r.t. } I\}$$

A literal $a = \#\text{aggr}(E) \gtrless u$ is *true* (resp., *false*) w.r.t. I if $\#\text{aggr}(\texttt{eval}(E, I)) \gtrless u$ is true (resp., false) according to the corresponding aggregate function and the total order $\preceq$ on U_P (extended with $+\infty$ and $-\infty$ symbols) previously defined; *not* a is *true* (resp., *false*) w.r.t. I if a is false (resp., true) w.r.t. I.

Example 2.2.2. *Let* $E = \{2, 1 : b(1, 2), \textit{not } c(1);\ 2, 2 : b(2, 2), \textit{not } c(2);\ 2, 1 : c(2)\}$ *be a collection of aggregate elements and* $I = \{b(1, 2), b(2, 2), c(2)\}$ *be an interpretation. We have that* $\texttt{eval}(E, I) = \{(2, 1)\}$ *and the literal* $\#sum(E) < 3$ *is true w.r.t.* I *since* $\#sum(\texttt{eval}(E, I)) = \#sum(\{(2, 1)\}) = 2$. □

Answer Sets

The semantics of ASP is given in terms of *answer sets*. Given a program P, we say that a *substitution* is a map σ from a set of variables V to the Herbrand universe U_P. For an object O (rule, weak constraint, query, literal, aggregate element, etc.), we denote by $O\sigma$

the object obtained by replacing each occurrence of a variable $v \in V$ by $\sigma(v)$ in O. Given a rule r in P, a substitution from the set of global variables in r is *global* for r; a substitution from the set of variables in an aggregate element e is *local* for e. The *instantiation* of a collection E of aggregate elements is the set

$$\bigcup_{e \in E} \{e\sigma \mid \sigma \text{ is a local substitution for } e\}.$$

A *ground instance* of a rule, a weak constraint or a query r is obtained by first applying a global substitution σ to r, and then replacing E with its instantiation, for each aggregate atom $\#\text{aggr}E \gtrless u$ appearing in $r\sigma$. The *full instantiation* of a program P, denoted *Ground(P)*, is the set of all ground instances of its rules and weak constraints.

We say that a ground rule

$$h_1 | \ldots | h_m \leftarrow b_1, \ldots, b_n.$$

in *Ground(P)* is *satisfied* w.r.t. an interpretation I if some $h \in \{h_1, \ldots, h_m\}$ is true w.r.t. I when $b_1, \ldots, b_n$ are true w.r.t. I; we say that I is a model of P if every rule in *Ground(P)* is satisfied w.r.t. I. The *Gelfond-Lifschitz reduct* [37] of P w.r.t. I, is the subset P^I of all rules of *Ground(P)* having all body literals true w.r.t. I. A model M of P is an *answer set* for a program P if and only if M is a minimal model (with respect to the subset inclusion relation $\subseteq$) for P^M. The set of all answer sets for P is denoted by $AS(P)$.

Example 2.2.3. *Consider the program P from Example 2.1.2 together with the set F of ground facts:*

$$\begin{array}{llll} vertex(1). & vertex(2). & vertex(3). & vertex(4). \\ edge(1,2). & edge(2,1). & edge(1,3). & edge(3,1). \\ edge(2,3). & edge(3,2). & edge(1,4). & edge(4,1). \end{array}$$

which encode the graph $G = \langle V, E\rangle$ *with* $V = \{1, 2, 3\}$ *and* $E = \{\{1, 2\}, \{1, 3\}\}$. *The answer sets of* $P \cup F$ *are:*

$$AS_0 = F \cup \{outClique(1), outClique(2), outClique(3)\},$$
$$AS_1 = F \cup \{inClique(1), outClique(2), outClique(3)\},$$
$$AS_2 = F \cup \{outClique(1), inClique(2), outClique(3)\},$$
$$AS_3 = F \cup \{outClique(1), outClique(2), inClique(3)\},$$
$$AS_4 = F \cup \{inClique(1), inClique(2), outClique(3)\},$$
$$AS_5 = F \cup \{inClique(1), outClique(2), inClique(3)\}.$$

Note that the correspondence:

$$AS_0 \mapsto \emptyset$$
$$AS_1 \mapsto \{1\}$$
$$AS_2 \mapsto \{2\}$$
$$AS_3 \mapsto \{3\}$$
$$AS_4 \mapsto \{1, 2\}$$
$$AS_5 \mapsto \{1, 3\}$$

is a bijective function between $AS(P \cup F)$ *and the set of all the possible cliques of* G. □

Optimal Answer Sets

Consider the function $\tilde{w}(P, \cdot)$ which for any interpretation I of P acts as follows:

$$\tilde{w}(P, I) := \{(w@l, t_1, \ldots, t_m) \mid \ :\sim b_1, \ldots, b_n.\ [w@l,\ t_1, \ldots, t_m] \text{ occurs in } \textit{Ground(P)}$$

and $b_1, \ldots, b_n$ are true w.r.t. $I\}$.

For any integer l, is defined the quantity

$$W_l^I := \begin{cases} \sum\limits_{\substack{(w@l,t_1,\ldots,t_m)\in\tilde{w}(P,I);\\ w \text{ is an integer}}} w & \text{if } |\{(w@l,t_1,\ldots,t_m)\in\tilde{w}(P,I) \mid w\neq 0\}| < +\infty \\ 0 & \text{otherwise} \end{cases}$$

representing the overall sum of the *weights* at *level* l in $\tilde{w}(P,I)$ w.r.t. I. Given two answer sets $M, M' \in AS(P)$, we say that M *dominates* M' (or, M' is *dominated* by M) if there is an integer $\bar{l}$ such that $W_{\bar{l}}^M < W_{\bar{l}}^{M'}$ and $W_l^M = W_l^{M'}$ for each $l > \bar{l}$. An answer set $M \in AS(P)$ is *optimal* if there is no $M' \in AS(P)$ that dominates M.

Example 2.2.4. *Let P and F as in Example 2.1.2 and Example 2.2.3. Then,*

$$\tilde{w}(P\cup F, AS_0) = \{(1@1,\ 1), (1@1,\ 2), (1@1,\ 3)\},$$
$$\tilde{w}(P\cup F, AS_1) = \{(1@1,\ 2), (1@1,\ 3)\},$$
$$\tilde{w}(P\cup F, AS_2) = \{(1@1,\ 1), (1@1,\ 3)\},$$
$$\tilde{w}(P\cup F, AS_3) = \{(1@1,\ 1), (1@1,\ 2)\},$$
$$\tilde{w}(P\cup F, AS_4) = \{(1@1,\ 3)\},\ \textit{and}$$
$$\tilde{w}(P\cup F, AS_5) = \{(1@1,\ 2)\}.$$

Thus, we get $W_1^{AS_0} = 3$, $W_1^{AS_1} = W_1^{AS_2} = W_1^{AS_3} = 2$ and $W_1^{AS_4} = W_1^{AS_5} = 1$. Therefore, the optimal answer sets are AS_4 and AS_5, which not by chance represent the cliques whose cardinality is 2. □

Queries

A ground query $Q = q?$ of a program P is *true* if $q \in M$ for each answer set $M \in AS(P)$; otherwise, Q is *false*. Given a non-ground query $Q = q(t_1, \ldots, t_n)?$ in a program P, the *set*

of answers to Q is

$$\text{Ans}(Q, P) = \{\sigma \mid \sigma \text{ is a substitution for } Q \text{ such that } Q\sigma \text{ is true}\}.$$

Note that, if $AS(P) = \emptyset$, all ground queries are true and $\text{Ans}(Q, P)$ contains all possible substitutions for Q.

Example 2.2.5. *Consider P and F as in Example 2.1.2 and Example 2.2.3, and the query $Q = inClique(X)?$. Since there is no vertex that belongs to all the cliques, the set of answers $Ans(Q, P \cup F)$ is empty.* □

2.3 GCO programming paradigm

Apart from its high expressive power (ASP captures the complexity class $\Sigma_2^P = \text{NP}^{\text{NP}}$), one of the main reasons for the success of ASP is that it is a *full declarative* language (the ordering of literals and rules is not relevant). A large variety of problems can be encoded using the *Guess-Check-Optimize* (GCO) methodology [53] in a very compact, readable, and elegant way. Given a set of facts F that specifies an instance I of a given problem, a GCO program P of that problem consists of the following three main parts:

- The *guessing part* $\mathcal{G} \subseteq P$ of the program consists of disjunctive rules such that the answer sets of $\mathcal{G} \cup F$ represent "solution candidates" for I. $\mathcal{G}$ defines the search space in which rule applications are branching points.

- The (optional) *checking part* $\mathcal{C} \subseteq P$ of the program prunes illegal branches. $\mathcal{C}$ consists of strong constraints that filter the solution candidates in such a way that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup F$ represent the admissible solutions for the problem instance I.

- The (optional) *optimization part* $\mathcal{O} \subseteq P$ of the program allows to specify the best solutions according to an optimization function by using weak constraints. Indeed, weak constraints implicitly define an objective function $f : AS(\mathcal{G} \cup \mathcal{C} \cup F) \to \mathbb{N}$ mapping

the answer sets of $\mathcal{G} \cup \mathcal{C} \cup F$ to natural numbers (see Subsection 2.2). The semantics of $\mathcal{G} \cup \mathcal{C} \cup F \cup \mathcal{O}$ optimizes f by filtering out those answer sets on which f reaches the minimum value; this way, the optimal (least cost) solutions are computed.

Each layer of the GCO program P may have further auxiliary predicates, defined by stratified rules, for local computations.

Example 2.3.1. *In this example we show an elegant ASP encoding for the* 3-Col *problem, which is known to be a* NP*-complete problem [61]. Let* $G = \langle V, E \rangle$ *be an undirected graph whose vertex set is* $V = \{1, \ldots, n\}$ *and edges are represented by a subset* $E \subseteq \{\{i, j\} \mid i, j \in V, i \neq j\}$*. Given a set* $Col = \{r, g, b\}$ *of three colors, the* 3*-Col problem is to decide whether there exists a* 3-Coloring *map* $\gamma : V \to Col$ *such that* $\gamma(i) \neq \gamma(j)$ *if* $\{i, j\} \in E$ *(i.e., adjacent vertices have different colors). Suppose that* G *is represented by a set of facts* F *using a unary predicate symbol "vertex" for the elements of* V*, and a binary predicate symbol "edge" for the elements of* E*. Then, the following GCO program, combined with* F*, computes all the possible* 3*-Colorings of that graph.*

$$r_1 : \quad color(X, r) \mid color(X, g) \mid color(X, b) \leftarrow vertex(X).$$
$$r_2 : \quad \leftarrow color(X_1, C),\ color(X_2, C),\ edge(X_1, X_2).$$

In particular, the rule r_1 *is the guessing part of the program and expresses that each vertex must either be colored red, green, or blue: due to minimality of the answer sets, a vertex cannot be assigned more than one color. The constraint* r_2 *constitutes checking part of the program and ensures that no pair of adjacent vertices is assigned to the same color. It is easy to see that there is a one-to-one correspondence between the solutions of* 3*-Col and* $AS(F \cup \{r_1, r_2\})$*. In other words,* G *is* 3*-colorable if and only if* $F \cup \{r_1, r_2\}$ *has some answer set.* □

Note that 3-Col is not an optimization problem, therefore there is no optimization part in its ASP encoding. We next see how to use the GCO programming paradigm for

the resolution of the *Travel Salesman Problem* (TSP), another well-known NP-complete problem [61, 64], by means of an ASP encoding whose answer sets correspond to the problem solutions.

Example 2.3.2. *Let $G = \langle N, A, c\rangle$ be a weighted directed graph, where N is a set of nodes, $A \subseteq N \times N$ is a set of arcs and $c : A \to \mathbb{N}^+$ is a* cost function *mapping each arc in A to a positive natural number. Given a node $n_0 \in N$, the TSP asks to find a minimum-cost cycle (closed path) in G starting at n_0 and passing through each node of N exactly once. Suppose again that G is specified by a set of facts F using the unary predicate symbol "node" which represents the set N, the ternary predicate symbol "arc" representing the arcs in A with the respective costs specified by c, and the unary predicate symbol "start" representing the starting node. Then, below is a GCO encoding for the TSP.*

$$r_1: \quad inPath(X,Y,C) \mid outPath(X,Y,C) \leftarrow start(X),\ arc(X,Y,C).$$

$$r_2: \quad inPath(X,Y,C) \mid outPath(X,Y,C) \leftarrow reached(X),\ arc(X,Y,C).$$

$$r_3: \quad reached(X) \leftarrow inPath(\_,X,\_).$$

$$r_4: \quad \leftarrow inPath(X,Y,\_),\ inPath(X,Y_1,\_),\ Y \neq Y_1.$$

$$r_5: \quad \leftarrow inPath(X,Y,\_),\ inPath(X_1,Y,\_),\ X \neq X_1.$$

$$r_6: \quad \leftarrow node(X),\ not\ reached(X).$$

$$r_7: \quad :\sim inPath(X,Y,C).\ [C@1,\ X,Y,C]$$

The guessing part of the program uses the disjunctive rules r_1 and r_2 for the generations of all (nondeterministic) paths in G starting at n_0, and the rule r_3 for the computation of the reached nodes in any path. The checking part consists of the three strong constraints r_4, r_5 and r_6 which select paths (among all the guessed ones) representing cycles passing through each node exactly once. Finally, the optimization part consists of the weak constraint r_7 which picks cycles with lower cost. Thus, it is easy to see that the answer sets of $P \cup F$ are in a one-to-one correspondence with the optimal tours of the input graph. □

2.4 Datalog

In this section we introduce the Datalog language [29] and the notation with which we indicate some of its extensions. As we will see, some fragments of the *Web Ontology Language OWL* can be rewritten in Datalog. Here Datalog programs are seen as a special case of ASP ones and the fixpoint semantics is provided.

Syntax

A *Datalog program* is an ASP program not allowing for:

(*i*) arithmetic terms,

(*ii*) built-in atoms,

(*iii*) aggregate atoms,

(*iv*) negated atoms,

(*v*) weak constraints, and

(*vi*) disjunction in the heads of its rules.

The language obtained from Datalog allowing the negation in the body of the rules is referred as Datalog$^{\neg}$ [36]. In other words, a Datalog$^{\neg}$ program is a set of rules of the form:

$$\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_m, \mathit{not}\ \alpha_{m+1}, \ldots, \mathit{not}\ \alpha_n .$$

where $n \geq m \geq 0$, and each α_i is a predicate atom. We define $B^+(r) = \{\alpha_1, \ldots, \alpha_m\}$ (the *positive body*) and $B^-(r) = \{\mathit{not}\ \alpha_{m+1}, \ldots, \mathit{not}\ \alpha_n\}$ (the *negative body*). If $B(r) = \emptyset$ then r is a *fact*. The set of all the facts of a program P is denoted $\mathit{Facts}(P)$.

The notion of *safety* is inherited from Section 2.1. In particular, we have that a Datalog$^{\neg}$ rule r is *safe* if each variable in r occurs at least once in $B^+(r)$. In the following, only

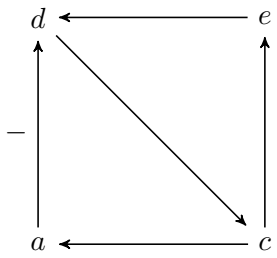

Figure 2-1: Dependency graph of program P from Example 2.4.1.

safe rules are considered, and programs are required to satisfy *stratification of negation*, defined next. With a Datalog$^\neg$ program P we can associate a labeled graph $\mathcal{G}_P$, called *dependency graph*, whose nodes are all the *IDB* predicates (namely, those that appear in the head of the rules of $P \backslash Facts(P)$) of P and (b, a) is an arc if predicate a depends on b (i.e., b appears in the body of a rule where a appears in the head); moreover (b, a) is labeled with the symbol "$-$" if the occurrence of b is negated. A Datalog$^\neg$ program P is said to be *stratified* if no cycle of the dependency graph of P contains a labeled arc. In other words, P is stratified if the negation is not involved in recursion. We indicate with Datalog$^{\neg s}$ the Datalog extension allowing for stratified negation.

Example 2.4.1. *The following program P:*

$$a(X) \leftarrow b(X, Y),\ c(Y).$$
$$c(X) \leftarrow d(X).$$
$$e(X, Y) \leftarrow b(Y, X),\ c(Y).$$
$$d(X) \leftarrow e(X, Y),\ not\ a(X).$$

is not stratified. The dependency graph $\mathcal{G}_P$ is shown in Figure 2-1 and contains a cycle with a label. □

Fixpoint semantics

As a fragment of the ASP language, the Datalog semantics is directly inherited from the ASP semantics described in Section 2.2 using the *model-theoretic* approach. In the following we give the semantics of Datalog$^{\neg s}$ making use of the *operational (fixpoint)* approach.

A substitution σ is a mapping from variables to constants; for a set of literals L, let $L\sigma$ be the set obtained from L by replacing each variable X by $\sigma(X)$. Recall that a *strongly connected component* (SCC) of a directed graph $\mathcal{G}$ is a maximal subgraph of $\mathcal{G}$ in which there exists an oriented path between each pair of nodes belonging to it. Given a program P, let $C_1, \ldots, C_n$ (for some $n \geq 1$) be the SCCs of $\mathcal{G}_P$, sorted in such a way that for every $p \in C_i$ and $p' \in C_j$ with $1 \leq i < j \leq n$, there is no path from p' to p in $\mathcal{G}_P$. Let $\mathtt{sub}_P(C_i)$ denote the subprogram containing all the rules of P whose head predicates belong to C_i. Given $i \in \{1, \ldots, n\}$, the *immediate consequence operator* of P at stage i, denoted T_P^i, is defined as

$$T_P^i(I) := \{H(r)\sigma \mid r \in \mathtt{sub}_P(C_i), B^+(r)\sigma \subseteq I, B^-(r)\sigma \cap I = \emptyset\}$$

for each set of ground atoms I. For $i \in \{1, \ldots, n\}$, consider the sequence:

$$\begin{aligned} A_0^i(I) &:= I \\ A_k^i(I) &:= A_{k-1}^i(I) \cup T_P^i(A_{k-1}^i(I)) \qquad (k = 1, 2, \ldots); \end{aligned}$$

and let $\bar{k} := \min\{k \geq 0 \mid A_k^i(I) \equiv A_{k+1}^i(I)\}$. The *least fixpoint* of T_P^i (w.r.t. I) is defined as

$$T_P^i \Uparrow I := A_{\bar{k}}^i(I).$$

Let $I_0 := \mathit{Facts}(P)$, and $I_i := T_P^i \Uparrow I_{i-1}$, for $i = 1, \ldots, n$. The semantics of P is given by I_n, in the following denoted $TP(P)$.

Example 2.4.2. *Consider a directed graph. Nodes that can not be reached from a source*

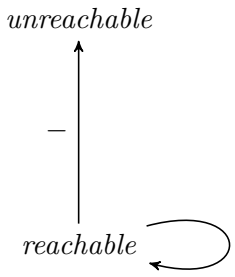

Figure 2-2: Dependency graph of program P from Example 2.4.2.

node are identified by the following program P*:*

$$reachable(Y) \leftarrow source(X),\ arc(X,Y).$$
$$reachable(Y) \leftarrow reachable(X),\ arc(X,Y).$$
$$unreachable(Y) \leftarrow source(X),\ node(Y),\ not\ reachable(X,Y).$$

The dependency graph $\mathcal{G}_P$ *is shown in Figure 2-2. The SCCs of* $\mathcal{G}_P$ *are* $C_1 = \{reachable\}$ *and* $C_2 = \{unreachable\}$*. Given the dataset* D

$$source(1).\quad node(2).\quad node(3).\quad node(4).$$
$$arc(1,2).\quad arc(2,3).\quad arc(4,2).$$

$TP(P \cup D)$ *extends* D *with*

$reachable(2).$	(stage 1, application 1)
$reachable(3).$	(stage 1, application 2)
$unreachable(4).$	(stage 2)

□

Finally, we refer as Datalog$^{\neg_s,\neq}$ the extension of Datalog that allows the use of the stratified negation and built-in atoms. Such an extension represents the target language in the rewriting of our OWL 2 RL ontologies.

Chapter 3

Description Logics and OWL

Description Logics (DLs) are a family of formal knowledge representation languages used to describe and reason about the "concepts" of an application domain [14, 20, 46]. Among the most relevant applications of DLs there is the OWL Web Ontology Language,[1] a knowledge representation language standardised by the World Wide Web Consortium (W3C) and designed to facilitate the development of Semantic Web applications. The current version of the OWL specification[2] is OWL 2 [38], developed by the W3C OWL Working Group. The syntactic elements of OWL 2 are almost analogous to those of a DL, with the main difference that *concepts* and *roles* are called *classes* and *properties* respectively.

