HAL
open science

Querying RDF(S) with Regular Expressions

Faisal Alkhateeb

To cite this version:

Faisal Alkhateeb. Querying RDF(S) with Regular Expressions. Computer Science [cs]. Université Joseph-Fourier - Grenoble I, 2008. English. NNT : . tel-00293206v1

HAL Id: tel-00293206

https://theses.hal.science/tel-00293206v1

Submitted on 3 Jul 2008 (v1), last revised 7 Jul 2008 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée à

l'Université Joseph Fourier - Grenoble 1

pour obtenir le grade de

DOCTEUR

spécialité

Informatique

intitulée

Querying RDF(S) with Regular Expressions

présentée et soutenue publiquement le 30 juin 2008 par

Faisal Alkhateeb

devant le jury composé de:

Jean-François Baget	**Co-encadrant**
Vassilis Christophides	**Rapporteur**
Jérôme Euzenat	**Directeur de thèse**
Ollivier Haemmerlé	**Rapporteur**
Amedeo Napoli	**Examinateur**
Marie-Christine Rousset	**Présidente**

Dedicated to my father and my wife Ebtesam Abushareah

Abstract

RDF is a knowledge representation language dedicated to the annotation of resources within the Semantic Web. Though RDF itself can be used as a query language for an RDF knowledge base (using RDF semantic consequence), the need for added expressivity in queries has led to define the SPARQL query language. SPARQL queries are defined on top of graph patterns that are basically RDF graphs with variables. SPARQL queries remain limited as they do not allow queries with unbounded sequences of relations (e.g. "does there exist a trip from town A to town B using only trains or buses?"). We show that it is possible to extend the RDF syntax and semantics defining the PRDF language (for Path RDF) such that SPARQL can overcome this limitation by simply replacing the basic graph patterns with PRDF graphs, effectively mixing RDF reasoning with database-inspired regular paths. We further extend PRDF to CPRDF (for Constrained Path RDF) to allow expressing constraints on the nodes of traversed paths (e.g. "Moreover, one of the correspondences must provide a wireless connection."). We have provided sound and complete algorithms for answering queries (the query is a PRDF or a CPRDF graph, the knowledge base is an RDF graph) based upon a kind of graph homomorphism, along with a detailed complexity analysis. Finally, we use PRDF or CPRDF graphs to generalize SPARQL graph patterns, defining the PSPARQL and CPSPARQL extensions, and provide experimental tests using a complete implementation of these two query languages.

Keywords: Knowledge Representation Languages, RDF(S), Querying Semantic Web, SPARQL, Graph Homomorphism, Regular Languages, Regular Expressions, SPARQL Extensions, PRDF, PSPARQL, CPRDF, CPSPARQL.

Résumé

RDF est un langage de représentation des connaissances dédié à l'annotation des ressources dans le Web Sémantique. Bien que RDF peut être lui-même utilisé comme un langage de requêtes pour interroger une base de connaissances RDF (utilisant la conséquence RDF), la nécessité d'ajouter plus d'expressivité dans les requêtes a conduit à définir le langage de requêtes SPARQL. Les requêtes SPARQL sont définies à partir des patrons de graphes qui sont fondamentalement des graphes RDF avec des variables. Les requêtes SPARQL restent limitées car elles ne permettent pas d'exprimer des requêtes avec une séquence non-bornée de relations (par exemple, "Existe-t-il un itinéraire d'une ville A à une ville B qui n'utilise que les trains ou les bus?"). Nous montrons qu'il est possible d'étendre la syntaxe et la sémantique de RDF, définissant le langage PRDF (pour Path RDF) afin que SPARQL puisse surmonter cette limitation en remplaçant simplement les patrons de graphes basiques par des graphes PRDF. Nous étendons aussi PRDF à CPRDF (pour Constrained Path RDF) permettant d'exprimer des contraintes sur les sommets des chemins traversés (par exemple, "En outre, l'une des correspondances doit fournir une connexion sans fil."). Nous avons fourni des algorithmes corrects et complets pour répondre aux requêtes (la requête est un graphe PRDF ou CPRDF, la base de connaissances est un graphe RDF) basés sur un homomorphisme particulier, ainsi qu'une analyse détaillée de la complexité. Enfin, nous utilisons les graphes PRDF ou CPRDF pour généraliser les requêtes SPARQL, définissant les extensions PSPARQL et CPSPARQL, et fournissons des tests expérimentaux en utilisant une implémentation complète de ces deux langages.

Mots-Clés: Langage de Représentation des Connaissances, RDF(S), Web Sémantique, Langages de Requêtes, SPARQL, Homomorphisme de Graphes, Langages Réguliers, Expressions de Chemins, Expressions Régulières, Extensions de SPARQL , PRDF, PSPARQL, CPRDF, CPSPARQL.