3.1 OWL 2 profiles

Reasoning over OWL 2 is a very expensive task: fact entailment (i.e., checking whether an individual is an instance of a concept) is already 2NExpTime-hard, while decidability of conjunctive query answering is even an open problem. To balance expressiveness and scalability, the W3C identified three tractable profiles[3]—OWL 2 EL, OWL 2 QL, and OWL

[1] See http://www.w3.org/TR/owl2-overview/
[2] See https://www.w3.org/TR/owl2-syntax/
[3] See http://www.w3.org/TR/owl2-profiles/

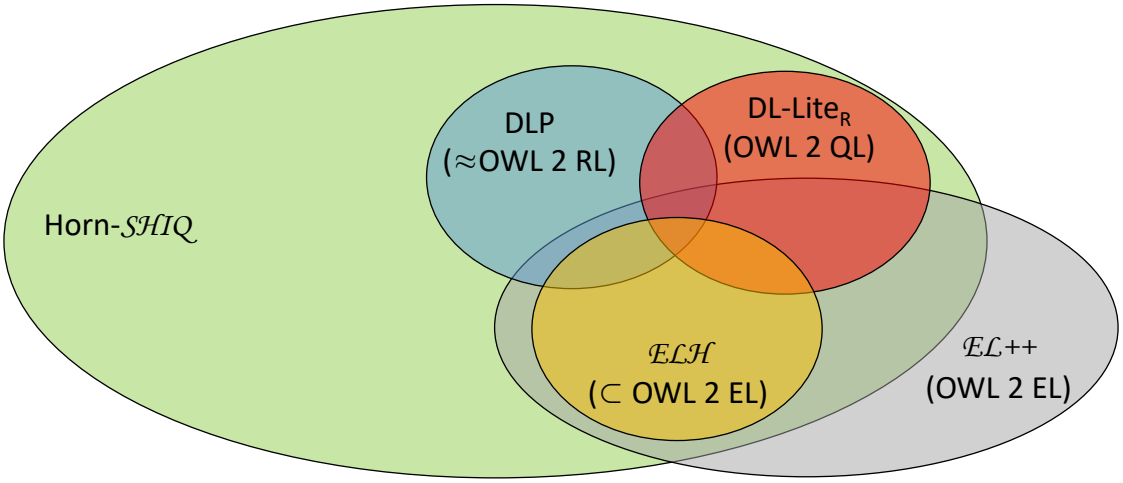

Figure 3-1: The OWL 2 profiles.

2 RL—exhibiting good computational properties: the evaluation of conjunctive queries over Knowledge Bases falling in these fragments is in PTIME in data complexity (where query and TBox are considered fixed) and in PSPACE in combined complexity (nothing is fixed) [69]. Figure 3-1 shows how the three standard OWL 2 profiles are placed within the DLs context, and also how they are related to each other. In particular, we have that $\mathcal{EL}$++ [13], *DL*-Lite$_R$ [65] and *DLP* [40] are the DLs underpinning OWL 2 EL, OWL 2 QL and OWL 2 RL, respectively. Table 3.1 reports those axioms that are at the basis of the OWL 2 profiles, involving just atomic concept and role names. Among these three profiles, OWL 2 RL is the only one that does not admit the usage of existential quantification in the right-hand side of axioms.

3.2 OWL 2 RL

In this thesis we focus on OWL 2 RL, which, from the knowledge representation point of view, enables scalable reasoning without scarifying too much the expressiveness. In the following we provide the notation related to the DL underlying the OWL 2 RL profile and its semantics.

Horn-$\mathcal{SHIQ}$	$\mathcal{EL}$++ (OWL 2 EL)	DL-Lite$_R$ (OWL 2 QL)	DLP ($\approx$ OWL 2 RL)	axioms
✓	✓	✓	✓	$B \sqsubseteq A$
✓	✓		✓	$B_1 \sqcap B_2 \sqsubseteq A$
✓		✓	✓	$B \sqsubseteq \forall R.A$
✓	✓		✓	$\exists R.B \sqsubseteq A$
✓	✓	✓	✓	$\exists R.\top \sqsubseteq A$
✓	✓	✓		$B \sqsubseteq \exists R.A$
✓		✓	✓	$B \sqsubseteq \neg A$
✓			✓	$B \sqsubseteq \leqslant 1\, R.A$
✓	✓	✓	✓	$R \sqsubseteq S$
✓		✓	✓	$R^- \sqsubseteq S$
✓	✓		✓	$R \circ R \sqsubseteq R$
	✓			$R \circ S \sqsubseteq P$
✓		✓	✓	$R \sqsubseteq \neg S$

Table 3.1: Axioms of the DLs underlying OWL 2: A, B, B_1 and B_2 are atomic concepts; R, S and P are role names.

Syntax

Let N_C (*atomic concepts*), N_R (*role names*), and N_I (*individuals*) be mutually disjoint discrete sets. A *role* is either $r \in N_R$ or an inverse role r^- with $r \in N_R$. We denote by R^- the *inverse* of a role R defined by $R^- := r^-$ when $R = r$ and $R^- := r$ when $R = r^-$.

Definition 3.2.1. *The set of* concepts *is the smallest set such that:*

- $\top$, $\bot$, *and every atomic concept* $A \in N_C$ *is a concept;*
- *if* C *and* D *are concepts and* R *is a role, then* $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall R.C$, $\exists R.C$, $\geq nR.C$ *and* $\leq nR.C$, *for* $n \geq 1$, *are concepts.*

Definition 3.2.2. Subconcept *and* Superconcept Expressions *are recursively defined as follows:*

- $\top$ *and every atomic concept* $A \in N_C$ *is both a Subconcept and a Superconcept Expression;*

- *if C and D are Subconcept Expressions and R is a role, then $C \sqcap D$, $C \sqcup D$, $\exists R.C$ and $\geq 1R.C$ are Subconcept Expressions;*

- *if C and D are Superconcept Expressions and R is a role, then $C \sqcap D$, $\neg C$, $\forall R.C$, and $\leq 1R.C$ are Superconcept Expressions.*

An *assertion* is of the form $C(a)$ or $R(a, b)$, where C is a concept, R is a role, and $a, b \in N_I$. A finite set of assertional axioms is called *ABox* (*assertional box*). An expression $C \sqsubseteq D$, where C, D are concepts, is a *concept inclusion (CI)*. An expression $R \sqsubseteq S$, where R, S are roles, is a *role inclusion (RI)*. A *transitivity axiom* is an expression trans(R), where R is a role. A finite set of CIs, RIs and transitivity axioms is called *TBox* (*terminological box*).

Definition 3.2.3. *A* knowledge base *(KB) is any pair $\mathcal{K} = (\mathcal{A}, \mathcal{T})$ where $\mathcal{A}$ is an ABox and $\mathcal{T}$ is a TBox.*

We will generally refer to the elements of an ABox or a TBox by calling them *axioms*.

Definition 3.2.4 (Polarities)**.** Positive *and* negative occurrences *of a concept C in concepts are defined as follows:*

- *C occurs positively in itself;*

- *C occurs positively (resp., negatively) in $\neg D$ or $\leq nS.D$ if C occurs negatively (resp., positively) in D ;*

- *C occurs positively (resp., negatively) in $\exists R.D$, $\forall R.D$ or $\geq nS.D$ if C occurs positively (resp., negatively) in D; and*

- *C occurs positively (resp., negatively) in $D_1 \sqcap D_2$, $D_1 \sqcup D_2$ if C occurs positively (resp., negatively) in D_1 or in D_2 .*

C occurs positively (resp., negatively) in $D_1 \sqsubseteq D_2$ if C occurs positively (resp., negatively) in D_2, or negatively (resp., positively) in D_1; C occurs positively (resp., negatively) in a TBox $\mathcal{T}$ if C occurs positively (resp., negatively) in some axiom of $\mathcal{T}$.

Example 3.2.1. *In the TBox*

$$\texttt{linkedViaTrain} \sqsubseteq \texttt{linked}$$

$$\mathit{trans}(\texttt{linkedViaTrain})$$

$$\texttt{CommutingArea} \sqsubseteq \exists \texttt{linked}.\texttt{Capital}$$

$$\exists \texttt{linked}.\texttt{Capital} \sqsubseteq \texttt{DesirableArea}$$

$$\texttt{Capital} \sqsubseteq \texttt{DesirableArea}$$

we have the following: `Capital` *occurs both positively (in the third axiom) and negatively (in the last two axioms), whereas* `DesirableArea` *occurs positively and* `CommutingArea` *occurs negatively.* □

Definition 3.2.5. *We say that a TBox $\mathcal{T}$ belongs to the OWL 2 RL profile if the following conditions are satisfied:*

- *no concept of the form $C \sqcup D$ occurs positively in $\mathcal{T}$; and*
- *no concept of the form $\neg C$, $\forall R.C$, or $\leq 1 R.C$ occurs negatively in $\mathcal{T}$;*
- *for each CI $C \sqsubseteq D$ it turns out that C is a Subconcept Expression and D is a Superconcept Expression;*

Example 3.2.2. *The TBox of the Example 3.2.1 does not fall in the OWL 2 RL profile. In fact, in* $\texttt{CommutingArea} \sqsubseteq \exists \texttt{linked}.\texttt{Capital}$*, the concept* $\exists \texttt{linked}.\texttt{Capital}$ *is not a Superconcept Expression.* □

Semantics

An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$ (*domain*) and a function $\cdot^{\mathcal{I}}$ which maps every individual to an element of $\Delta^{\mathcal{I}}$, every atomic concept A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$, and every role R to a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ such that $(x, y) \in R^{\mathcal{I}}$ iff $(y, x) \in (R^-)^{\mathcal{I}}$. Moreover, for each pair of concepts C and D, each role R, and each $n \geq 1$,

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}, \qquad \bot^{\mathcal{I}} = \emptyset,$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}, \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}, \qquad (\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}},$
- $(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \{R^{\mathcal{I}}(x, y) \mid y \in \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}\} = \emptyset\},$
- $(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \{R^{\mathcal{I}}(x, y) \mid y \in C^{\mathcal{I}}\} \neq \emptyset\},$
- $(\geq nR.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \ |\{R^{\mathcal{I}}(x, y) \mid y \in C^{\mathcal{I}}\}| \geq n\},$
- $(\leq nR.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \ |\{R^{\mathcal{I}}(x, y) \mid y \in C^{\mathcal{I}}\}| \leq n\}.$

An interpretation $\mathcal{I}$ is a *model* of an assertion $A(a)$ (resp., $R(a, b)$) if $a^{\mathcal{I}} \in A^{\mathcal{I}}$ (resp., $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$). $\mathcal{I}$ is a *model* of a CI (resp., RI) $C \sqsubseteq D$ (resp., $R \sqsubseteq S$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ (resp., $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$); $\mathcal{I}$ is a *model* of a transitivity axiom trans(R) if $R^{\mathcal{I}} \circ R^{\mathcal{I}} \subseteq R^{\mathcal{I}}$, where $R^{\mathcal{I}} \circ R^{\mathcal{I}} := \{(x, y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists z \in \Delta^{\mathcal{I}} \text{ s.t. } (x, z), (z, y) \in R^{\mathcal{I}}\}$. An interpretation $\mathcal{I}$ is a *model* of an ABox $\mathcal{A}$ (resp., TBox $\mathcal{T}$) if it is a model of every axiom in $\mathcal{A}$ (resp., $\mathcal{T}$); $\mathcal{I}$ is a *model* of an knowledge base $\mathcal{K} = (\mathcal{A}, \mathcal{T})$ if $\mathcal{I}$ is a model of $\mathcal{A}$ and $\mathcal{I}$ is a model of $\mathcal{T}$.

3.3 Ontology-mediated query answering

In this section we first introduce *ontology-mediated query answering* (OMQA)—an ontological reasoning service which plays a fundamental role in the development of the Semantic Web—and then formally define the problem of performing OMQA via Datalog rewriting.

OMQA problem

Let N_V *(variables)* be a countably infinite set disjoint from N_C, N_R, and N_I. A *DL atom* has form $A(x)$ (*concept atom*) or $R(x, y)$ (*role atom*), where $A \in N_C$, $R = r$ or $R = r^-$ with $r \in N_R$, and $x, y \in N_V \cup N_I$. A *conjunctive query* q is an expression of the form:

$$q(x_1, \ldots, x_m) \equiv \exists(x_{m+1}, \ldots, x_n).\phi(x_1, \ldots, x_m, x_{m+1}, \ldots, x_n) \tag{3.1}$$

where $0 \leq m \leq n$ and ϕ is a non-empty conjunction of concept and role atoms over the variables $x_1, \ldots, x_n$, possibly involving individuals in N_I. With $\mathtt{var}(q)$ and $\mathtt{ind}(q)$ we denote the set of variables and the set of individuals occurring in q, respectively. Given an interpretation $\mathcal{I}$, a *match* for $\mathcal{I}$ and q is a function $\pi : \mathtt{var}(q) \cup \mathtt{ind}(q) \to \Delta^{\mathcal{I}}$ such that:

- $\pi(a) = a^{\mathcal{I}}$ for each $a \in \mathtt{ind}(q)$;
- $\pi(x) \in A^{\mathcal{I}}$ for each concept atom $A(x)$ which occurs in ϕ; and
- $(\pi(x), \pi(y)) \in R^{\mathcal{I}}$ for each role atom $R(x, y)$ which occurs in ϕ.

With a conjunctive query q of the form 3.1, the *answers* to q with respect to an interpretation $\mathcal{I}$ are defined as the set

$$q(\mathcal{I}) := \{(\pi(x_1), \ldots, \pi(x_m)) \mid \pi \text{ is a match for } \mathcal{I} \text{ and } q\}$$

of m-tuples obtained by evaluating q against $\mathcal{I}$. The set of *certain answers* to q over a knowledge base $\mathcal{K}$ is

$$\mathtt{cert}(\mathcal{K},q) := \bigcap_{\mathcal{I} \in \mathtt{mods}(\mathcal{K})} q(\mathcal{I})$$

where $\mathtt{mods}(\mathcal{K})$ denotes the set of all models of $\mathcal{K}$. Finally, *ontology-mediated query answering* is formally defined as the problem of computing $\mathtt{cert}(\mathcal{K}, q)$.

Figure 3-2: OMQA via Datalog rewriting with a pure approach.

Datalog rewriting

For the DL fragments introduced in Section 3.2, OMQA can be performed via rewriting the knowledge base and the query into a Datalog$^{\neg s,\neq}$ program in which a fresh *output predicate* is used to collect all the answers.

We can distinguish different rewriting approaches [3]. In this document we will focus on the one in which $\mathcal{A}$ is independent from $\mathcal{T}$—known as *pure approach*. From a knowledge base $\mathcal{K} = (\mathcal{A}, \mathcal{T})$ and a conjunctive query q, the approach in this rewriting method (shown in Figure 3-2) is to rewrite:

(i) $\mathcal{A}$ into a database D; and

(ii) both $\mathcal{T}$ and q into a Datalog$^{\neg s,\neq}$ program P with a fresh output predicate *ans* of arity $m = |\mathtt{var}(q)|$ so that

$$\mathtt{cert}(\mathcal{K}, q) = \{(a_1, \ldots, a_m) \mid ans(a_1, \ldots, a_m) \in TP(D \cup P)\}.$$

In Chapter 9 we discuss in depth about the main existing tools in the literature supporting OMQA over ontologies falling at least in OWL 2 RL, whose approach is to express inference tasks for OWL in terms of inference tasks for Datalog.

Part II

DaRLing rewriter

Chapter 4

Rewriting techniques

The W3C Web Ontology Language (OWL) is a powerful knowledge representation formalism at the basis of many semantic-centric applications. Since its unrestricted usage makes reasoning undecidable already in case of very simple tasks, expressive yet decidable fragments have been identified. Among them, we focus on OWL 2 RL, which offers a rich variety of semantic constructors, apart from supporting all RDFS datatypes. Although popular Web resources - such as DBpedia - fall in OWL 2 RL, only a few systems have been designed and implemented for this fragment. None of them, however, fully satisfy all the following desiderata: (*i*) being freely available and regularly maintained; (*ii*) supporting ontology-mediated query answering; (*iii*) properly applying the *sameAs* property without adopting the unique name assumption; (*iv*) dealing with concrete datatypes. To fill this gap, we conceived *DaRLing* [34], a freely available Datalog rewriter for OWL 2 RL ontological reasoning under SPARQL queries.

In what follows, after providing an overview of the system, we describe the rewriting strategies that *DaRLing* implements and how it handles the `owl:sameAs` property via Datalog. For the sake of simplicity, we use Datalog to refer to Datalog$^{\neg_s,\neq}$ in this chapter.

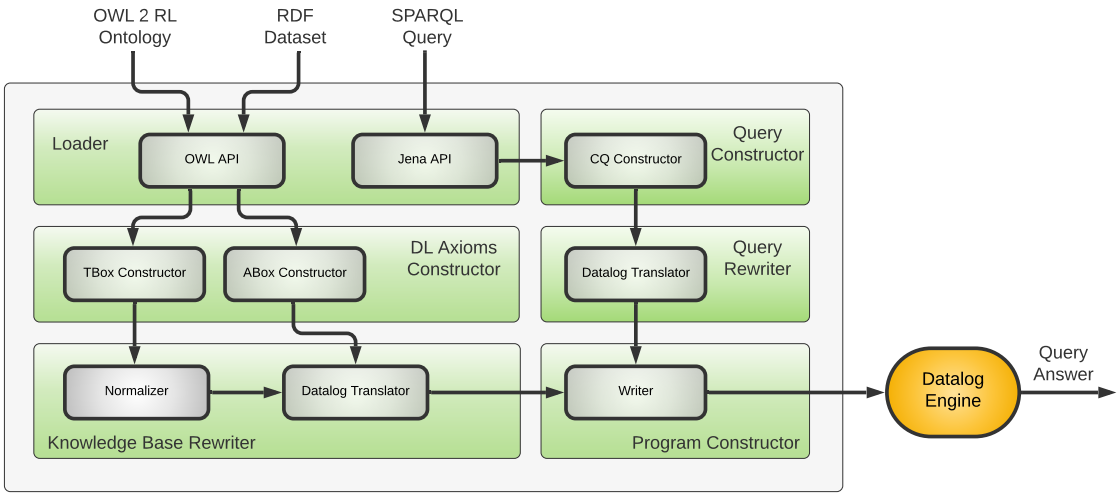

Figure 4-1: *DaRLing*'s Architecture.

4.1 System overview

DaRLing is an open-source Datalog rewriter for OWL 2 RL ontologies available on the online webpage https://demacs-unical.github.io/DaRLing/.

The rewriter implements the translation techniques described in Section 4.2 and supports the `owl:sameAs` management method described in Section 4.3. In addition to the Datalog rewriting of OWL 2 RL ontologies, *DaRLing* also supports the Datalog translation of datasets in RDF/XML or Turtle format and SPARQL queries.

The architecture of *DaRLing* is synthesized in Figure 4-1. *DaRLing* uses the OWL API [43] to load the RL fragment of OWL 2 ontologies and datasets into internal data structures representing TBoxes and ABoxes, respectively. The system supports the loading of the datatypes *xsd:string* and *xsd:integer*, provided by the OWL 2 datatype map (a list of the datatypes that can be used in OWL 2 ontologies) for the representation of strings and integers. A rewrite module is implemented for the translation of a TBox and/or an ABox into a Datalog program. More in detail, a part of the axioms (those "directly rewritable") is passed to the *Datalog Translator*, whereas the remaining part is first subjected to the

normalization procedure described in Section 4.2.

DaRLing also allows for Datalog translation of SPARQL queries containing *Basic Graph Patterns* (BGPs) (i.e., sets of triple patterns forming a graph). SPARQL queries are parsed using Jena [1]—a Java API which can be used to create and manipulate RDF graphs. Then they are translated into conjunctive Datalog queries.

The system supports different input formats and knowledge bases organized in multiple files. More in detail:

(i) the ontology (resp., the dataset) can be contained in a single file or in a folder containing multiple files with one of the extensions *.owl* or *.rdfs* (resp., *.owl*, *.rdf* or *.ttl*);

(ii) one or more queries have to be contained in a file with the *.SPARQL* extension.

Moreover, *DaRLing* can produce a suitable rewriting also if some inputs are missing: for each file (or folder) received as input, a single *.asp* file containing the respective Datalog translation is returned as output. For example, in case the ABox is missing, then the generated program is simply equivalent to the pair TBox plus query.

By default, *DaRLing* rewrites under UNA, it is however possible to explicitly choose to enable rewriting with the "`owl:sameAs` management mode" described in Section 4.3.

4.2 From OWL 2 RL to Datalog

In this section we describe how the TBox underlying an OWL 2 RL ontology is rewritten into Datalog. Since translating role inclusions and transitive axioms is almost trivial, we will focus only on the concept inclusions. In particular, the rewriting process takes place as follows:

(i) a class of axioms for which we provide a direct translation is identified;

[1] See https://jena.apache.org/

(*ii*) the remaining part of the axioms is subjected to a *normalization procedure* (i.e., a procedure that simplifies their complex nature) before being translated.

Concerning the ABox rewriting, assertions of type $A(a)$ and $R(a, b)$—where A is an atomic concept and R is a role—are directly translated into unary and binary Datalog facts, respectively. Finally, any assertion of type $C(a)$, with C not atomic, is treated as if it were the pair of axioms $A_C(a)$ and $A_C \sqsubseteq C$, where A_C is a fresh atomic concept uniquely associated with C.

4.2.1 Direct translation

For the first phase, we need to introduce the concepts of the description logic $\mathcal{ELI}$.

Definition 4.2.1 ($\mathcal{ELI}$-concepts)**.** $\mathcal{ELI}$-concepts *are inductively defined as follows:*

(*i*) $\top$ *and each* $A \in N_C$ *is an* $\mathcal{ELI}$*-concept, and*

(*ii*) *if* C, D *are* $\mathcal{ELI}$*-concepts and* R *is a role, then* $C \sqcap D$, $\exists R.C$ *and* $\geq 1R.C$ *are* $\mathcal{ELI}$*-concepts.*

We show how to directly get the equivalent Datalog rule for an axiom of the form

$$\sqcup_{i=1}^{m} C_i \ \sqsubseteq \ \sqcap_{j=1}^{n} A_j \tag{4.1}$$

where C_i's are $\mathcal{ELI}$-concepts and A_j's are atomic concepts. Note that, since an axiom of the form 4.1 is equivalent to the set of axioms $C_i \sqsubseteq A_j$, with $1 \leq i \leq m$ and $1 \leq j \leq n$, it is sufficient to show how the translation works on concept inclusions of the form $C \sqsubseteq A$, where C is an $\mathcal{ELI}$-concept and A is atomic.

Algorithm 1 shows a recursive algorithm for generating literals starting from an $\mathcal{ELI}$-concept C. Intuitively, for each conjunction $\sqcap C_i$ (possibly consisting of a single clause), we have a variable common to each clause C_i. For every clause C_i of the form $\exists R.D$ or $\geq 1R.D$ we introduce a fresh variable obtained by adding i to the subscript of the variable

shared by the conjunction to which C_i belongs. The translation of an axiom $C \sqsubseteq A$ (as

Algorithm 1: translate$\mathcal{ELI}$

```
Input: ELI-concept C, String Var, int clause, Set<Literal> bodyLiterals
Output: Addition of Datalog literals to bodyLiterals
if C is atomic then
    bodyLiterals.add(C(Var));
else if C ≡ ⊓C_i then
    foreach i do
        translateELI(C_i, Var, i, bodyLiterals);
    end
else if C ≡ ∃R.D ∨ C ≡ ≥ 1R.D then
    String newVar = Var_clause;
    if R is an inverse role then
        bodyLiterals.add(R⁻(newVar,Var)) ;
    else
        bodyLiterals.add(R(Var,newVar))
    translateELI(D, newVar, 1, bodyLiterals);
```

above) is a rule whose head is the literal $A(X)$ and whose body literals are obtained by invoking Algorithm 1 on C with Var $= X$ and clause $= 1$.

Example 4.2.1. *The concept inclusion* $\exists r.(\exists s.(C \sqcap D)) \sqcap \geq 1t.(E \sqcap \exists u^{-}.F) \sqsubseteq A$ *translates directly to the following Datalog rule:*

$$A(X) \leftarrow r(X, X_1),\ s(X_1, X_{1,1}),\ C(X_{1,1}),\ D(X_{1,1}),$$
$$t(X, X_2),\ E(X_2),\ u(X_{2,1}, X_2),\ F(X_{2,1}).$$

In more detail, the variable shared by clauses $r.(\exists s.(C \sqcap D))$ *and* $\geq 1t.(E \sqcap \exists u^{-}.F)$ *is* X*. The recursive call on these two clauses generates the body literals* $\{r(X, X_1),\ s(X_1, X_{1,1}),\ C(X_{1,1}),\ D(X_{1,1})\}$ *and* $\{t(X, X_2),\ E(X_2),\ u(X_{2,1}, X_2),\ F(X_{2,1})\}$ *respectively.* □

Concept Inclusion (CI)	Equivalent Datalog Rule
$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq \bot$	$\bot \leftarrow A_1(X),\ \ldots,\ A_n(X).$
$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq A$	$A(X) \leftarrow A_1(X),\ \ldots,\ A_n(X).$
$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq \forall R.A$	$A(Y) \leftarrow R(X,Y),\ A_1(X),\ \ldots,\ A_n(X).$
$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq \leq 1R.A$	$sameAs(Y_1,Y_2) \leftarrow A_1(X),\ \ldots,\ A_n(X),$ $R(X,Y_1),\ R(X,Y_2),\ A(Y_1),\ A(Y_2),\ Y_1 \neq Y_2.$

Table 4.1: Translation of normal concept inclusions in normalized form.

4.2.2 Normalization procedure

For the second phase of the rewriting process, we transform the remaining axioms of the TBox (those not directly translatable) into a form from which the Datalog translation is immediate.

Definition 4.2.2 (Normal form)**.** *We say that a TBox $\mathcal{T}$ is in* normalized form *if each concept inclusion axiom in $\mathcal{T}$ has form*

$$\sqcap A_i \sqsubseteq C$$

where $\sqcap A_i$ is a finite conjunction of atomic concepts and C is a concept of the form $\bot$, A, $\forall R.A$ or $\leq 1R.A$, with A atomic.

Table 4.1 shows how the Datalog translation of a TBox in normalized form takes place; in particular, $A, A_1, \ldots, A_n$ are atomic concepts, and $R(X,Y) = r(Y,X)$ if $R = r^-$.

We bring our axioms to a normalized form by readapting and optimizing a normalization procedure described by Kazakov [44, 45], which is applicable to any Horn-$\mathcal{SHIQ}$ TBox and preserves the logical consequences of the ontological axioms.

If on the one hand normalization allows us to easily translate a given ontology into Datalog, on the other, it could significantly increase the number of axioms. In what follows we describe, with the help of some examples, a version of the normalization procedure ad-hoc for OWL 2 RL. We also show how we enhance that procedure in order to avoid, where

possible, an unnecessary growth of the number of axioms.

Normalization aims at reducing the complex structure of axioms by introducing fresh concept names for substructures and substituting them. Intuitively, the transformation works as follows: let C be a complex concept containing D as a sub-expression; then, a fresh concept name A_D is introduced and constrained to extensionally coincide with D. This enables us to exchange all occurrences of D in C by A_D.

Formally, given a OWL 2 RL TBox $\mathcal{T}$, for every (sub-)concept C in $\mathcal{T}$ we introduce a fresh atomic concept A_C and define a function $st(C)$ by:

$$st(A) = A \text{ } (A \text{ atomic}); \qquad st(\bot) = \bot; \qquad st(\top) = \top;$$
$$st(\neg C) = \neg A_C; \qquad st(C \sqcap D) = A_C \sqcap A_D; \qquad st(C \sqcup D) = A_C \sqcup A_D;$$
$$st(\forall R.C) = \forall R.A_C; \qquad st(\exists R.C) = \exists R.A_C; \qquad st(\geq nR.C) = \geq nR.A_C;$$
$$st(\leq nR.C) = \leq nR.A_C \ .$$

The result of applying *structural transformation* to $\mathcal{T}$ is an ontology $\mathcal{T}'$ that contains all role inclusions and transitivity axioms in $\mathcal{T}$ in addition to the following axioms:

- $A_C \sqsubseteq st(C)$ for every C occurring positively in $\mathcal{T}$
- $st(C) \sqsubseteq A_C$ for every C occurring negatively in $\mathcal{T}$
- $A_C \sqsubseteq A_D$ for every concept inclusion $C \sqsubseteq D \in \mathcal{T}$

Example 4.2.2. *The axiom $\exists r.(B \sqcap C) \sqsubseteq \forall s^-.D$ will be transformed into:*

$$(R.1)\ A_{\forall s^-.D} \sqsubseteq \forall s^-.A_D \qquad (R.2)\ A_D \sqsubseteq D \qquad (R.3)\ \exists r.(A_{B \sqcap C}) \sqsubseteq A_{\exists r.(B \sqcap C)}$$
$$(R.4)\ A_B \sqcap A_C \sqsubseteq A_{B \sqcap C} \qquad (R.5)\ B \sqsubseteq A_B \qquad (R.6)\ C \sqsubseteq A_C$$
$$(R.7)\ A_{\exists r.(B \sqcap C)} \sqsubseteq A_{\forall s^-.D}$$

where $(R.1)$-$(R.2)$ derive from the positive occurrences of the concepts $\forall s^-.D$ and D,

whereas $(R.3)$-$(R.6)$ *derive from the negative occurrences of* $\exists r.(B \sqcap C)$, $B \sqcap C$, B *and* C, *respectively.* □

By applying structural transformation to $\mathcal{T}$, we obtain a TBox $\mathcal{T}'$ containing only concept inclusions of the form $A_1 \sqsubseteq A_2$, $A \sqsubseteq st(C_+)$ and $st(C_-) \sqsubseteq A$, where C_+ occurs positively and C_- occurs negatively in $\mathcal{T}$. Since $\mathcal{T}$ belongs to OWL 2 RL, C_+ can only be of the form $\top$, A, $\neg C$, $C \sqcap D$, $\forall R.C$ or $\leq 1R.C$, whereas C_- can only be of the form $\top$, A, $C \sqcap D$, $C \sqcup D$, $\exists R.C$ or $\geq 1R.C$. Therefore, axioms in $\mathcal{T}'$ which do not appear in normalized form are transformed as follows:

$$
\begin{array}{lll}
(t1) & A \sqsubseteq st(\neg C) = \neg A_C & \Longrightarrow A \sqcap A_C \sqsubseteq \bot; \\
(t2) & A \sqsubseteq st(C \sqcap D) = A_C \sqcap A_D & \Longrightarrow A \sqsubseteq A_C,\ A \sqsubseteq A_D; \\
(t3) & A_C \sqcup A_D = st(C \sqcup D) \sqsubseteq A & \Longrightarrow A_C \sqsubseteq A,\ A_D \sqsubseteq A; \\
(t4) & \exists R.A_C = st(\exists R.C) \sqsubseteq A & \Longrightarrow A_C \sqsubseteq \forall R^-.A; \\
(t5) & \geq 1R.A_C = st(\geq 1R.C) \sqsubseteq A & \Longrightarrow A_C \sqsubseteq \forall R^-.A.
\end{array}
$$

Example 4.2.3. *The axiom* $(R.3)$ *of the Example 4.2.2 is not in normalized form and will be replaced with the axiom* $A_{B \sqcap C} \sqsubseteq \forall r^-.A_{\exists r.(B \sqcap C)}$. □

The rewriting process takes a huge advantage from the fact that many axioms (those described in 4.2.1) are not subject to the normalization procedure. We further enhance that procedure by providing that:

(i) no fresh concept is introduced for $\top$, $\bot$ and all the atomic concepts in the TBox;

(ii) axioms already in normalized form are not subjected to the normalization process.

The following examples show how our rewriting technique significantly reduces the number of rewritten axioms (and consequently the number of Datalog rules that derive from them) compared with the standard Kazakov normalization.

Example 4.2.4. *With the concept inclusion of Example 4.2.2, through the application of the structural transformation we produce, in addition to* $(R.3)$ *and* $(R.7)$*, only the two axioms:*

$$(R'.1)\ A_{\forall s^-.D} \sqsubseteq \forall s^-.D \qquad\qquad (R'.2)\ B \sqcap C \sqsubseteq A_{B \sqcap C}\,.$$

Then $(R.3)$ *is replaced as in Example 4.2.3. Eventually, a concept inclusion already in normalized form like* $A \sqcap B \sqsubseteq \forall r.C$*, for which the standard normalization would generate* 6 *further axioms, is not subjected to the normalization procedure.* □

Example 4.2.5. *The application of the standard Kazakov normalization over the concept inclusion*

$$\exists r.(\exists s.(C \sqcap D)) \sqcap\ \geq 1t.(E \sqcap \exists u^-.F) \sqsubseteq A$$

from Example 4.2.1, would give rise to the 13 axioms in normalized form reported below, and as many Datalog rules in the rewriting.

(1) $A_{\exists r.(\exists s.(C \sqcap D))\ \sqcap\ \geq 1t.(E \sqcap \exists u^-.F)} \sqsubseteq A_A$ (2) $A_C \sqcap A_D\ \sqsubseteq A_{C \sqcap D}$

(3) $A_{\exists r.(\exists s.(C \sqcap D))}\ \sqcap\ A_{\geq 1t.(E \sqcap \exists u^-.F)} \sqsubseteq A_{\exists r.(\exists s.(C \sqcap D))\ \sqcap\ \geq 1t.(E \sqcap \exists u^-.F)}$

(4) $A_{(\exists s.(C \sqcap D))}\ \sqsubseteq \forall r^-.A_{\exists r.(\exists s.(C \sqcap D))}$ (5) $A_{(E \sqcap \exists u^-.F)} \sqsubseteq \forall t^-.A_{\geq 1t.(E \sqcap \exists u^-.F)}$

(6) $A_{(C \sqcap D))}\ \sqsubseteq \forall s^-.A_{\exists s.(C \sqcap D)}$ (7) $A_F \sqsubseteq \forall u.A_{\exists u^-.F}$ (8) $A_E \sqcap A_{\exists u^-.F} \sqsubseteq A_{E \sqcap \exists u^-.F}$

(9) $E \sqsubseteq A_E$ (10) $F \sqsubseteq A_F$ (11) $C \sqsubseteq A_C$ (12) $D \sqsubseteq A_D$ (13) $A_A \sqsubseteq A$.

The translation algorithm implemented in DaRLing instead produces a single Datalog rule for the axiom above (compare Example 4.2.1).

The following result is a direct consequence of Proposition 1 and Lemma 2 present in [44] and of the fact that the DL underpinning the OWL 2 RL profile constitutes a fragment of Horn-$\mathcal{SHIQ}$. It ensures that the normalization procedure described in this section preserves the logical implications of a OWL 2 RL ontology and can be performed

in polynomial time.

Proposition 4.2.1. *Let $\mathcal{T}$ be an OWL 2 RL TBox and $\mathcal{T}'$ be the TBox obtained by applying the structural transformation to $\mathcal{T}$. Then the TBox $\mathcal{T}''$ obtained by applying the transformation laws $(t_1)-(t_5)$ to $\mathcal{T}'$ preserves (non)entailment of axioms over the signature of $\mathcal{T}$ (i.e., axioms containing no new symbols).*

Finally, the correctness of the rewriting technique derives from Proposition 4.2.1 and the equivalence between the axioms (in normalized form) and the rules shown in Table 4.1, according to the rule-based semantics specified in [57](Section 4.3).

4.3 Handling `owl:sameAs` via Datalog

The `owl:sameAs` is a property which is used by many OWL 2 ontologies to declare equalities between resources. Hereinafter we will use (teletype) `sameAs` to denote the DL role related to the `owl:sameAs` property, whereas (italics) *sameAs* will refer to the respective Datalog binary predicate name in the rewritings.

The assertion $\texttt{sameAs}(a, b)$ states that the individuals a and b are synonyms, i.e., a can be replaced with b without affecting the meaning of the ontology. Note that the equivalence between two resources is not solely derivable from explicit assertions occurring in the ABox, it can be in fact derived continuously during materialisation. Suffice it to observe, for example, how the rewriting procedure of the TBox can generate rules whose head contains the *sameAs* predicate (compare Table 4.1, last line).

Example 4.3.1. *Consider an OWL 2 RL knowledge base containing (in the ABox) the following axioms:*

$$(\leq 1R.C)(a),\ R(a,b),\ R(a,c),\ C(b) \text{ and } C(c);$$

where R is a role, C is an atomic concept, and a, b and c are three syntactically different

individuals. The assertion $(\leq 1R.C)(a)$ *states that, with respect to the role* R*, the individual* a *can be related to at most one individual who is an instance of the concept* C*. Therefore, necessarily (unless the knowledge base is inconsistent) the individuals* b *and* c *must be equivalent, that is, they must belong to the same* `owl:sameAs`*-clique. According to the translation methods described in the previous section, this assertion is in fact processed as it were the following pair of axioms:*

$$D \sqsubseteq \leq 1R.C \text{ and } D(a),$$

where D *is a fresh atomic concept uniquely associated with the concept* $\leq 1R.C$*. In this way the following Datalog rule is obtained:*

$$sameAs(Y_1, Y_2) \leftarrow D(X),\ R(X, Y_1),\ R(X, Y_2),\ C(Y_1),\ C(Y_2),\ Y_1 \neq Y_2.$$

and the fact $sameAs(b, c)$ *will be derived from the materialization of the rewriting.* □

As we will see in the course of this section, other knowledge relating to equivalence between individuals is obtained during materialization in order to compute the reflexive, symmetric and transitive closure of the *sameAs* predicate.

Similarly to logic programming approaches, Datalog works under the *Unique Name Assumption* (UNA), i.e., it presumes that different names represent different objects of the world. With the following example we highlight the need of handling the `owl:sameAs` property in order to enable—in the equivalent Datalog program which derives from a given OWL 2 RL ontology—the match of equivalent constants for each join between variables in the body of a rule.

Example 4.3.2. *Let us consider an ontology featuring the rule*

$$DogOwner(X) \leftarrow hasPet(X, Y),\ Dog(Y).$$

together with the following set of facts:

$$\mathit{hasPet}(\text{“Peter”}, \text{“Brian”}). \quad \mathit{Dog}(\text{“BrianGriffin”}). \quad \mathit{sameAs}(\text{“Brian”}, \text{“BrianGriffin”}).$$

Note how, despite that sameAs(“Brian”,“BrianGriffin”) has the purpose of making the constants “Brian” and “BrianGriffin” interchangeable, the fact DogOwner(“Peter”) is not derived as it should. □

In the rest of this section we describe how we handle the rewriting of Web ontologies containing the `owl:sameAs` property. In particular, in order to preserve the `owl:sameAs` semantics in rewriting we must take into account that:

(*i*) `owl:sameAs` is an equivalence relation and it is therefore necessary to identify its equivalence classes (the `owl:sameAs`-*cliques*); and

(*ii*) individuals belonging to the same `owl:sameAs`-clique must match in joins between variables.

4.3.1 Cliques detection

To implement item (*i*) above, the idea is to simulate the reflexivity, the symmetry and the transitivity of the `owl:sameAs` through a fresh binary predicate which connects all the elements of a `owl:sameAs`-*clique* to a representative (the lexicographic minimum) of that clique. To this end, given $N \geq 0$, we add the following block of rules to our Datalog program:

$$\mathit{sameAs}(X, Y) \leftarrow \mathit{sameAs}(Y, X). \tag{1}$$

$$\mathit{noStart}(X_1) \leftarrow \mathit{sameAs}(X_0, X_1),\ X_0 < X_1. \tag{2.1}$$

$$\mathit{noStart}(X_2) \leftarrow \mathit{sameAs}(X_0, X_1),\ \mathit{sameAs}(X_1, X_2),\ X_0 < X_2. \tag{2.2}$$

$$\vdots \qquad\qquad \vdots$$

$$noStart(X_N) \leftarrow sameAs(X_0, X_1),\ \ldots,\ sameAs(X_{N-1}, X_N),\ X_0 < X_N. \qquad (2.N)$$

$$sameComp(X, Y) \leftarrow sameAs(X, Y),\ not\ noStart(X),\ X < Y. \qquad (3)$$

$$sameComp(X, Z) \leftarrow sameComp(X, Y),\ sameAs(Y, Z),\ X < Y,\ X < Z. \qquad (4)$$

$$sameComp(X, X) \leftarrow sameComp(X, Y). \qquad (5)$$

Here, rule (1) is the symmetric closure of the *sameAs* predicate, representing the pairs of equivalent individuals. Rules (2.1)-(2.N) map into the unary predicate *noStart* all individuals that are greater than another individual at a distance less than or equal to N with respect to the *sameAs* predicate. Note that by $N = 0$ we mean that rules (2.1)-(2.N) are not considered and *noStart* is not populated. Given a constant c that is not part of the extension of *noStart*, rules (3) and (4) compute the pairs (c, d) where $c < d$ and d can be reached from c via the *sameAs* transitive closure. Since the minimum constant $c_{\min}$ of every owl:sameAs-clique C can not populate the *noStart* predicate, it turns out that a fact $sameComp(c_{\min}, d)$ is generated for each individual d of C. Eventually, rule (5) is the reflexive closure of *sameComp* with respect to its first argument.