Acknowledgments

I would like to thank first my thesis supervisor Jérôme Euzenat for accepting me as a member in his team. I am greatly indebted to him for his guidance and all kinds of supports throughout my research period. His perception and availability allowed me to get easily feedback and directions which were the necessary elements for success at different crucial points. To my co-advisor Jean-François Baget, I say thank you for your directions in the first stages of my research formation. I would like also to thank Professor Vassilis Christophides for his invaluable comments and questions during the reporting period that undoubtably helped improving the quality of the presentation in some parts; Professor Ollivier Haemmerlé, which is of my pleasure, to be a reporter of my thesis; Amedeo Napoli and Professor Marie-Christine Rousset for accepting to be members in the jury. I would like to thank my friends in the EXMO team and in other teams: Antoine Zimmermann, Sébastien Laborie, Arun Sharma, Seungkeun Lee, Jason Jung, Jérôme Pierson, Jérôme David, Chan Leduc, Nizar Ghoula and Julien Burlet. I would like to give a special thank to Gilles Kuntz that helped us to have necessary documents for my daughter Ayah to enter the France.

Special thanks to my family for supporting me during the study period: Ebtesam Abushareah, Deema Alkhateeb, Ayah Alkhateeb, Hebah Alkhateeb and my father that encouraged me for having the success. Finally, I would like to thank Yarmouk University for giving me the opportunity and the scholarship to continue my graduate studies.

Table of Contents

1 Introduction **1**
1.1 Motivations and Objectives 1
1.2 Main Contributions 4
1.3 Thesis outline 6

I Background **9**

2 The RDF Language **11**
2.1 RDF Syntax 12
2.2 Simple RDF Semantics 15
2.3 Inference Mechanism 18
2.4 RDF Entailment: Definition and Complexity 20
2.5 RDF vs. Graph Database Models 20
2.6 Conclusion 22

3 Querying RDF Graphs **23**
3.1 (Semi)-Structured Query Languages 24
3.2 RDF Query Languages 26
3.3 The SPARQL Query Language 27
3.4 Extensions to SPARQL 33
3.5 Work on SPARQL 35
3.6 Comparison with other Query Languages 36
3.7 Conclusion 37

II Research Work **39**

4 A General Graph Framework with Paths **41**

4.1 PRDF Syntax 42
4.2 PRDF Semantics 47
4.3 Querying RDF with PRDF Graphs 51
4.4 Containment of PRDF Queries 58
4.5 Conclusion 64

5 The PSPARQL Query Language 65
5.1 PSPARQL Syntax 66
5.2 Formal Semantics of PSPARQL 67
5.3 Translation from PSPARQL to SPARQL 70
5.4 Algorithms for PSPARQL Query Evaluation 72
5.5 Complexity of Evaluating PSPARQL Graph Patterns 82
5.6 Conclusion 83

6 Constrained Paths in SPARQL 85
6.1 CPSPARQL by examples 86
6.2 CPRDF: Constrained Paths in RDF 89
6.3 The CPSPARQL Query Language 100
6.4 Summary 103

7 Other Possible Extensions 105
7.1 Path Variables 105
7.2 Similarity-Based Path Matching 110
7.3 Nested Queries 113
7.4 Extending Constraints in CPSPARQL 114
7.5 Conclusion 115

8 Querying RDFS Graphs 117
8.1 RDF(S) 119
8.2 RDF(S) Closure and Query Answering 121
8.3 RDF(S) Entailment and Query Rewriting 123
8.4 Conclusion 129

9 Implementation and Experiments 131
9.1 Implementation 131
9.2 Experiments 134
9.3 Conclusion 148

10 Conclusion **149**
10.1 Summary 149
10.2 Future Directions 150

III Appendices **165**

A CPSPARQL Grammar **167**

B Résumé étendu **173**
B.1 Motivations et objectifs 174
B.2 Résumé des contributions 176
B.3 Organisation de la thèse 178
B.4 Conclusions 180

Introduction 1

Contents

1.1 Motivations and Objectives 1
1.2 Main Contributions 4
1.3 Thesis outline 6

The world wide web (or simply the web) has become the first source of knowledge for all life domains. It can be seen as an extensive information system that allows changing the resources as well as documents. The semantic web is an evolving extension of the Web aiming at giving well defined form and semantics to the web resources (*e.g.* content of an HTML web page) [Berners-Lee *et al.*, 2001]. In particular, query answering is an essential functionality of any information system, and so of the semantic web. This thesis studies the current query mechanisms for the semantic web and studies the problem of supporting path expressions and path retrievals in the semantic web knowledge bases. The motivation of this study arises from limitations in the current query languages for supporting and extracting paths from knowledge bases.