Let $P^N_{\sim}$ be the Datalog program consisting of the rules (1)-(5) above, with $N \geq 0$. Then, for each owl:sameAs-clique C whose lexicographical minimum is denoted $c_{\min}$, the following result ensures that $P^N_{\sim}$ materializes the set $\{sameComp(c_{\min}, c) \mid c \in C\}$ over C, for each $N \geq 0$. As we will see, the value of N determines the size of the extension of the *sameComp* predicate. In particular, as the size of N increases, such an extension decreases. On the other hand, it is also true that the choice of a too large value of N could result in a waste in terms of time for the materialization of the cliques.

Theorem 4.3.1. *Let S be a set of ground facts over the binary predicate sameAs and $\bar{S}$ be the smallest superset of S such that*

(*i*) $sameAs(c, c) \in \bar{S}$ *for each constant c occurring in S;*

(*ii*) $sameAs(c, d) \in \bar{S}$ *if* $sameAs(d, c) \in \bar{S}$*; and*

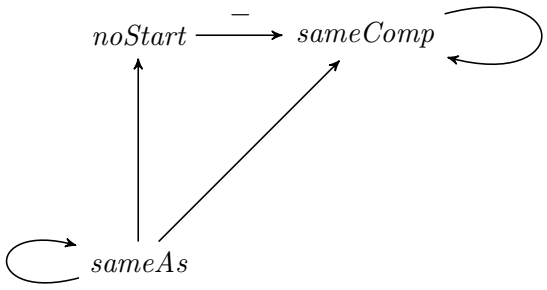

Figure 4-2: Dependency graph of program $P^N_{\sim}$.

(iii) $sameAs(c, e) \in \bar{S}$ *if* $sameAs(c, d) \in \bar{S}$ *and* $sameAs(d, e) \in \bar{S}$.

Then, for each $sameAs(c, d) \in \bar{S}$ *with* $c \neq d$*, there exists an individual* $\tilde{c}$ *such that both* $sameComp(\tilde{c}, c)$ *and* $sameComp(\tilde{c}, d)$ *belong to the (unique) answer set* $TP(P^N_{\sim} \cup S)$ *of* $P^N_{\sim} \cup S$*, for any* $N \geq 0$.

Proof. Figure 4-2 reports the dependency graph $\mathcal{G}_{P^N_{\sim}}$ of $P^N_{\sim}$. The strongly connected component of $\mathcal{G}_{P^N_{\sim}}$ are $C_1 = \{sameAs\}$, $C_2 = \{noStart\}$ and $C_3 = \{sameComp\}$. According to the fixpoint semantics given in Section 2.4, the answer set of $P^N_{\sim}$ over S is $TP(P^N_{\sim} \cup S)$. In particular, $TP(P^N_{\sim} \cup S) = I_3 = T^3_{P^N_{\sim}} \Uparrow I_2$, where

$$
\begin{aligned}
&I_0 = S;\\
&I_1 = T^1_{P^N_{\sim}} \Uparrow I_0 = I_0 \cup \{sameAs(d, c) \mid sameAs(c, d) \in I_0\};\ \text{and}\\
&I_2 = T^2_{P^N_{\sim}} \Uparrow I_1 = I_1 \cup \{noStart(c) \mid \exists 0 \leq n \leq N \text{ and } c_0, \ldots, c_{n-1} \text{ s.t. } c_0 < c,\\
&\qquad\qquad sameAs(c_0, c_1) \in I_1, \ldots, sameAs(c_{n-1}, c) \in I_1\}
\end{aligned}
$$

Let $sameAs(c, d) \in \bar{S}$ with $c \neq d$, C be the clique to which c and d belong, and $c_{\min} \in C$ be the lexicographical minimum of C.

Case $c_{\min} \in \{c, d\}$**.** Without any loss of generality, assume that $c_{\min} = c$.

- If $sameAs(c, d) \in I_1$, since $noStart(c) = noStart(c_{\min}) \notin I_2$—otherwise there would be an element in C less than $c_{\min}$—the fact $sameComp(c_{\min}, d)$ derives directly from rule (3) of $P^N_{\sim}$.

- If $sameAs(c, d) \notin I_1$ instead, there exist $0 < k \leq N$ and $c_1, \ldots, c_k \in C$ such that $sameAs(c, c_1) \in I_1$, ..., $sameAs(c_k, d) \in I_1$. Therefore, the fact $sameComp(c_{\min}, c_1)$ derives—for a similar argument of that of the previous point—from rule (3), whereas $sameComp(c_{\min}, c_2)$, ..., $sameComp(c_{\min}, c_k)$ and finally $sameComp(c_{\min}, d)$ are obtained by repeatedly applying rule (4).

In both scenarios, once $sameComp(c_{\min}, d)$ is derived, the fact $sameComp(c_{\min}, c)$ derives from rule (5).

Case $c_{\min} \notin \{c, d\}$. We prove that $sameComp(c_{\min}, c) \in TP(P^N_{\sim} \cup S)$, the other case (i.e., $sameComp(c_{\min}, d) \in TP(P^N_{\sim} \cup S)$) is specular.

- If $sameAs(c_{\min}, c) \in I_1$ there is nothing to prove.

- If $sameAs(c_{\min}, c) \notin I_1$, there exist $0 < k \leq N$ and $c_1, \ldots, c_k \in C$ such that $sameAs(c_{\min}, c_1) \in I_1$, ..., $sameAs(c_k, c) \in I_1$. Hence, the fact $sameComp(c_{\min}, d)$ derives from rule (3) and repeatedly applying rule (4) as above.

We have therefore proved that, regardless of the value assumed by N, for each pair of individuals c and d belonging to a owl:sameAs-clique C, there exists an individual $\tilde{c}$ (the lexicographical minimum of C) such that both $sameComp(\tilde{c}, c)$ and $sameComp(\tilde{c}, d)$ belong to $TP(P^N_{\sim} \cup S)$. □

Even though rules (1)-(5) ensure, for any $N \geq 0$, that each element of a owl:sameAs-clique is linked to the minimum of that clique by means of the *sameComp* predicate, many $sameComp(c, d)$ facts could arise, where c is not the minimum of any clique. The purpose of rules (2.1)-(2.N) is precisely to avoid the generation of these extra facts as much as possible.

Example 4.3.3. *Consider a lexicographically ordered set of* 5 *individuals* $i_1 < \cdots < i_5$ *together with the set* F *of facts:*

$$sameAs(i_1, i_4). \quad sameAs(i_2, i_5). \quad sameAs(i_3, i_4). \quad sameAs(i_4, i_5).$$

Figure 4-3 shows a graph representing F*, and also how individuals* $i_1, \ldots, i_5$ *belong to the same* `owl:sameAs`*-clique. The application of rules (1)-(5) over* F*, by varying* N*, produces the following facts about the predicate sameComp:*

$$sameComp(i_1, i_1). \quad sameComp(i_1, i_2). \quad sameComp(i_1, i_3). \qquad (N \geq 1);$$
$$sameComp(i_1, i_4). \quad sameComp(i_1, i_5).$$

$$sameComp(i_2, i_2). \quad sameComp(i_2, i_3). \quad sameComp(i_2, i_4). \qquad (N = 1, 2);$$
$$sameComp(i_2, i_5).$$

$$sameComp(i_3, i_3). \quad sameComp(i_3, i_4). \quad sameComp(i_3, i_5). \qquad (N \geq 3).$$

Among the derived facts, the only necessary are those corresponding at the "$N \geq 1$*" label. Those are the ones that connect the individual* i_1 *(the least one) to the rest of the clique's individuals, and they are in fact derived for each value* $N \geq 1$*. It is possible to see how as* N *varies, in addition to the above facts, other extra facts about predicate sameComp are materialized. In this case, the minimum size extension of predicate sameComp is reached for the first time in* $N = 3$*.* □

As we will see later, the *sameComp* predicate is used whenever any join variable occurs in a rule or query in order to enable matches between individuals belonging to the same clique, therefore it is very important to keep its growth under control. It is easy to foresee that the larger N is chosen the more the extension size of *sameComp* is reduced (due

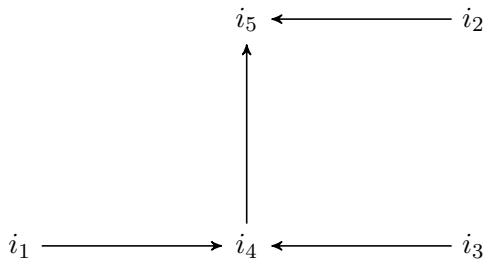

Figure 4-3: owl:sameAs graph from Example 4.3.3.

to the growing size of the extension of *noStart*), but at the expense of a possible greater consumption of time due to the computation of the paths in the generation of *noStart* instances. The choice of N must therefore be weighted, according to the application domain at hand, in such a way that ideally, both the time and the extension size of *sameComp* are minimized. More detailed considerations are reported in Chapter 5 where we considered different values of N for the real-world domain of DBpedia and experimentally identified $N = 2$ as the best compromise in the trade-off between time and space.

4.3.2 Preserving owl:sameAs semantics in joins

In the following we specify how we rewrite any rule in which appears at least a join variable. For each rule r, let J_r (the *join variables* of r) be the set of the variables occurring more than once in the body of r, and 2^{J_r} be the power set of J_r. For $X \in J_r$ we denote with $\#^r_X$ the number of occurrences of the variable X in the body of r. For each $V \in 2^{J_r}$ we produce a new rule r_V obtained from r as follows: for any $X \in V$ and $1 \leq i \leq \#^r_X$, we substitute the i-th occurrence of X with a fresh variable X_i and add a new body literal $sameComp(X, X_i)$.

Example 4.3.4. *For instance, with the rule in Example 4.3.2 we have that* $J_r = \{Y\}$ *and*

we get

$$DogOwner(X) \leftarrow hasPet(X, Y_1),\ Dog(Y_2),\ sameComp(Y, Y_1),\ sameComp(Y, Y_2).$$

as additional rule of the program. □

Intuitively, since in general the *sameAs* (and consequently *sameComp*) predicate does not involve all the constants of a program, any possible combination in which some join variables enables matches between constants syntactically different but linked by the `owl:sameAs` relationship has to be considered.

We eventually observe that, in most cases, only a few join variables occur in programs deriving from web ontologies. However, in order to prevent a potentially exponential growth of the program, we rewrite rules that have 3 or more join variables by projecting the *sameComp* predicate on each argument in which such a variable appears.

Example 4.3.5. *Consider the rule r as follows:*

$$GGFather(W) \leftarrow\ Alive(W),\ fatherOf(W, X),\ fatherOf(X, Y),\ fatherOf(Y, Z).$$

We have that $J_r = \{W, X, Y\}$. In this case, the 3 rules below are produced to make sameComp reflexive with respect to all the individuals that populate argument in which appear a join variable.

$$sameComp(X, X) \leftarrow\ Alive(X).$$
$$sameComp(X, X) \leftarrow\ fatherOf(X, \_).$$
$$sameComp(X, X) \leftarrow\ fatherOf(\_, X).$$

Then, the rule

$$GGFather(W) \leftarrow Alive(W_1),\ fatherOf(W_2, X_1),\ fatherOf(X_2, Y_1),\ fatherOf(Y_2, Z),$$

$$sameComp(W, W_1),\ sameComp(W, W_2),\ sameComp(X, X_1),$$
$$sameComp(X, X_2),\ sameComp(Y, Y_1),\ sameComp(Y, Y_2).$$

is added to the Datalog ontology, instead of the 8 *rules we would have obtained considering the power set* 2^{J_r} *of* J_r*.* □

By doing this we avoid all the possible combinations of the variables and we get a single new rule that binds the predicate *sameComp* to all the join variables simultaneously.

Further attempts have been made to manage `owl:sameAs` via Datalog (starting from the most naive one in which the reflexive, symmetrical and transitive closure is performed over the *sameAs* predicate directly, passing through more refined techniques in which the transitive closure relies on a fresh predicate name) but none of them was found to be applicable in practice. This reinforces our approach and highlights how important is handling `owl:sameAs` in an optimal way.

Chapter 5

Experimental evaluation

To demonstrate the practical applicability of *DaRLing*, we designed and conducted an experimental evaluation based on the following working hypotheses:

(*i*) over synthetic OWL 2 RL benchmarks, *DaRLing*'s output is comparable with the one produced by existing tools in terms of both number of produced rules and quality of the rewriting (the latter measured via execution time fixed the Datalog engine);

(*ii*) over real-world OWL 2 RL knowledge bases, *DaRLing*'s rewriting strategy enables scalable query answering even in case the UNA is not a viable option, i.e., in cases where there is an implicit or explicit use of the `owl:sameAs` relation and the UNA cannot be adopted.

5.1 Set-up

Since *DaRLing* is a rewriter which does not rely on any Datalog engine, according to Table 1.1 and Chapter 9, the only tool that can be fairly tested against *DaRLing* is the part of Clipper providing the Datalog rewriting of an OMQ, hereinafter called *Clipper-Rew*. Moreover, since these rewriters are independent from the evaluation phase, for the purpose of our testing, the choice of the Datalog engine is immaterial. In our case, we simply opted

for I-DLV[1] [4, 23], an Answer Set Programming instantiator and full-fledged state-of-the-art Datalog reasoner. Indeed, the system when fed with a disjunction-free and stratified under negation program is able to fully evaluate it and compute its perfect model. As benchmarks, we relied on LUBM,[2] Adolena, Stock Exchange, Vicodì[3] and DBpedia [12].

LUBM (Lehigh University BenchMark) [41] consists of a university domain OWL 2 ontology along with customizable and repeatable synthetic data and a set of 14 SPARQL queries. Queries 2, 6, 9 and 14 involve constants, while the other queries are constant-free. We restricted to the fragment falling in RL and considered its 14 canonical queries.

Adolena, Stock Exchange and Vicodì have been derived from a well-established benchmark [51]. They are expressed in DL-Lite$_R$ and provided with 5 queries each. Adolena (Abilities and Disabilities OntoLogy for ENhancing Accessibility) has been developed for the South African National Accessibility Portal. Stock Exchange and Vicodì are two real world ontologies widely used in literature for the evaluation of query rewriting systems [56]. Stock Exchange is an ontology of the domain of financial institution within the EU, while Vicodì is an ontology of European history. For LUBM we adopted 6 datasets, wherease for each of Adolena, Stock Exchange and Vicodì, we used 5 datasets, downloaded from https://www.mat.unical.it/dlve.

Differently from the other aforementioned benchmarks, that are synthetic and assume the UNA, DBpedia is a real-world knowledge base requiring a proper handling of the `owl:sameAs` property. More in detail, DBpedia is a well-known KB falling in OWL 2 RL and created with the aim of sharing on the Web the multilingual knowledge collected by Wikimedia projects in a machine-readable format. The dataset has been extracted from the latest stable release of the whole DBpedia dataset. The considered part consists of the English edition of Wikipedia and is composed by about half a billion RDF triples. We inherited a set of 10 queries from an application conceived to query DBpedia in natural

[1] See https://github.com/DeMaCS-UNICAL/I-DLV
[2] See http://swat.cse.lehigh.edu/projects/lubm/
[3] See http://www.vicodi.org

	Materialize		*Query-Driven*	
	C-Rew	**DaRL**	**C-Rew**	**DaRL**
1. LUBM				
ABox	*28.33*			
TBox	*5.30*	*4.75*	-	-
q1	13.85	15.43	0.80	1.10
q2	14.04	14.36	4.37	3.28
q3	13.67	13.69	0.79	1.10
q4	16.32	16.87	3.51	3.41
q5	13.21	14.55	2.19	4.17
q6	15.33	14.71	1.72	2.13
q7	14.38	14.38	1.13	1.19
q8	15.15	14.99	2.26	2.38
q9	15.02	15.53	2.56	2.71
q10	14.17	13.84	1.09	0.93
q11	12.94	13.43	0.07	0.13
q12	13.10	12.87	0.05	0.05
q13	13.32	13.92	2.86	4.35
q14	14.79	15.45	1.54	1.54

2. Adolena				
ABox	*4.78*			
TBox	*11.13*	*11.28*	-	-
q1	0.36	0.47	6.14	6.1
q2	1.03	0.58	6.72	6.63
q3	47.75	46.2	57.08	56.58
q4	1.13	0.89	6.82	6.83
q5	51.94	49.68	61.08	58.95
3. Stock Exchange				
ABox	*1.13*			
TBox	*3.80*	*3.30*	-	-
q1	0.05	0.04	0.16	0.16
q2	0.81	0.85	3.47	2.93
q3	28.51	29.37	31.14	30.01
q4	7.68	7.77	10.62	9.7
q5	265.72	258.36	270.85	257.1
4. Vicodì				
ABox	*7.75*			
TBox	*9.65*	*9.89*	-	-
q1	7.63	7.36	0.13	0.05
q2	12.55	12.69	11.37	11.31
q3	9.50	9.33	3.19	3.37
q4	11.18	11.33	3.69	3.47
q5	11.13	11.44	3.69	3.51

Table 5.1: Experiments over all datasets per each query of LUBM, Adolena, Stock Exchange and Vicodì. C-Rew stands for Clipper-Rew, DaRL for *DaRLing*.

language [50]. All tests were performed on a machine having two 2.8GHz AMD Opteron 6320 processors and 128 GB of RAM. All rewritings for each benchmark and executables are available at https://demacs-unical.github.io/DaRLing.

5.2 Quality

In this former set of experiments, we first generated *Clipper-Rew* and *DaRLing* rewritings for all queries of LUBM, Adolena, Stock Exchange and Vicodì; then, over these rewritings and for all considered datasets, we executed I-DLV under two different scenarios: in the scenario *materialize* the system is forced to materialize the whole ontology and then

prompted to answer to each query individually; in the scenario *query-driven* the system still runs each query one by one, but performs a more efficient evaluation tailored on the query at hand by enabling the magic sets technique [10].

Table 5.1 reports average running times in seconds of I-DLV executions over all datasets on LUBM, Adolena, Stock Exchange and Vicodì, respectively. In particular, for better highlighting differences in performance, for each execution we extracted from total time the static and fixed times spent on ABox loading and on TBox materialization over the ABox. These fixed times are reported in the table as *ABox* and *TBox*, respectively. Notice that TBox materializing times are not reported in case of scenario *query-driven* as there no materialization is done. Results show that in most cases performance achieved by I-DLV when using *DaRLing* outputs is comparable w.r.t. *Clipper-Rew*. Some worsening is observable especially on LUBM queries 5 and 13. This is reasonable since *Clipper-Rew* requires to take into account the query at hand for properly translating the query and the TBox, thus generating an output optimized on the basis of the query. *DaRLing* instead follows a different principle as it is designed to produce general and query-independent TBox rewritings without specific query-oriented enhancements. On Adolena and Stock Exchange, I-DLV times with *DaRLing* rewritings are generally, slightly better than with *Clipper-Rew*. Regarding Vicodì, I-DLV performance are practically the same since *Clipper-Rew* and *DaRLing* produced almost identical rewritings. This is mainly because Vicodì TBox, when rewritten by both *DaRLing* and *Clipper-Rew* into Datalog, consists of linear rules, i.e., rules having only an atom in body; thus, when the presence of joins is limited both approaches result almost equivalent.

5.3 Scalability

We also experimented on DBpedia with two types of rewritings of 10 DBpedia queries generated via *DaRLing*. In particular, we first measured the costs, in terms of both time and space, of materializing the *sameComp* predicate by varying the value of N (see Sec-

Parameter	N = 0	N = 1	N = 2	N = 3	N = 4
Time (seconds)	5,033	2,673	864	26,355	timeout
#sameComp	523M	342M	103M	77M	-
Memory (GB)	35	23	7	5	-

Table 5.2: Costs of the *sameComp* materialization on DBpedia (time limit set to 10 hours).

tion 4.3). To this end, we generated via I-DLV the materialization of the whole TBox under the UNA and filtered out the tuples of the *sameAs* predicate. The resulting dataset has been paired with rules (1)-(5) by considering different values of N. Table 5.2 reports the values of the following three parameters by varying N: the time (in seconds) needed for materializing the *sameComp* predicate, the extension size of the *sameComp* predicate and the memory consumption in GB.

Ideally, the optimal value of N should be the one minimizing both the time and the space (the memory consumption or, equivalently, the extension size of *sameComp*). As it is evident from Table 5.2, such a unique value does not exist. Indeed, the minimum value of time is reached for $N = 2$, whereas the minimum value of space is reached for $N = 3$. Between the two values, the best compromise seems to be offered by $N = 2$, since a small blow-up in terms of space is highly reward in terms of time saving.

We thus decided to focus on $N = 2$ and to evaluate, per each of the 10 queries, performance when I-DLV is provided with *DaRLing* rewritings generated for this value of N. Results are reported in Table 5.3. We restricted experiments to *DaRLing* as we purposely want to investigate scalability of *DaRLing* under the non-UNA. In addition, comparisons with other tools (cfr. Table 1.1) would result unfair since to our knowledge, *DaRLing* is the only open-source project empowered with an ad-hoc handling of the `owl:sameAs` property. As in quality-measuring experiments, we considered both the *materialize* and *query-driven* scenarios. We reported execution times in seconds. *ABox* and *TBox* times, as in the above results, represent times spent on ABox loading and on TBox materialization over the ABox, respectively. Columns report for each query running times from which

	Materialize	***Query-Driven***
ABox	*3,209*	
TBox	*4,048*	-
q1	27.86	1,574
q2	44.14	1,576
q3	46.17	3,453
q4	46.58	6,192
q5	77.29	1,636
q6	43.94	1,544
q7	1.05	1,416
q8	19.50	1,419
q9	0.01	1,496
q10	0.11	1,377

Table 5.3: Experiments on DBpedia for $N = 2$. Times are in seconds.

ABox and *TBox* times have been subtracted. In the *materialize* scenario, we observe that in general, once the TBox is materialized, the system is able to compute query answers in at most 1.3 minutes. A greater effort is paid if instead of materializing the TBox, the system is requested to perform a query-driven computation. Notably, the total time needed to answer the hardest query (namely q4 in the *query-driven* scenario) is 36% of the time spent for materializing just the *sameComp* predicate for $N = 3$ (see Table 5.2). This behaviour reinforces the choice of $N = 2$ as the best value for this domain. Concerning memory, in both scenarios and for each query, the memory peak is around 62 GB (including loading and materialization).

5.4 Discussion

To demonstrate the practical applicability of *DaRLing*, we have designed and conducted an experimental evaluation based on two working hypotheses, which have been both confirmed. The first release of *DaRLing* demonstrates to produce over synthetic OWL 2 RL benchmarks more general rewritings equivalent or sightly differing from the ones generated by *Clipper-Rew*, the closer open-source competitor of *DaRLing*. *DaRLing*'s output is

comparable with the one produced by *Clipper-Rew* in terms of both number of produced rules and, as evidenced by the results in Table 5.1, quality—in term of execution time—of the rewriting. As additional feature, *DaRLing* can be used for transparently handling the `owl:sameAs` property independently from the Datalog reasoner at hand albeit requiring extra work due to the intrinsic need of computing the transitive closure. Such costs are strictly dependent from the ontology at hand; our experimentation in an unfriendly setting of a large ontology such as DBpedia proved a (not taken for granted [58]) applicability of the approach. The system allows to select the value of N in the non-UNA rewriting mode. As shown by the results in Table 5.2, the value $N = 2$ is optimal for the knowledge base in question, but it may not be optimal for other domains. In general, the optimal value of N depends on the structure of the data but also on the ontology from which the implicit knowledge of the `owl:sameAs` property can be obtained. To derive the optimal value of N, an ad-hoc experimentation is therefore necessary for the knowledge base taken into consideration. In most large-scale applications, where the knowledge base does not undergo large variations over time, the optimal value of N can remain unchanged even for very long periods of time. It will therefore be sufficient to experiment and update this value at intervals of length depending on the variation of the data and the ontology. Precisely to accommodate these scenarios, it is among our future plans to make the choice of this value for N automatic based on the knowledge base at hand.

Part III

Evaluation planner

Chapter 6

Planning techniques

The development of rewriting techniques—through which query answering on OWL is reduced to the evaluation of queries on Datalog programs—has meant that Datalog reasoning has become a topic of fundamental importance for the Semantic Web. Furthermore, with the more and more growing demand for semantic Web services over large datasets, an efficient evaluation of Datalog queries is arousing a renewed interest among researchers and industry experts.

Classical Datalog reasoners typically adopt sophisticated internal policies to speed-up the computation trying to limit the memory consumption. However, when the amount of data exceeds a certain size, these policies may result inadequate. This happens, for instance, for the full-fledged Datalog system I-DLV [23, 24]—originally conceived as grounding engine in DLV2 [8]. I-DLV historically uses strategies for join orderings and indexing that are applied rule-by-rule at runtime and that are based on local statistics over data that become available during the computation. As a result, for databases up to a few millions tuples, these strategies ensure fast evaluation at the expense of a reasonable amount of extra memory. Conversely, for databases with billions of tuples, both the time and the space used for implementing these strategies are too high.

This is the case of reasoning tasks over large-scale domains, such as those emerging

in industrial-level applications fostered by the advances in Industry 4.0, the Internet of Things, and Big Data[16]. When dealing with high volumes of data, in fact, performing heavy operations (such as loading or indexing) multiple times should be definitely avoided; thus, traditional ASP systems based on one-shot executions are rather unsuited. Recently, to cope with large-scale scenarios, I-DLV has been optimized and partially re-engineered by implementing novel techniques and heuristics to reduce memory consumption and possibly optimize execution times. This process also gave rise to DLV2-SERVER [49, 50], an extension of the system with a server-like modality, which is able to keep the main process alive and to receive and process user's requests on demand.

In this chapter we present one of the key approaches that is at the basis of the aforementioned improvements: the precomputation via Answer Set Programming (ASP) of an *evaluation plan* [7, 33] for a given Datalog program. The idea underlying the new technique is to precompute a global indexing schema for the underlying database associated with suitable body-orderings for all the program rules. As far as we know, this is the first state-of-the-art work whose approach is to compute a plan for evaluating a Datalog program in a pre-processing phase.

After a formal definition of an *evaluation plan*, we introduce the notion of *strategy* and then we specify the concept of *optimal* evaluation plan with respect to a certain strategy.

6.1 Admissible plans

Let P be a set of Datalog rules with non-empty body and let D be a database, namely a set of Datalog facts. We indicate with $\texttt{pred}(P \cup D)$ the set of all predicates occurring in $P \cup D$ and with $\texttt{rel}(p)$ the set $\{\alpha \in D : pred(\alpha) = p\}$ of the elements of D sharing the predicate name p. We write $p[i]$ to indicate the i-th argument of the predicate p.

In the following, after formalizing the standard notions of *ordering* of a rule and *indexing schema* of a Datalog program, we introduce the novel notion of *evaluation plan* along with

some preliminary definitions.

Definition 6.1.1. *Let r be a rule in P and $B(r)$ be the set of the atoms appearing in the body of r. Let F_a be a (possibly empty) subset of atoms in $B(r)$ and F_p be a subset of $\{1, \cdots, |B(r)|\}$. A* position assignment p_r on r *is a one-to-one map*

$$p_r : F_a \to F_p.$$

A pair (α, p) such that $p_r(\alpha) = p$ is called a fixed position *w.r.t. p_r. An* ordering on r *is a bijective function*

$$pos(r, \cdot) : B(r) \to \{1, \cdots, |B(r)|\}.$$

*Having fixed a position assignment p_r on r, we define a p_r-*ordering on r *as an ordering on r such that $pos(r, \alpha) = p_r(\alpha)$ for each $\alpha \in F_a$.*

The definition above presents a body ordering as a rearrangement of the literals in the body, but notably, allows for having a certain number of atoms in the body in some fixed positions. This is because, according to the knowledge of the domain at hand, if one is aware that a particular choice for the orderings is convenient, the planner can be driven so that only plans complying with this choice are identified.

Definition 6.1.2. *Let $U := \{p[i] : p \in \texttt{pred}(P \cup D), 1 \leq i \leq a(p)\}$, where $a(p)$ represents the arity of the predicate p. An* indexing schema $\mathcal{S}$ *over $P \cup D$ is a subset of U. Given a subset $I \subseteq U$, we say that $\mathcal{S}$* fixes *I if $I \subseteq \mathcal{S}$.*

In other words, an indexing schema is a subset of the arguments of all predicates in $\texttt{pred}(P \cup D)$. Furthermore, similarly to the definition of ordering that may allow for fixed positions, we give the possibility to fix also a set of indices.

Example 6.1.1. *As a running example in this section, we consider the following Datalog rule r :*

$$h(X, Z, W) \leftarrow \ a(X, Z),\, b(V, W),\, c(Z),\, d(V),\, e(Y, Z).$$

Consider the position assignment p_r which fixes the atom $b(V, W)$ in first position. A possible p_r-ordering may be:

$$pos(r, a(X, Z)) = 3, \quad pos(r, b(V, W)) = 1, \quad pos(r, c(Z)) = 5,$$
$$pos(r, d(V)) = 2, \quad pos(r, e(Y, Z)) = 4.$$

By means of such ordering the body atoms of r are rearranged as follows:

$$h(X, Z, W) \leftarrow \ b(V, W),\ d(V),\ a(X, Z),\ e(Y, Z),\ c(Z).$$

The set $\mathcal{S} := \{a[2], c[1], d[1], e[2]\}$ is an example of indexing schema over the predicates appearing in r. □

With a rule $r \in P$ we can associate a hypergraph $H(r) = (V, E)$ whose vertex set V is the set of all terms appearing in $B(r)$ and the edges in E are the term sets of each atom in $B(r)$. Given a rule r of P, a *connected component* of r is a set of atoms in $B(r)$ that define a connected component in $H(r)$.

Example 6.1.2. *Figure 6-1 shows the hypergraph $H(r)$ associated to r from Example 6.1.1. In particular $V = \{X, Y, Z, V, W\}$ and $E = \{\{X, Z\}, \{V, W\}, \{Z\}, \{V\}, \{Y, Z\}\}$. The connected components of r are $C_1 = \{a(X, Z), c(Z), e(Y, Z)\}$ and $C_2 = \{b(V, W), d(V)\}$.* □

We introduce now the notions of *separation* between two connected components and *well ordering* of a component of a rule.

Definition 6.1.3. *Let r be a rule of P and $pos(r, \cdot)$ be an ordering on r. Two connected components C_1 and C_2 of r are* separated *w.r.t. $pos(r, \cdot)$ if*

$$\max\{pos(r, \alpha) : \alpha \in C_1\} < \min\{pos(r, \beta) : \beta \in C_2\}$$

or vice versa.

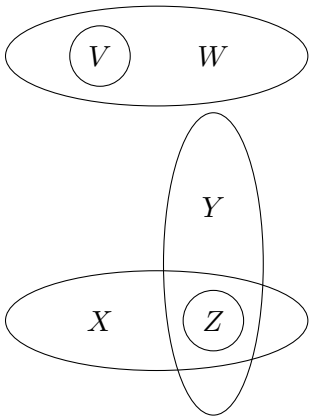

Figure 6-1: Hypergraph $H(r)$ associated to r from Example 6.1.1.

An argument of an atom appearing in the body of a rule r, is said to be *bound*, w.r.t. an ordering on r, if it is either a constant or a variable appearing in a previous atom, and is said to be *indexBound*, w.r.t. an ordering on r and an indexing schema $\mathcal{S}$, if it is bound and it belongs to the schema $\mathcal{S}$. The definition below provides the notion of *well ordering* of a connected component in a rule.

Definition 6.1.4. *Let r be a rule of P, $\mathcal{S}$ be an indexing schema and $pos(r, \cdot)$ be an ordering on r. A connected component C of r is* well-ordered w.r.t. $\mathcal{S}$ and $pos(r, \cdot)$ *if, assuming $m = \min\{pos(r, \alpha) : \alpha \in C\}$, for each $\beta \in C$ with $pos(r, \beta) = j$ and $j > m$, it holds that:*

(*i*) *β has at least an argument which is indexBound, and*

(*ii*) *either all the arguments of β are bound or there is no other atom in a later position (in the same component) that, placed in place of β, would have all the arguments bound.*

Example 6.1.3. *Let's consider the rule r with the ordering $pos(r, \cdot)$ and the indexing schema $\mathcal{S}$ as in our running example. The two connected components of r are clearly separated w.r.t. $pos(r, \cdot)$. The indexBound arguments w.r.t. $pos(r, \cdot)$ and $\mathcal{S}$ are $c[1]$, $d[1]$*

and $e[2]$. It can be easily seen that the connected component C_2 is well-ordered w.r.t. $\mathcal{S}$ and $pos(r, \cdot)$. The same cannot be said for the component C_1; in fact, not all the arguments of the atom $e(Y, Z)$ are bound and the atom $c(Z)$, positioned in place of $e(Y, Z)$, would have all the arguments bound. □

The notion of separation among connected components is needed for identifying, within rule bodies, clusters of literals that do not share variables. The idea is that the ordering computed by the planner should keep separated these clusters in order to avoid, as much as possible, the computation of Cartesian products during the instantiation; at the same time, literals within the clusters are properly rearranged in order to comply with the selected indexing schema, thus avoiding the creation of further indices.

Next, we provide the *admissibility* property which, in turn, characterizes the evaluation plans.

Definition 6.1.5. *Given a rule $r \in P$ and an indexing schema $\mathcal{S}$, we say that an ordering $pos(r, \cdot)$ is* admissible *w.r.t. $\mathcal{S}$ if the connected components of r are mutually separated (w.r.t. $pos(r, \cdot)$) and well-ordered (w.r.t. $pos(r, \cdot)$ and $\mathcal{S}$).*

We define below an evaluation plan for a Datalog program.

Definition 6.1.6. *Let $\{p_r\,;\ r \in P\}$ be a given set of position assignments, and I be a given subset of $\{p[i] : p \in \texttt{pred}(P \cup D),\ 1 \leq i \leq a(p)\}$. An* evaluation plan *$\mathcal{E}$ of P consists of an indexing schema $\mathcal{S}$ that fixes I together with a p_r-ordering for each $r \in P$ being admissible w.r.t. $\mathcal{S}$. We say that P enjoys an* efficient evaluation *if it is associated to an evaluation plan.*

Example 6.1.4. *In our running example, the ordering $pos(r, \cdot)$ is not admissible w.r.t. the schema $\mathcal{S}$. Thus, $\mathcal{S}$ and $pos(r, \cdot)$ do not represent an evaluation plan of the program $P = \{r\}$. However, an evaluation plan of P could be obtained by exchanging the assignments of the atoms $c(Z)$ and $e(Y, Z)$ in $pos(r, \cdot)$. It would be appropriate to note that, in the latter case, we would obtain a further evaluation plan by excluding the argument $a[2]$ from the*

indexing schema (and thus saving an index). Starting from this consideration, we introduce the concepts of preference and evaluation strategy in the next section. □

With the next example we motivate the notion of *evaluation plan* given in this section. As announced, our goal is to plan an efficient instantiation of a Datalog program (i.e., the construction of the TP model by means of the fixpoint operator) by fixing a schema of indices and an ordering of each rule without knowing any statistics of the atoms that occur in the program, thus also avoiding the costs of calculating and keeping these statistics updated at runtime. Therefore, the purpose is to simulate the behavior of the most modern grounding systems [22, 35, 52, 63], relying exclusively on structural information on the input program.

The proposed theoretical construction aims to reduce the cost of instantiating the rules of a program in terms of the number of comparisons between tuples that populate pairs of adjacent predicates (with respect to a given ordering). In particular, we rely on the following simplifications:

1. The presence of an index in a position $p[i]$ in which occurs a bound variable X allows direct access (with unit cost) to the collection of tuples that populate p with respect to the i-th position. In other words, given a constant c that occurs in a previous atom in a position where the variable X appears, we get access at unit cost to the first constants tuple of p in which c occurs in position i, without the need to scan (at worst) the entire extension of p.

2. On the basis of the previous consideration, if all the arguments of an atom are indexBound, the cost of accessing its tuples will be unitary throughout the entire instantiation process.

3. The presence of an index on a non-bound position does not bring any benefit to the grounding process, but instead involves a waste of time and memory necessary to create a useless index.

4. Each Cartesian product (i.e., a pair of atoms in adjacent positions belonging to two distinct connected components) involves the scanning and therefore the comparison between all the tuples that populate the atoms that make up such a product. Furthermore, considering that the number of tuples satisfying the match of the atoms in a component is less than or equal to the product of their extensions, avoiding the instantiation of a Cartesian product in the middle of a component reduces the number of comparisons.

5. The absence of indexBound arguments in an atom involves a comparison of each tuple of its extension and each tuple of constants satisfying the preceding atoms.

6. The cost (in terms of memory consumption) of creating an index on an argument of an atom is proportional to the number of constants that occur in that argument.

Example 6.1.5. *Consider the program P consisting of the following rules:*

$$\begin{aligned} r_1 &: \quad h(X,Y) \leftarrow a(X), b(Y), c(X). \\ r_2 &: \quad k(Y,X) \leftarrow c(X), d(Y,X), e(X). \\ r_3 &: \quad j(X) \leftarrow c(X), f(X). \end{aligned}$$

together with the database D:

$$\begin{array}{lll} a(1). \ldots a(10). & b(1). \ldots b(1000). & c(6). \ldots b(15). \\ d(1,1). \ldots d(1,10). & & \\ \vdots & & \\ d(100,1). \ldots d(100,10). & & \\ e(10). \ldots e(19). & f(1). \ldots f(10). & \end{array}$$

Consider first an ordering of the rules that reflects the way they are written, and suppose

$\mathcal{S} = \emptyset$. *In that case, according to the assumptions* 1-6, *we get that:*

- *During the instantiation of rule* r_1, 10^4 *comparisons are made from the products of atoms* $a(X)$ *and* $b(Y)$. *The resultant* 10^4 *pairs of constants are then compared with each of the* 10 *facts related to the extension of* c. *For a total of* 10^5 *operations.*
- *Concerning* r_2, *the product between* $c(X)$ *and* $d(Y, X)$ *generates* 5×10^2 *pairs within* 10^4 *comparison operations. Such a pairs are then compared with the* 10 *facts related to the predicate* e. *For a total of* 1.45×10^4 *operations needed to instantiate* r_2.
- *Finally, instantiating* r_3 *involves* 10^2 *operations.*

We now separate the only pair of connected components present in P *by reversing the order of the atoms* $b(Y)$ *and* $c(X)$ *in* r_1. *In that case, since the join between* $a(X)$ *and* $c(X)$ *is satisfied only by* 5 *individuals, the number of operations performed to instantiate* r_1 *is reduced to* $95 + 5 \times 10^3$.

Consider now the indexing schema $\mathcal{S} = \{c[1], d[2], e[1], f[1]\}$ *in order to meet condition* (i) *of Definition 6.1.4. Since for each individual who populates* a *the cost of checking whether the same individual populates* c *is equal to* 1, *the number of operations required to instantiate* r_1 *is further reduced to* $5 + 5 \times 10^3$. *Concerning* r_2 *and* r_3, *you need* $5 + 5 \times 100$ *and* 10 *operations respectively.*

Reversing the order of the atoms $d(Y, X)$ *and* $e(X)$ *in* r_2, *we finally get an evaluation plan for* P. *The number of operations to instantiate* r_2 *is further reduced to* $9 + 10^2$. *Overall, we go from a total of about* 10^5 *to about* 5×10^3 *operations to instantiate the entire program* P. □

6.2 Preferences

In order to choose, among the various evaluation plans of a program P, those that minimize the memory consumption necessary for the construction of the indexing schema, we introduce the concept of *preferences*.

Let P be a Datalog program, E_P be the set of all the evaluation plans of P and $w : E_P \to \mathbb{N}$ be a function that we call *cost function on* E_P. Given two evaluation plans $\mathcal{E}_1, \mathcal{E}_2 \in E_P$, we say that $\mathcal{E}_1$ *is preferable to* $\mathcal{E}_2$ *w.r.t. the cost function* w if $w(\mathcal{E}_1) < w(\mathcal{E}_2)$, while we say that $\mathcal{E}_1$ *is equivalent to* $\mathcal{E}_2$ *w.r.t.* w if $w(\mathcal{E}_1) = w(\mathcal{E}_2)$.

We define an *evaluation strategy for* P as a finite sequence $\Sigma = (w_1, \ldots, w_k)$ of distinct of cost functions on E_P. We say that $\mathcal{E}_1 \in E_P$ *is preferable to* $\mathcal{E}_2 \in E_P$ *w.r.t. the strategy* $\Sigma = (w_1, \ldots, w_k)$ if either:

- $\mathcal{E}_1$ is preferable to $\mathcal{E}_2$ w.r.t. w_1, or
- there exists $j \in \{2, \ldots, k\}$ such that $\mathcal{E}_1$ is equivalent to $\mathcal{E}_2$ w.r.t. w_i for each $i = 1, \ldots, j-1$, and $\mathcal{E}_1$ is preferable to $\mathcal{E}_2$ w.r.t. w_j.

According to the notion of *preference* of an evaluation plan over another w.r.t. a strategy, we can now introduce the definition of "optimal" plans w.r.t. that strategy.

Definition 6.2.1. *Let E_P be the set of all the evaluation plans for a Datalog program P and Σ be an evaluation strategy for P. An evaluation plan $\mathcal{E}_0$ is said to be* optimal w.r.t. Σ *if it is either preferable or equivalent to each $\mathcal{E} \in E_P$ w.r.t. Σ.*

Intuitively, finding the optimal evaluation plans against a strategy $\Sigma = (w_1, \ldots, w_k)$ means finding those that minimize the cost function w_1, then, among these, finding those that minimize the function w_2, and so on.

We report next four functions used for defining our evaluation strategies. From now on when we talk about the functions w_1, w_2, w_3 and w_4 we will refer to the following:

- $w_1(\mathcal{E}) := \sum_{p[i] \in \mathcal{S}} c(p, i)$, where $c(p, i)$ is the cost of building an index over $p[i]$ in the indexing schema $\mathcal{S}$. Note that we presuppose the knowledge of $c(p, i)$ values. Such costs can be estimated via heuristics or actually computed, depending on the application domain at hand. As said in the introduction, the novel approach is based on the natural assumption that, when dealing with very large databases, some

information and statistics about the user domain are known in advance since they do not vary as fast as the actual data. Apart from primary keys and foreign keys, which in OMQA are related to the ontological schema, some statistics on the data can be also taken into account. This is the case, for example, of the *estimation of the selectivity* of an attribute, which gives an indication of the average number of times that a constant (or individual) occurs in the relation in correspondence of the given attribute (note that, the special case of estimation of the selection equal to 1 indicates that the given attribute is actually a key). This value can be taken into account for estimating the size (and thus, the cost) of an index for the given attribute.

- $w_2(\mathcal{E})$ is defined as the sum of the positions of atoms involved in recursion. We prefer that atoms involved in recursion are placed as soon as possible. The extension of such atoms might considerably grow and change during computation; placing them before other atoms in the body could avoid the creation of expensive indices.

- $w_3(\mathcal{E})$ is the number of indices set on arguments that are not primary keys. In other words we prefer indices set on arguments representing primary keys.

- $w_4(\mathcal{E}) := \sum_{r \in P} \sum_{\alpha \in B(r)} [maxArity - u(\alpha, r)] * pos(r, \alpha)$, where $maxArity$ represents the maximum arity of the atoms appearing in P and $u(\alpha, r)$ is the number of unbound arguments of the atom α in the rule r. We prefer that atoms having large number of unbound arguments (that is, those that minimize the first factor in the above summation) are placed as soon as possible as they possibly will lead to have new completely bound atoms to be placed in successive positions.

Example 6.2.1. *Consider the program P and the set D as in Example 6.1.5. According to assumption 6, it is clear that the evaluation plan described in the example is not optimal with respect to the cost function w_1. It is easy to verify that, unless reordering the connected components of rule $r1$, the only optimal evaluation plan foresees that the rules are ordered*

as follows:

$$
\begin{aligned}
r_1 : &\qquad h(X,Y) \leftarrow a(X), c(X), b(Y).\\
r_2 : &\qquad k(Y,X) \leftarrow e(X), c(X), d(Y,X).\\
r_3 : &\qquad j(X) \leftarrow f(X), c(X).
\end{aligned}
$$

with $\mathcal{S} = \{c[1], d[2]\}$. □

In the next chapter we provide an ASP based implementation for the computation of the optimal evaluation plans with respect to a given strategy.

Chapter 7

ASP-based implementation

In the following we describe the ASP code devised in order to compute optimal evaluation plans. For the sake of simplicity, as the program is rather long and involved, we report here only some key parts; the full ASP code is available online.[1]

The program is based on the classical GCO paradigm (see Section 2.3) and combines:

(i) *choice and disjunctive rules* to guess an indexing schema $\mathcal{S}$ over $P \cup D$ and, for each rule r in P, an ordering $pos(r, \cdot)$;

(ii) *strong constraints* to guarantee, for each rule r, the admissibility of $pos(r, \cdot)$ w.r.t. $\mathcal{S}$;

(iii) *weak constraints* to find out the optimal evaluation plans of P w.r.t. the chosen strategy.

7.1 Data model

The planner consists of an ASP program taking as input a set of facts representing P and the database D; each rule of P is represented by means of facts of the form:

$$rule(Rule, Description, NumberOfBodyAtoms).$$

[1] See https://www.mat.unical.it/perri/iclp2019.zip.

$$headAtom(Rule, Atom, Predicate).$$

$$bodyAtom(Rule, Atom, Predicate).$$

$$sameVariable(Rule, Atom1, Arg1, Atom2, Arg2).$$

$$constant(Rule, Atom, Arg).$$

Facts over the predicate *rule* associate each rule r to an identifier and provide the number of its body atoms. Furthermore, the second argument of such predicates is the string representing the Datalog rule itself. Atoms in the body and in the head of each rule r are represented by *bodyAtom* and *headAtom* predicates respectively. In particular, given an atom in the body or head of a given rule, the arguments of these ternary predicates represent, respectively, the rule identifier, the string by which the atom is represented in the rule and the predicate name of the atom. The predicate *sameVariable* provides the common variables related to every pair of atoms appearing in r, whereas *constant* states that a constant term occurs in the argument of an atom of r. An example of the basic input concepts described above is the following:

Example 7.1.1. *The following program* P :

$$h1(X) \leftarrow a(X, Y),\ b(Y).$$
$$h2(Y) \leftarrow a(Y, X).$$

is represented by means of the facts:

$$rule(0, \text{“}h1(X) \leftarrow a(X, Y),\ b(Y).\text{”}, 2).$$

$$headAtom(0, \text{“}h1(X)\text{”}, \text{“}h1/1\text{”}).$$

$$bodyAtom(0, \text{“}a(X, Y)\text{”}, \text{“}a/2\text{”}).$$

$$bodyAtom(0, \text{“}b(Y)\text{”}, \text{“}b/1\text{”}).$$

$$sameVariable(0, \text{“}h1(X)\text{”}, 1, \text{“}a(X,Y)\text{”}, 1).$$
$$sameVariable(0, \text{“}a(X,Y)\text{”}, 2, \text{“}b(Y)\text{”}, 1).$$

$$rule(1, \text{“}h2(Y) \leftarrow a(Y,X).\text{”}, 1).$$
$$headAtom(1, \text{“}h2(Y)\text{”}, \text{“}h2/1\text{”}).$$
$$bodyAtom(1, \text{“}a(Y,X)\text{”}, \text{“}a/2\text{”}).$$
$$sameVariable(1, \text{“}h2(Y)\text{”}, 1, \text{“}a(Y,X)\text{”}, 1).$$

□

The database D is represented by means of facts over predicate *relation*, whereas the costs of building indices over arguments are given by facts over predicate *index*.

$$relation(Predicate, Arity).$$
$$index(Predicate, Arg, Cost).$$

Furthermore, the planner allows for having a certain number of atoms in the body in some previously fixed positions and a set of indices fixed in $\mathcal{S}$. This is because, according to the knowledge of the domain at hand, if one is aware that a particular choice for the orderings and the indexing policy is convenient, the planner can be driven so that only plans complying with this choice are identified. The planner can also exploit the presence of arguments representing primary keys for predicates in $P \cup D$. Such information, if available, can be given in input to the ASP planner by means of facts of the form:

$$fixedPosition(Rule, Atom, Pos).$$
$$fixedIndex(Predicate, Arg).$$
$$key(Predicate, Arg).$$

7.2 Guessing part

The guessing part generates the solution search space, which in this case is represented by the Cartesian product between the set of all the possible indexing schemes and the set of all the possible ordering of the program rules.

The following choice rule (see Section 2) guesses a subset of the arguments of all predicates, namely an indexing schema $\mathcal{S}$, over $P \cup D$. Notably, the arguments to be indexed are chosen among a restricted set of arguments, called *indexable*, in order to keep the search space smaller. For instance, arguments that are not involved in joins are not indexable.

$$\{setIndex(Predicate, Arg)\} \leftarrow indexable(Predicate, Arg).$$

Beside this choice rule, the guess part contains also the following rule for guessing a body-ordering for each rule r in P. In particular, the choice guesses a position in the body for each atom whose position has not been previously fixed (*fixedAtomRule*). Clearly, only positions not already occupied by another body atom in the same rule (*fixedPositionRule*) are guessable. Predicates *fixedAtomRule* and *fixedPositionRule* are computed according to the predicate *fixedPosition* described above.

$$\{pos(A, R, P) : position(P),\ P \geq 1,\ P \leq Size,\ not\ fixedPositionRule(R, P)\} = 1$$
$$\leftarrow\ rule(R, Size),\ bodyAtom(R, A, \_),\ not\ fixedAtomRule(R, A).$$

7.3 Checking part

This part discards, by means of *strong constraints*, solutions that do not satisfy the conditions to be considered admissible evaluation plans. According to the definitions provided in Section 6.1, conditions that have to be necessarily satisfied are the following:

1. The *connected components* of each rule of P must be kept separate. According to

Definition 6.1.3, this is ensured by the following constraint.

$$\leftarrow pos(A_1, Rule, P_1),\ pos(A_2, Rule, P_2),\ sameComponent(Rule, A_1, A_2),$$
$$pos(A_3, Rule, P_3),\ not\ sameComponent(Rule, A_1, A_3),\ P_1 < P_3,\ P_3 < P_2.$$

2. According to the first point of Definition 6.1.4, to guarantee that each connected component is *well-ordered*, each atom, except those in the first position of each component, must have at least an argument *indexBound*. This condition is guaranteed by the constraint below, where predicates *firstPosition* and *indexBound* suggest, respectively, the first positions of the components in each rule, and the indexBound arguments of each atom in a rule.

$$\leftarrow pos(Atom, Rule, Pos),\ firstPosition(Rule, FirstPos),$$
$$Pos > FirstPos,\ \#\text{count}\{Arg :\ indexBound(Arg, Atom, Rule)\} = 0.$$

3. The second condition of Definition 6.1.4 is modeled by means of the following constraint. Here the predicate *atomVars* indicates the number of variables occurring in every atom.

$$\leftarrow pos(A, Rule, P),\ not\ boundAtom(A, Rule, P),\ pos(A_1, Rule, P_1),$$
$$not\ boundAtom(A_1, Rule, P_1),\ P_1 > P,\ P_2 > P_1,\ boundAtom(A_2, Rule, P_2),$$
$$atomVars(A_2, Rule, N),\ \#\text{count}\{Arg :\ sameVariable(Rule, A_2, Arg, A, \_)\} = N.$$

The checking part contains also an additional constraint encoding the following basic check for guaranteeing the correctness of the plans. In particular, this basic check ensures that

two different atoms do not occupy the same position in any rule:

$$\leftarrow \ pos(Atom_1, Rule, Pos),\ pos(Atom_2, Rule, Pos),\ Atom_1 \neq Atom_2.$$

7.4 Optimization part

Finally, in this section we describe the part for identifying the optimal evaluation plan according to the evaluation strategy that one decides to apply. Remember that a strategy is a finite combination of cost functions. Currently, the planner is equipped with the four cost functions described in Section 6.2, each of which is represented by a specific *weak constraint*. Note that, weak constraints allow for expressing preferences possibly having different importance levels. The planner allows to fix these priority levels according to the chosen strategy by providing in input facts which indicate that the cost function w_N has priority level P.

$$priorityCostFunction(N, P).$$

For instance, suppose we want to represent the strategy $\Sigma = (w_1, w_3, w_2)$, then we need the following input facts to indicate that the cost function w_1 has priority level 3, w_3 has priority level 2 and w_2 has priority level 1. Note that in this case the cost function w_4 is not activated.

$$priorityCostFunction(1, 3).$$
$$priorityCostFunction(3, 2).$$
$$priorityCostFunction(2, 1).$$

This means that the planner is customizable. Indeed, depending on the knowledge of the domain at hand, one can choose to adapt the strategy to his own needs simply by

exchanging the priority levels of the cost functions among those already present in the planner, or even by integrating new cost functions (with the addition of new constraints in the encoding). In the following we illustrate the weak constraints representing the cost functions defined in Section 6.2.

1. The rule below aims to minimize index occupation. To this end, we presuppose the knowledge of the costs (or their estimation) of building indices over arguments and we represent them by facts of form $index(Predicate, Arg, Cost)$.

$$:\sim setIndex(Predicate, Arg),\ index(Predicate, Arg, Cost),$$
$$priorityCostFunction(1, P).\ [Cost@P,\ Predicate, Arg, Cost]$$

2. We prefer that atoms involved in recursion are placed as soon as possible. The weak constraint makes uses of the auxiliary predicate *recursivePredicate* providing information about which predicates of the program are recursive. We do not report its definition for the sake of readability.

$$:\sim pos(Atom, Rule, Pos),\ bodyAtom(Rule, Atom, Predicate),$$
$$recursivePredicate(Predicate),\ priorityCostFunction(2, P).$$
$$[Pos@P,\ Rule, Pos]$$

3. Indices set on arguments representing primary keys are possibly preferred:

$$:\sim setIndex(Predicate, Arg),\ not\ key(Predicate, Arg),$$
$$priorityCostFunction(3, P).\ [1@P,\ Predicate, Arg]$$

4. Atoms having large number of unbound arguments should be placed as soon as possible in the body. Also in this case we make use of an auxiliary predicate: *num-*

BoundArgs provides the number of bound arguments of an atom in a rule.

$$:\sim \mathit{numBoundArgs}(\mathit{Atom}, \mathit{Rule}, \mathit{Pos}, B),\ \mathit{maxArity}(N),$$
$$\mathit{bodyAtom}(\mathit{Rule}, \mathit{Atom}, \mathit{Predicate}),\ \mathit{relation}(\mathit{Predicate}, \mathit{Arity}),$$
$$\mathit{priorityCostFunction}(4, P).\ [(N - \mathit{Arity} + B) * \mathit{Pos}@P,\ \mathit{Rule}, \mathit{Pos}]$$

Chapter 8

Experimental evaluation

Hereafter we report the results of an experimental activity carried out to assess the effectiveness of the ASP-based evaluation planner. Our experimental analysis relies on four benchmarks: LUBM, LUBM-LUTZ, Stock Exchange and Vicodì. In particular, LUBM, Stock Exchange and Vicodì are the same used in the experimentation of the *DaRLing* system, and their description is present in Chapter 5. LUBM-LUTZ [55] is a variant of LUBM which consists of an OWL 2 ontology and 11 queries (both different from those of LUBM) along with a modified version of the LUBM official generator allowing to set the level of incompleteness in the database.

8.1 Setting

The original LUBM ontology and the official 14 queries have been translated into Datalog via the Clipper system. [1] The official LUBM generator has been adopted to generate four databases of increasing sizes: LUBM-500, LUBM-1,000, LUBM-2,000 and LUBM-4,000, where the number associated to each database name indicates the number of universities

[1] The choice of Clipper as the system used for the rewriting of ontologies and queries depends on the fact that the experimental phase on the planner precedes the design of the *DaRLing* system. Furthermore, the ontologies used for the tests fall within the Horn-$\mathcal{SHIQ}$ description logic, therefore to use Darling it would have been necessary to eliminate the rules that do not fall within the RL fragment of OWL 2.

composing it. The number of facts in the databases ranges from about 67,000,000 to about half a billion facts. As done for LUBM, the ontology and the queries of LUBM-LUTZ have been translated into Datalog via the Clipper system. All queries are without constants. We generated five databases of increasing sizes and having an incompleteness percentage of 10%: LUTZ-500, LUTZ-1,000, LUTZ-2,000, LUTZ-4,000 and LUTZ-8,000. Again, the number associated to each database name indicates the number of universities composing it. For each of Stock Exchange and Vicodì, we selected 5 queries featuring constants and we used the SyGENiA generator [39] to produce five databases having from $1,000$ to $40,000$ tuples and a number of individuals varying from 100 to $4,000$. These are the maximum sizes that can be generated using SyGENiA.

Experiments on LUBM-LUTZ have been performed on a Dell Linux server with an Intel Xeon Gold 6140 CPU composed of 8 physical CPUs clocked at 2.30 GHz, with 297GB of RAM. Experiments on LUBM, Vicodì and Stock Exchange have been performed on a NUMA Linux machine equipped with two 2.8 GHz AMD Opteron 6320 processors and 128GB RAM. Unlimited time and memory were granted to running processes. Benchmarks and executables used for the experiments are available at https://www.mat.unical.it/perri/iclp2019.zip.

Two different executions have been compared:

(i) a classical execution of I-DLV which, given as input the so generated encodings, chooses body orderings and indexing strategies with its default policies; and

(ii) an execution driven by the planner in which I-DLV is forced to follow the precomputed evaluation plan that decided body orderings and indices in order to reduce memory consumption. These constraints have been defined via annotations, that represent specific means to express preferences over its internal computational process [23].

8.2 Planner customization

In the context of OMQA, where the objective is to answer a query, the rewritten Datalog program typically benefits from the application of the so-called Magic Sets technique [9]. This produces a new equivalent program containing extra intensional predicates that could have very small extensions during the computation. These predicates, in a setting where memory consumption should be limited, could be moved towards the end of the body so that it is more likely saving space for needed indices. Hence, to instantiate our planner, we consider this additional domain information and use facts of the form $\mathit{fixedPosition}(Rule, Atom, Pos)$ to specify it. It is worth remarking that in I-DLV this customization has an impact only in case of queries featuring constants since magic atoms are not generated for queries without constants. Furthermore, for all attributes involved in extensional relations, we provide, via facts of the form $\mathit{index}(Predicate, Arg, Cost)$, an estimation of the size of an index for that attribute. In particular, in our experiments, to have available this information we generate and analyze a "small" database for each benchmark.

In our experiments, we considered different planner customizations depending on the domain at hand. In particular, for LUBM, Vicodì and Stock Exchange, consisting mainly of queries featuring constants, we adopted the strategy $\Sigma_1 = (w_2, w_4)$. The idea underlying this choice is that, in such domains, I-DLV can benefit from the Magic Sets technique and fixing the positions of magic atoms as described above is already sufficient to drive the planner. On LUBM-LUTZ, having instead constant-free queries, we adopted the strategy $\Sigma_2 = (w_1, w_2, w_3, w_4)$; indeed, since Magic Sets are not active, no fixed positions can be provided and a richer strategy is necessary for avoiding an almost blind plan computation.

Query	*No Planner*		*Planner*			Saved Memory	Saved Time
	Time	Memory	Time	Memory	Planning Time		
				LUBM			
q01	1,949.24	17.27	1,526.94	17.67	0.01	-0.4	422.3
q02	2,292.53	19.22	1,121.70	17.27	0.14	2.0	1,170.8
q03	1,643.14	17.89	1,026.31	17.27	0.01	0.6	616.8
q04	2,250.26	37.21	1,241.08	33.28	0.30	3.9	1,009.2
q05	1,303.48	28.28	1,087.97	17.27	0.17	11.0	215.5
q06	1,485.75	23.74	1,383.24	21.12	0.05	2.6	102.5
q07	1,284.16	25.94	1,124.31	21.23	0.18	4.7	159.9
q08	1,349.36	32.47	1,162.35	22.49	0.32	10.0	187.0
q09	1,464.82	23.74	1,326.15	21.32	0.22	2.4	138.7
q10	1,271.19	24.94	1,101.55	20.84	0.14	4.1	169.6
q11	1,016.69	17.27	1,014.10	17.27	0.01	0.0	2.6
q12	1,275.24	27.13	1,115.07	18.68	0.20	8.4	160.2
q13	1,281.30	30.06	1,129.73	18.66	0.17	11.4	151.6
q14	1,122.83	17.27	1,080.39	17.27	0.00	0.0	42.4
Saved Memory		Average: 4.34		Maximum: 11.40		Improvements: 13/14	
Saved Time		Average: 324.94		Maximum: 1,170.83		Improvements: 14/14	
				LUBM-LUTZ			
q01	353.88	8.17	328.94	8.15	0.00	0.0	24.9
q02	237.25	6.19	232.24	6.20	0.04	0.0	5.0
q03	268.90	6.63	274.70	6.20	0.26	0.4	-5.8
q04	249.86	6.67	240.43	6.63	0.02	0.0	9.4
q05	246.67	6.21	240.60	6.21	0.08	0.0	6.1
q06	273.83	6.88	258.81	6.63	0.28	0.2	15.0
q07	219.15	6.20	218.20	6.20	0.05	0.0	0.9
q08	293.97	7.59	284.52	6.80	0.11	0.8	9.5
q09	260.12	6.80	247.42	6.20	0.01	0.6	12.7
q10	289.85	7.19	283.63	7.22	0.04	0.0	6.2
q11	288.08	7.14	280.95	6.44	0.07	0.7	7.1
Saved Memory		Average: 0.26		Maximum: 0.79		Improvements: 11/11	
Saved Time		Average: 8.28		Maximum: 24.94		Improvements: 10/11	

Table 8.1: Planner performance in terms of average running time and memory usage of I-DLV with and without planner, computed over all considered databases per each query of LUBM and LUBM-LUTZ. Memory is in GB, Time in seconds.

Query	*No Planner*		*Planner*			Saved Memory	Saved Time
	Time	Memory	Time	Memory	Planning Time		
Stock Exchange							
q01	0.83	11.98	0.83	12.16	0.00	-0.2	0.0
q02	1.10	23.48	1.15	21.10	0.02	2.4	-0.1
q03	2.06	39.64	2.41	36.60	0.03	3.0	-0.4
q04	1.31	29.04	1.32	25.28	0.04	3.8	0.0
q05	2.35	46.40	2.44	41.34	0.09	5.1	-0.1
Saved Memory	Average: 2.81		Maximum: 5.06			Improvements: 4/5	
Saved Time	Average: -0.10		Maximum: 0.00			Improvements: 2/5	
Vicodì							
q01	0.83	9.88	0.83	9.96	0.01	-0.1	0.0
q02	0.93	20.56	0.83	12.02	0.01	8.5	0.1
q03	0.82	11.38	0.78	10.72	0.02	0.7	0.0
q04	0.70	12.88	0.70	13.84	0.02	-1.0	0.0
q05	0.78	12.54	0.78	13.10	0.08	-0.6	0.0
Saved Memory	Average: 1.52		Maximum: 8.54			Improvements: 2/5	
Saved Time	Average: 0.03		Maximum: 0.11			Improvements: 5/5	

Table 8.2: Planner performance in terms of average running time and memory usage of I-DLV with and without planner, computed over all considered databases per each query of Stock Exchange and Vicodì. Memory is in MB, Time in seconds.

8.3 Discussion

The results of our experiments are reported in Table 8.1, Table 8.2 and in Figure 8-1 and 8-2.

Table 8.1 and Table 8.2 show performance in terms of average running time and memory usage of I-DLV (with and without planner) computed over all considered databases per each benchmark query. Columns 2 and 3 refer to the classical computation, while columns 4 and 5 to the computation driven by the planner. In the 6th column, we reported the time spent to compute the optimal plan, in the 7th column, the memory saving per query computed as difference of the corresponding fields in columns 3 and 5. Similarly, the 8th column reports the time saving per query computed as difference of the corresponding fields in columns 2 and 4. The table reports also some aggregated data per benchmark. In particular, it shows information on the average/maximum saved memory/time, as well as the number

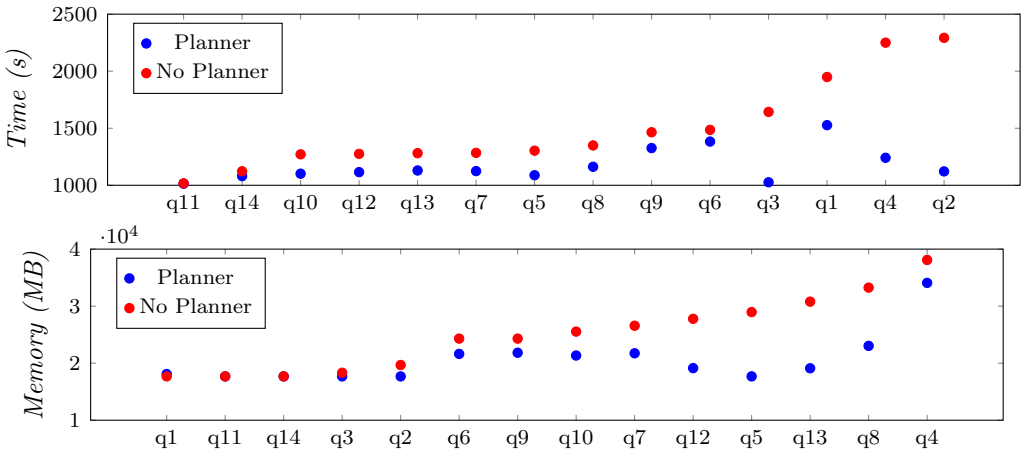

Figure 8-1: Plots of the average running time and memory usage of I-DLV (with and without planner) over all considered databases per each query of LUBM. Queries are ordered by increasing values w.r.t. the No-Planner execution.

of queries where an improvement in terms of saved memory (resp. saved time) has been obtained. In addition, to provide a clearer picture of the behavior of the two versions of I-DLV, we reported in Figures 8-1 and 8-2 plots of the average running time and memory usage over all considered databases for the largest benchmarks: LUBM and LUBM-LUTZ.

As it can be seen, we obtained a significant saving of memory on LUBM where, for instance, the planner allows to save 11.4 GB on query *q13* (about 40% less) w.r.t. the no-planner version, and a gain both in terms of memory and time over almost all queries. Only on query *q01* we experimented a small worsening on memory. In general, no significant increase of computation time is observable and, in several cases, the planner-driven approach leads also to improvements in terms of time. This can be explained considering that indices selected by the planner, being on the overall less memory expensive, are more efficiently computable.

Concerning LUBM-LUTZ, we first note that the benefits appear less evident. This is due to the nature of the queries in the benchmark which are constant-free and require a different (less informed) customization, as described in Section 8.2. Nonetheless, the

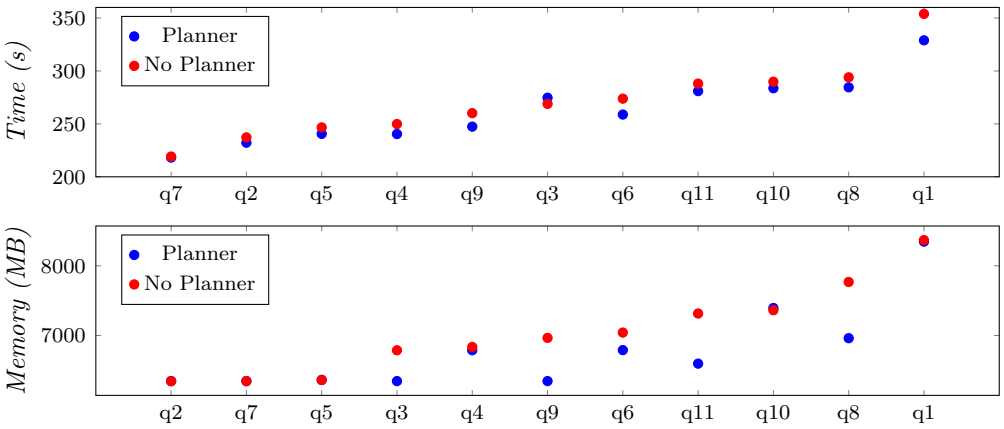

Figure 8-2: Plots of the average running time and memory usage of I-DLV (with and without planner) over all considered databases per each query of LUBM-LUTZ. Queries are ordered by increasing values w.r.t. the No-Planner execution.

execution of I-DLV driven by the planner performs generally better (both in time and memory) of the standard execution. On the queries *q08*, *q09* and *q11* we have a memory saving of 9-10% w.r.t. the no-planner version; moreover, we observe no worsening in memory consumption and only one case in which there is a negligible worsening in time.

As for Stock Exchange and Vicodì, although these are not data intensive domains, I-DLV can benefit by the planner as well. Indeed, worsenings in terms of memory range from 1% to 7% in a few queries which are somehow expected when measuring memory of the order of megabytes.

Further aggregated data and statistics on the results are given in Table 8.3. This shows, for both the tested versions of I-DLV and for each benchmark, the maximum peak of memory and the total sum of execution times computed over all databases and queries, along with the corresponding profits. In all benchmarks, the peak of memory when the planner is used is less than the one obtained using the standard version of I-DLV. Regarding times, although we experimented a small worsening for Stock Exchange (6.5%), we observe a general improvement which is greater than 20% in our large-scale benchmark.

Overall, we can claim that the proposed approach promotes considerable memory sav-

Domain	*Peak of Memory*			*Sum of Times*		
	No Planner	*Planner*	Profit (%)	*No Planner*	*Planner*	Profit (%)
LUBM	81,439.90	72,733.30	10.7%	83,960.48	65,763.48	21.7%
LUBM-LUTZ	21,701.70	21,634.20	0.3%	14,907.74	14,452.21	3.1%
Stock Exchange	77.40	68.90	11.0%	38.25	40.72	-6.5%
Vicodì	38.50	26.60	30.9%	20.32	19.58	3.7%

Table 8.3: Statistics on LUBM, LUBM-LUTZ, Stock Exchange and Vicodì: the maximum peak of memory and the total sum of execution times computed over all databases and queries, along with the corresponding profits. Time is in seconds, memory is in MB.

ings, especially in data-intensive benchmarks. The computation of a low memory consumption indexing schema also favors a saving in terms of execution time. Having an overall view of the ontology allows in fact to choose ordering in which only the indices necessary for the binding of the variables are created, thus avoiding the cost of creating unused indices.

However, there are some queries on which a noteworthy memory saving is not observable. This behavior can be explained by the fact that the standard I-DLV approach builds indices and reorders the rule bodies based on data statistics that are updated at runtime, each time a rule generates new instances of its head predicate. Our approach is based on the assumption that, on a large scale, the selectivities of predicate arguments do not vary greatly during materialization. However, it is clear that if the selectivity of an IDB predicate grows a lot, the cost of its indexing is unpredictable in a preprocessing phase. In fact, it would be necessary to estimate in advance the selectivity of the arguments of each IDB predicate at the end of the materialization. Although formulas could be used to estimate the variations in selectivity every time a rule is processed—i.e., every time the immediate consequence operator produces a non empty set of new facts on that rule—it would in any case be undecidable to establish in advance the number of times a rule will be processed in case of recursion.

This problem is already partially addressed by the planner through a cost function with which ordering is preferred in which the recursive predicates are placed as soon as possible.

This reduces the likelihood of indexing these predicates and therefore to lose control in the case of large variations on them. However, in order to optimally use this planner feature and give it the right level of priority, a thorough knowledge of the data and the ontology in question is required.

Part IV

Conclusion

Chapter 9

Related work

A number of effective practical approaches proposed in the literature perform ontology-mediated query answering via rewriting the ontology and the query into an equivalent Datalog program. Over the past decade, several rewriting algorithms have been proposed and systems performing query answering via rewriting have been designed, even for the same fragment of OWL. They may differ, apart from the rewriting techniques, also from the size and time spent to produce the rewritings, and their quality (measured in terms of time and space in the evaluation against popular benchmarks).

Concerning the existing tools supporting query answering against (at least) OWL 2 RL ontologies, in the following we mention, in chronological order, all those whose approach is to express inference tasks for OWL in terms of inference tasks for Datalog. We highlight the differences with our rewriter and explain why Clipper-Rew is the only system that can be fairly tested with.

- Orel [48] is a reasoning system which subsumes both the EL and the RL profile of the OWL 2 ontology language and its approach is based on a bottom-up materialisation of consequences in a database. In particular, ontological information are stored as facts, whereas logical ramifications are governed by "meta-rules" that resemble the rules of a deduction calculus. However, Orel supports neither SPARQL nor conjunctive

queries.

- OwlOntDB [32] works under the unique name assumption (UNA) to translate OWL 2 RL ontologies into Datalog programs, but it is no longer available.
- DReW [72] is a query answering system which supports OWL 2 RL and OWL 2 EL (modulo *datatypes*). It uses DLV as underlying Datalog engine and has not been conceived to generate ontology rewritings.
- RDFox [59] is a main-memory RDF store supporting Datalog reasoning with an efficient handling of `owl:sameAs` and SPARQL. After the initial development at University of Oxford, the system is now available commercially from Oxford Semantic Technologies, a spin-out of the University backed by Samsung Ventures and Oxford Sciences Innovation.
- Clipper [30] is a reasoner for conjunctive query answering over Horn-$\mathcal{SHIQ}$ ontology. Being more oriented to DL languages rather than OWL ontologies, Clipper lacks the support of the *datatypes* constructs and manages the `owl:sameAs` property under the UNA. It can also be used to generate ontology rewritings only.
- OWL2DLV [5] is a commercial system, builts on the ASP reasoner DLV2-SERVER [50], for evaluating SPARQL queries over very large OWL 2 knowledge bases whose associated DL falls within Horn-$\mathcal{SHIQ}$. As well as Clipper, OWL2DLV works under the UNA, but it cannot be used to generate ontology rewritings only.

According to Table 1.1, among the aforementioned systems, only RDFox properly applies `owl:sameAs` without UNA, whereas datatypes are supported by OwlOntDB, RDFox and OWL2DLV. RDFox is a materialization based reasoner that adopts a rewriting with a single predicate representing each RDF/RDFS triple. This system handles the `owl:sameAs` by choosing a representative resource for each `owl:sameAs`-clique and then replacing all resources with their representative. This type of approach, unlike ours, requires the rules

to be updated whenever a `owl:sameAs` triple is derived during materialization. Then rewriting cannot be applied as preprocessing. It also seems that—apart from *DaRLing*—the management of the implicit knowledge of the sameAs deriving from some axioms of the TBox and some assertions of the ABox is not dealt with by any state-of-the-art system.

Furthermore, compared to *DaRLing*: Orel is a reasoning system that supports neither SPARQL nor conjunctive queries; OwlOntDB is no longer available; DReW is a query answering system (based on the first generation of DLV) which has not been conceived to generate ontology rewritings only; RDFox is a commercial query answering system (based on its own reasoner) the trial version of which is not freely provided for testing even in academic contexts; OWL2DLV is a commercial system (based on the second generation of DLV) which cannot be used to generate ontology rewritings only and also it cannot be tested due to Samsung's restrictions.

DaRLing is an OMQ rewriter which produces a standard Datalog program that can be evaluated by any Datalog reasoner. Clipper is a query answering system (based on the first generation of DLV) which can also be used to generate ontology rewritings only. Hence, the only tool that can be tested against *DaRLing* is the is the part of Clipper providing the Datalog rewriting of an OMQ (Clipper-Rew).

Moreover, we would like to stress out that *DaRLing* is not based on I-DLV (the grounder of DLV2), but we simply used I-DLV to evaluate the output of *DaRLing*. Note also that DLV2 is much faster than DLV, hence any comparison between *DaRLing*+I-DLV and DReW or Clipper would be unfair.

Concerning the OWL 2 QL profile, Presto [65] produces Datalog rewritings of polynomial size, whereas QuOnto [2] and Requiem [62] produce a union of conjunctive queries (i.e., a set of Datalog rules all having predicate *ans* in their head) of exponential size in the worst case.

Rewriting techniques are also used for other semantic services besides the OMQA. Among these, MASTRO [26] and Ontop [25] are open-source tools for *Ontology-Based*

Data Access (OBDA) [71] in which the ontology lies in the QL fragment of OWL 2. Ontop has its roots in MASTRO and is implemented through a query rewriting technique which avoids materializing triples. Kontchakov et al. [47] address the OBDA problem over both OWL 2 QL and OWL 2 EL with a combined approach, which embeds the information provided by the ontology into the data and uses query rewriting to eliminate spurious answers. Eventually, Stefanoni et al. [68] propose an approach which produces Datalog rewritings of polynomial size for performing OBDA over OWL 2 EL.

Among the systems dedicated to ontological query answering in the context of existential rules there are also Graal [15], DLV$^{\exists}$ [51] and VLog [27]. These systems are not ad-hoc for OWL 2 RL and typically capture more expressive ontologies and then support features such as the *skolem* and the *restricted (standard) chase* for reasoning over existential rules. Finally, Carral et al. [28] propose a combined approach for query answering over the DL Horn-$\mathcal{ALCHOIQ}$.

Regarding the approach presented in Part III of the thesis, not much progress has been made in that area before. The proposed framework is in fact completely new and does not exist, as far as we know, any tool computing plans for evaluating Datalog programs in a pre-processing phase.

Chapter 10

Combining the *DaRLing* rewriter and the planner

Although the two systems presented in Part II and III are designed and tested independently of each other, they can both be placed in the context of the OMQA over large-scale OWL 2 ontologies. Part of our future plans is certainly the design of a single system which, once the knowledge base has been translated into Datalog, allows us to take advantage of a planning in the evaluation of the rewritten programs. In this regard, as a first step towards merging the two systems, we conducted an experimental evaluation aimed at testing the behavior of the planner when it takes *DaRLing* rewritings as input. This chapter describes the set-up of this experimental activity in detail and the results are illustrated and discussed.

For these experiments we have chosen the intersection of the benchmarks used in Chapters 5 and 8: LUBM (the fragment falling into the OWL 2 RL profile), Stock Exchange and Vicodì. Datasets and queries are the same as those used in Chapter 8, and the customization policy applied to the planner for the three benchmarks is the same as described in Section 8.2. Ontologies and queries have been translated into Datalog via the *DaRLing* system.

As in Chapter 8 we performed two different executions: (*i*) a classical execution of I-DLV, and (*ii*) an execution in which I-DLV is forced—by means of annotations—to follow the evaluation plan precomputed by the planner. The experiments have been performed on a NUMA Linux machine equipped with two 2.8 GHz AMD Opteron 6320 processors and 128GB RAM.

The results of our experiments are summarized in Table 10.1, Figure 10-1 and Table 10.2. Note that only results related to the tests on LUBM and Stock Exchange are reported. In fact, on Vicodì, it emerged that the annotated programs resulting from the application of the planner on the *DaRLing* rewritings are identical to those resulting from the application of the planner on the Clipper rewritings with respect to all the queries. In other words, the planner's output is the same on *DaRLing*'s and Clipper's rewritings for Vicodì. For this reason it makes no sense to repeat the experiments for this benchmark, and we can take the results obtained in Table 8.2 for good.

We reported in Table 10.1 I-DLV performance in terms of average running time and memory usage computed over all considered databases and per each query of LUBM and Stock Exchange. Columns 2 and 3 refer to the classical I-DLV execution, whereas columns 4 and 5 to the execution where I-DLV is forced to follow the planner's output via annotations. In the 6th column, we reported the time spent to compute the optimal plan, and in the 7th and 8th column, respectively the memory and time savings computed as difference of the corresponding fields. The table reports also some aggregated data per benchmark: the average/maximum saved memory/time and the number of queries where an improvement in terms of saved memory/time has been obtained. Beyond that, Figure 10-1 shows the plots of the average running time and memory usage over all considered databases for LUBM (the largest benchmark). Finally, in Table 10.2 are given the maximum peak of memory and the sum of execution times computed over all databases and queries—along with the corresponding profits—for both the tested versions of I-DLV and for both benchmarks.

As it can be seen, we obtained a significant saving of time on LUBM where the planner

Query	*No Planner*		*Planner*			Saved Memory	Saved Time
	Time	Memory	Time	Memory	Planning Time		
				LUBM			
q01	1,846.45	17.68	1,243.83	17.68	0.01	0.0	602.8
q02	1,864.95	18.17	1,244.68	18.17	0.03	0.0	620.2
q03	1,523.14	17.27	1,113.97	17.27	0.01	0.0	409.2
q04	1,154.64	17.26	1,045.16	17.26	0.00	0.0	109.5
q05	1,754.23	17.26	1,141.29	17.27	0.01	0.0	612.9
q06	1,612.32	17.29	1,141.52	17.27	0.00	0.0	470.8
q07	1,260.88	17.26	1,112.82	17.27	0.00	0.0	148.1
q08	1,264.76	17.27	1,112.33	17.26	0.03	0.0	152.4
q09	1,906.18	19.76	1,336.55	19.76	0.02	0.0	569.6
q10	1,100.80	17.27	1,103.97	17.26	0.01	0.0	-3.2
q11	1,077.63	17.28	1,076.80	17.26	0.01	0.0	0.8
q12	1,085.58	17.27	1,083.54	17.27	0.00	0.0	2.0
q13	1,197.43	17.59	1,159.96	17.59	0.01	0.0	37.5
q14	1,143.86	17.27	1,142.67	17.26	0.00	0.0	1.2
Saved Memory		Average: 0.01		Maximum: 0.02		Improvements: 9/14	
Saved Time		Average: 266.71		Maximum: 620.23		Improvements: 13/14	
				Stock Exchange			
q01	0.64	10.03	0.78	12.24	0.00	-2.2	-0.1
q02	0.97	22.78	0.93	22.24	0.01	0.5	0.0
q03	1.65	41.06	1.59	40.68	0.00	0.3	0.1
q04	1.17	29.22	1.17	29.94	0.03	-0.7	0.0
q05	3.26	46.52	4.11	30.93	0.09	15.6	-0.9
Saved Memory		Average: 2.70		Maximum: 15.59		Improvements: 3/5	
Saved Time		Average: -0.18		Maximum: 0.06		Improvements: 3/5	

Table 10.1: Planner performance over *DaRLing*'s rewritings in terms of average running time and memory usage of I-DLV with and without planner, computed over all considered databases per each query of LUBM and Stock Exchange. Time is in seconds, memory is in GB for LUBM and in MB for Stock Exchange.

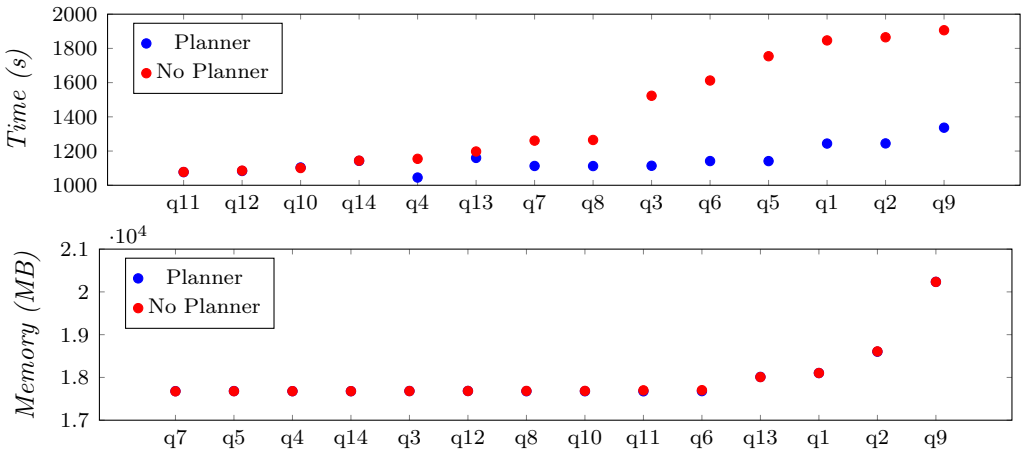

Figure 10-1: Plots of the average running time and memory usage of I-DLV (with and without planner) over all considered databases per each query of LUBM. Queries are ordered by increasing values w.r.t. the No-Planner execution.

saves an average of 266.71 seconds (about 4 and a half minutes) per query compared to the no-planner version. The greatest saving in terms of time is on query 2 where there is a saving of 620.2 seconds (about 10 minutes and 33% less). Although there is no worsening on any query in terms of memory, there are no notable improvements either. This trend could be due to the narrowing of the benchmarck to its OWL 2 RL version. In fact, as just noted in Chapter 8, planner performance improve as memory consumption increases with respect to I-DLV standard execution and, as it is easy to see, in this fragment the overall memory consumption has been drastically reduced (the memory peak on the RL fragment of LUBM is 40 gigabytes against the 80 gigabytes recorded in its full version). Despite this, execution times turn out to be better on almost all queries and, as shown in Figure 10-1, this improvement tends to grow as standard execution times increase.

Concerning Stock Exchange, although this is not a data intensive domain, I-DLV can benefit by the planner as well. Indeed, in query 5 there is a saving of 15.6 megabytes of memory, whereas a worsening is recorded only in queries 1 and 4 (of 2.2 and 0.6 megabytes, respectively).

Domain	*Peak of Memory*			*Sum of Times*		
	No Planner	*Planner*	Profit (%)	*No Planner*	*Planner*	Profit (%)
LUBM	43,217.70	43,211.30	0.0%	79,172.20	64,236.37	18.9%
Stock Exchange	76.20	73.30	3.8%	39.19	40.27	-2.7%

Table 10.2: Statistics on LUBM and Stock Exchange: the maximum peak of memory and the total sum of execution times computed over all databases and queries, along with the corresponding profits. Time is in seconds, memory is in MB.

Finally, Table 10.2 shows that the peak of memory when the planner is used is lower (about 3,8%) than the one obtained using the standard version of I-DLV for Stock Exchange, and it is slightly lower also for LUBM. Regarding times, although we experimented a small worsening for Stock Exchange (2.7%), we observe a 18.9% improvement in our large-scale benchmark.

Chapter 11

Discussion and future work

In this thesis we addressed the OMQA problem over OWL 2 RL ontologies in large-scale scenarios via Datalog rewriting.

Primarily, we presented *DaRLing*,[1] a Datalog-based rewriter for OWL 2 RL ontological reasoning under SPARQL queries. To demonstrate its practical applicability, we have designed and conducted an experimental evaluation based on two working hypotheses, which have been confirmed. The first release of *DaRLing* demonstrates to produce more general rewritings equivalent or sightly differing from the ones generated by Clipper, the closer open-source competitor of *DaRLing*. As additional feature, *DaRLing* can be used for transparently handling the `owl:sameAs` property independently from the Datalog reasoner at hand albeit requiring extra work due to the intrinsic need of computing the transitive closure. Such costs are strictly dependent from the ontology at hand; our experimentation in an unfriendly setting of a large ontology such as DBpedia proved a not taken for granted applicability of the approach.

Then, we introduced a *planner*[2] for the evaluation of Datalog programs. The planner has been conceived to be applied to OMQA contexts, where often, in case of large databases,

[1] https://demacs-unical.github.io/DaRLing/
[2] https://www.mat.unical.it/perri/iclp2019.zip

standard approaches are not convenient/applicable due to memory consumption. It relies on an ASP program that computes the plan, intended as an indexing schema for the database together with a body-ordering for each rule in the program. The computed plan minimizes the overall cost (in term of memory consumption) of indices; moreover, the usage of the plan with the DLV system allows to further reduce memory usage since some expensive internal optimizations of DLV can be disabled. Results of the experiments conducted on popular ontological benchmarks confirm the effectiveness of the approach. In case of reasoners with a server-like behavior, such as DLV2-SERVER and OWL2DLV, evaluation plans play an extremely important role, and the advantage of precomputing an evaluation plan is even more evident. Indeed, when the ontology is known in advance, it is possible to determine "offline" the optimal plan, and therefore further improve the reasoning phase with respect to both time and memory.

With reference to point (*i*) of the desiderata expressed in Chapter 1 (i.e., that concerning the regular maintenance of our OWL 2 RL system), among our future plans is the development of a freely available customizable *DaRLing*-based SPARQL endpoint for OWL 2 RL ontological reasoning complying with the W3C Recommendations. To this end, the next steps are:

(*i*) extending the rewriting to enable the meta-reasoning, namely SPARQL queries where variables may range also over the given schema;

(*ii*) handling `owl:sameAs` also over concept and role names;

(*iii*) enriching the set of supported datatypes;

(*iv*) extending the notion of *evaluation plan* to Datalog$^{\neg_s,\neq}$;

(*v*) designing a system that integrates both systems—rewriter and planner—to automate rewriting of the ontologies and planning of evaluation of the rewritings over large databases.

In this regard, we already started a first experimental phase in order to get an idea of how the planner behaves on *DaRLing*'s outputs. The results obtained in Chapter 10 confirm a good overall behavior of the planner on the *DaRLing* rewritings and suggest a combined use of the two systems especially in large-scale scenarios (those for which the planner was conceived).

Bibliography

[1] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley. 27

[2] Acciarri, A., Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., Palmieri, M., and Rosati, R. (2005). Quonto: Querying ontologies. In Veloso, M. M. and Kambhampati, S., editors, *Proceedings of AAAI'05*, pages 1670–1671. AAAI Press / The MIT Press. 18, 115

[3] Ahmetaj, S. (2020). Rewriting approaches for ontology-mediated query answering. *Künstliche Intell.*, 34(4):523–526. 52

[4] Allocca, C., Alviano, M., Calimeri, F., Costabile, R., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Leone, N., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., and Zangari, J. (2018). Reasoning over ontologies with DLV. In *Proceedings of IC3K'18*, volume 1222 of *Communications in Computer and Information Science*, pages 114–136. Springer. 76

[5] Allocca, C., Calimeri, F., Civili, C., Costabile, R., Cuteri, B., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., and Zangari, J. (2019a). Large-scale reasoning on expressive horn ontologies. In *Proceedings of Datalog 2.0*, volume 2368 of *CEUR Workshop Proceedings*, pages 10–21. CEUR-WS.org. 18, 114

[6] Allocca, C., Calimeri, F., Costabile, R., Fiorentino, A., Leone, N., Manna, M., Perri, S., and Zangari, J. (2019b). An asp-based approach for optimizing DLV evaluation. In *Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, June 19-21, 2019*, volume 2396 of *CEUR Workshop Proceedings*. CEUR-WS.org. 23

[7] Allocca, C., Costabile, R., Fiorentino, A., Perri, S., and Zangari, J. (2019c). Memory-saving evaluation plans for datalog. In *Proceedings of JELIA'19*, volume 11468 of *Lecture Notes in Computer Science*, pages 453–461. Springer. 20, 23, 84

[8] Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., and Zangari, J. (2017). The ASP system DLV2. In *Proceedings of LPNMR'17*, volume 10377 of *LNCS*, pages 215–221. 83

[9] Alviano, M., Faber, W., Greco, G., and Leone, N. (2012). Magic sets for disjunctive datalog programs. *Artificial Intelligence*, 187:156–192. 105

[10] Alviano, M., Leone, N., Veltri, P., and Zangari, J. (2019). Enhancing magic sets with an application to ontological reasoning. *Theory and Practice of Logic Programming*, 19(5-6):654–670. 78

[11] Alviano, M. and Manna, M. (2020). Large-scale ontological reasoning via datalog. *CoRR*, abs/2003.09698. 21

[12] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. G. (2007). DBpedia: A nucleus for a web of open data. In *Proceedings of ISWC'07*, volume 4825 of *LNCS*, pages 722–735. Springer. 76

[13] Baader, F., Brandt, S., and Lutz, C. (2005). Pushing the EL envelope. In *Proceedings of IJCAI'05*, pages 364–369. Professional Book Center. 46

[14] Baader, F., Horrocks, I., and Sattler, U. (2008). Description logics. In *Handbook*

of Knowledge Representation, volume 3 of *Foundations of Artificial Intelligence*, pages 135–179. Elsevier. 45

[15] Baget, J., Leclère, M., Mugnier, M., Rocher, S., and Sipieter, C. (2015). Graal: A toolkit for query answering with existential rules. In *Proceedings of RuleML'15*, volume 9202 of *LNCS*, pages 328–344. Springer. 18, 116

[16] Bernstein, A., Hendler, J. A., and Noy, N. F. (2016). A new look at the semantic web. *Commun. ACM*, 59(9):35–37. 84

[17] Bienvenu, M. (2016). Ontology-mediated query answering: Harnessing knowledge to get more from data. In *Proceedings of IJCAI'16*, pages 4058–4061. IJCAI/AAAI Press. 17

[18] Bienvenu, M. (2020). A short survey on inconsistency handling in ontology-mediated query answering. *Künstliche Intell.*, 34(4):443–451. 17

[19] Bonatti, P. A., Calimeri, F., Leone, N., and Ricca, F. (2010). Answer set programming. In *25 Years GULP*, volume 6125 of *Lecture Notes in Computer Science*, pages 159–182. Springer. 25

[20] Borgida, A. (2018). Description logics. In *Encyclopedia of Database Systems (2nd ed.)*. Springer. 45

[21] Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., and Schaub, T. (2020). Asp-core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309. 25

[22] Calimeri, F., Fuscà, D., Perri, S., and Zangari, J. (2016). I-DLV: The new intelligent grounder of DLV. In *Proceedings of AI*IA'16*, volume 10037 of *LNCS*, pages 192–207. Springer. 89

[23] Calimeri, F., Fuscà, D., Perri, S., and Zangari, J. (2017). I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20. 22, 76, 83, 104

[24] Calimeri, F., Perri, S., and Zangari, J. (2019). Optimizing answer set computation via heuristic-based decomposition. *Theory and Practice of Logic Programming*, 19(4):603–628. 83

[25] Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., and Xiao, G. (2017). Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487. 115

[26] Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R., Ruzzi, M., and Savo, D. F. (2011). The MASTRO system for ontology-based data access. *Semantic Web*, 2(1):43–53. 115

[27] Carral, D., Dragoste, I., González, L., Jacobs, C. J. H., Krötzsch, M., and Urbani, J. (2019). Vlog: A rule engine for knowledge graphs. In *Proceedings of ISWC'19*, volume 11779 of *LNCS*, pages 19–35. Springer. 18, 116

[28] Carral, D., Dragoste, I., and Krötzsch, M. (2018). The combined approach to query answering in Horn-ALCHOIQ. In *Proceedings of KR'18*, pages 339–348. AAAI Press. 116

[29] Ceri, S., Gottlob, G., and Tanca, L. (1989). What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166. 39

[30] Eiter, T., Ortiz, M., Simkus, M., Tran, T., and Xiao, G. (2012). Query rewriting for Horn-SHIQ plus rules. In *Proceedings of AAAI'12*. AAAI Press. 18, 114

[31] Faber, W., Leone, N., and Ricca, F. (2008). Answer set programming. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc. 25

[32] Faruqui, R. U. and MacCaull, W. (2012). O wl O nt DB: A scalable reasoning system for OWL 2 RL ontologies with large aboxes. In *Proceedings of FHIES'12*, volume 7789 of *LNCS*, pages 105–123. Springer. 18, 114

[33] Fiorentino, A., Leone, N., Manna, M., Perri, S., and Zangari, J. (2019). Precomputing datalog evaluation plans in large-scale scenarios. *Theory and Practice of Logic Programming*, 19(5-6):1073–1089. 20, 23, 84

[34] Fiorentino, A., Zangari, J., and Manna, M. (2020). Darling: A datalog rewriter for OWL 2 RL ontological reasoning under SPARQL queries. *Theory and Practice of Logic Programming*, 20(6):958–973. 20, 23, 55

[35] Gebser, M., Schaub, T., and Thiele, S. (2007). Gringo : A new grounder for answer set programming. In *Proocedings of LPNMR'07*, pages 266–271. 89

[36] Gelfond, M. and Lifschitz, V. (1990). Logic programs with classical negation. In *ICLP 1990*, pages 579–597. MIT Press. 39

[37] Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386. 25, 33

[38] Grau, B. C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P. F., and Sattler, U. (2008). OWL 2: The next step for OWL. *Journal of Web Semantics*, 6(4):309–322. 45

[39] Grau, B. C., Motik, B., Stoilos, G., and Horrocks, I. (2012). Completeness guarantees for incomplete ontology reasoners: Theory and practice. *Journal of Artificial Intelligence Research*, 43:419–476. 104

[40] Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. (2003). Description logic programs: combining logic programs with description logic. In *Proceedings of WWW'03*, pages 48–57. ACM. 46

[41] Guo, Y., Pan, Z., and Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182. 76

[42] Harris, S. and Seaborne, A. (2018). *SPARQL 1.1 Query Language*. W3C Recommendation. World Wide Web Consortium. 18, 20

[43] Horridge, M. and Bechhofer, S. (2009). The OWL API: A java API for working with OWL 2 ontologies. In *Proceedings of OWLED'09*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org. 56

[44] Kazakov, Y. (2009a). Consequence-driven reasoning for horn SHIQ ontologies. In *Proceedings of IJCAI'09*, pages 2040–2045. 21, 60, 63

[45] Kazakov, Y. (2009b). Consequence-driven reasoning for horn SHIQ ontologies. In *Proceedings of DL'09*. 60

[46] Knorr, M. and Hitzler, P. (2014). Description logics. In *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 659–678. Elsevier. 45

[47] Kontchakov, R., Lutz, C., Toman, D., Wolter, F., and Zakharyaschev, M. (2011). The combined approach to ontology-based data access. In *Proceedings of IJCAI'11*, pages 2656–2661. IJCAI/AAAI. 116

[48] Krötzsch, M., Mehdi, A., and Rudolph, S. (2010). Orel: Database-driven reasoning for OWL 2 profiles. In *Proceedings of DL'10*, volume 573 of *CEUR Workshop Proceedings*. CEUR-WS.org. 18, 113

[49] Leone, N., Allocca, C., Alviano, M., Calimeri, F., Civili, C., Costabile, R., Cuteri, B., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., and Zangari, J. (2019a). Large scale DLV: preliminary results. In *Proceedings of CILC'19*. 84

[50] Leone, N., Allocca, C., Alviano, M., Calimeri, F., Civili, C., Costabile, R., Fiorentino, A., Fuscà, D., Germano, S., Laboccetta, G., Cuteri, B., Manna, M., Perri, S., Reale, K., Ricca, F., Veltri, P., and Zangari, J. (2019b). Enhancing DLV for large-scale reasoning. In *Proceedings of LPNMR'19*, volume 11481 of *LNCS*, pages 312–325. Springer. 77, 84, 114

[51] Leone, N., Manna, M., Terracina, G., and Veltri, P. (2019c). Fast query answering over existential rules. *ACM Transactions on Computational Logic*, 20(2):12:1–12:48. 76, 116

[52] Leone, N., Perri, S., and Scarcello, F. (2001). Improving ASP instantiators by join-ordering methods. In *Proceedings of LPNMR'01*, volume 2173 of *Lecture Notes in Computer Science*, pages 280–294. Springer. 89

[53] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562. 36

[54] Lifschitz, V. (1999). Action languages, answer sets, and planning. In *The Logic Programming Paradigm*, Artificial Intelligence, pages 357–373. Springer. 25

[55] Lutz, C., Seylan, I., Toman, D., and Wolter, F. (2013). The combined approach to OBDA: taming role hierarchies using filters. In *Proceedings of ISWC'13*, volume 8218 of *LNCS*, pages 314–330. Springer. 103

[56] Mora, J. and Corcho, Ó. (2013). Towards a systematic benchmarking of ontology-based query rewriting systems. In *Proceedings of ISWC'13*, volume 8219 of *LNCS*, pages 376–391. Springer. 76

[57] Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., and Lutz, C. (2012). *OWL 2 Web Ontology Language Profiles (Second Edition)*. W3C Recommendation. World Wide Web Consortium. 18, 64

[58] Motik, B., Nenov, Y., Piro, R. E. F., and Horrocks, I. (2015). Handling owl: sameas via rewriting. In *Proceedings of AAAI'15*, pages 231–237. AAAI Press. 81

[59] Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., and Banerjee, J. (2015). Rdfox: A highly-scalable RDF store. In *Proceedings of ISWC'15*, volume 9367 of *LNCS*, pages 3–20. Springer. 18, 114

[60] Ortiz, M. (2013). Ontology based query answering: The story so far. In *Proceedings of AMW'13*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org. 17

[61] Papadimitriou, C. H. (1994). *Computational complexity*. Addison-Wesley. 37, 38

[62] Pérez-Urbina, H., Motik, B., and Horrocks, I. (2010). Tractable query answering and rewriting under description logic constraints. *Journal of Applied Logic*, 8(2):186–209. 18, 115

[63] Perri, S., Scarcello, F., Catalano, G., and Leone, N. (2007). Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):195–228. 89

[64] Rich, E. (2008). *Automata, computability and complexity: theory and applications*. Pearson Prentice Hall Upper Saddle River. 38

[65] Rosati, R. and Almatelli, A. (2010). Improving query answering over DL-Lite ontologies. In *Proceedings of KR'10*. AAAI Press. 18, 46, 115

[66] Sirin, E. and Parsia, B. (2007). SPARQL-DL: SPARQL query for OWL-DL. In *Proceedings of OWLED'07*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org. 20

[67] Smith, M. K., Welty, C., and McGuinness, D. L. (2004). *OWL Web Ontology Language Guide*. W3C Recommendation. World Wide Web Consortium. 17

[68] Stefanoni, G., Motik, B., and Horrocks, I. (2012). Small datalog query rewritings for EL. In *Proceedings of DL'12*, volume 846 of *CEUR Workshop Proceedings*. 116

[69] Stefanoni, G., Motik, B., Krötzsch, M., and Rudolph, S. (2014). The complexity of answering conjunctive and navigational queries over OWL 2 EL knowledge bases. *J. Artif. Intell. Res.*, 51:645–705. 46

[70] W3C OWL Working Group (2012). *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C Recommendation. World Wide Web Consortium. 17

[71] Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., and Zakharyaschev, M. (2018). Ontology-based data access: A survey. In *Proceedings of IJCAI'18*, pages 5511–5519. ijcai.org. 116

[72] Xiao, G., Eiter, T., and Heymans, S. (2012). The drew system for nonmonotonic dl-programs. In *Proceedings of CSWS'12*, pages 383–390. Springer. 18, 114