1.1 Motivations and Objectives

RDF (Resource Description Framework [Miller *et al.*, 2004]) is a knowledge representation language dedicated to the annotation of documents and more generally of resources within the semantic web. In its abstract syntax, an RDF document is a set of triples *(subject, predicate, object)*, that can be represented by a directed labeled graph (hence the name, RDF graph). The language is provided with a

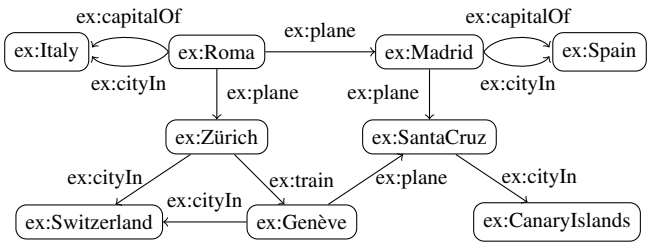

Figure 1.1: An RDF graph.

model-theoretic semantics [Hayes, 2004], that defines the notion of consequence between two RDF graphs, *i.e.,* does an RDF graph G entails an RDF graph H (RDF ENTAILMENT).

Example 1.1.1 *The RDF graph of Figure 1.1, for example, consists of a set of arcs relating cities with transportation means such that each arc or triple of the form* $\langle C_1, t, C_2 \rangle$ *indicates that there exists a transportation mean from* C_1 *to* C_2 *(*C_2 *is directly reachable from* C_1 *by* t*).*

Nowadays, more resources are annotated via RDF due to its simple data model, formal semantics, and a sound and complete inference mechanism. A query language that provides a range of querying paradigms is therefore needed. Though RDF was initially designed as a knowledge representation language, it can be used for querying RDF graphs (the knowledge base and the query are two RDF graphs). Answers to an RDF query over an RDF knowledge base are determined by consequence, and can be computed using a particular *map* (a mapping from terms of the query to terms of the knowledge base that preserves constants and graph structure), a *graph homomorphism* [Gutierrez *et al.*, 2004; Baget, 2005], which is known as projection in conceptual graphs [Mugnier and Chein, 1992]. More precisely, the answer to a query Q relies on calculating the set of possible homomorphisms from Q into the RDF graph representing the knowledge base.

The need for added expressivity to RDF query has led to define SPARQL [Prud'hommeaux and Seaborne, 2008], a W3C recommendation developed in order to query an RDF knowledge base (*cf.* [Haase *et al.*, 2004] for a comparison of query languages for RDF). The heart of a SPARQL query, the graph pattern, is an RDF graph (and more precisely a Generalized RDF graph allowing variables as predicates, as done in [Horst, 2004]). The maps that are used to compute answers to a graph pattern query in an RDF knowledge base are exploited by [Perez *et al.*,

Figure 1.2: A SPARQL graph pattern.

2006] to define answers to the more complex, more expressive SPARQL queries (using, for example, disjunctions or functional constraints).

Example 1.1.2 *SPARQL graph patterns allows to match a query graph against an actual RDF graph. Figure 1.2 presents such a graph pattern. It can be used for finding the names of cities and countries connected to Roma. If this pattern is used in a SPARQL query against the graph G of Figure 1.1, it will return "Madrid" with country "Spain" and transportation mean "plane", and "Zürich" with country "Switzerland" and transportation mean "plane".*

Unfortunately, most of the query languages that are based upon RDF semantics, like SPARQL, lack the ability of expressing and retrieving paths, which is necessary for many applications. For example, if one wants to check if there exists a trip (not necessary direct) from one city to another (see Example 1.1.3).

Another approach, that has been successfully used in databases [Consens and Mendelzon, 1990; Cruz *et al.*, 1988; Mendelzon and Wood, 1995; Tarjan, 1981; Yannakakis, 1990] but little in the context of the semantic web, uses path queries, *i.e.,* regular expressions, for finding regular paths in a database graph. The answer to a path query R over a database graph G, is the set of all pairs of nodes in G satisfying the language denoted by R, *i.e.,* all pairs connected by a directed path such that the concatenation of the labels of the arcs along the path forms a word that belongs to the language denoted by R (see Example 1.1.3).

Example 1.1.3 *Assuming an RDF graph representing transportation network, like the graph G of Figure 1.1, the regular expression* $(\texttt{ex:train}|\texttt{ex:plane})^+$*, when used as a query, searches all pairs of nodes connected by paths with a sequence of train and plane relations, i.e., the reachable cities. Applied to node* `ex:Roma` *of G, it should match the paths leading to* `ex:Madrid`*,* `ex:SantaCruz`*,* `ex:Zürich` *and* `ex:Genève`*. This query, as it represents paths of unknown length, cannot be expressed in SPARQL. On the other hand, the graph of Figure 1.2, which represents a basic graph pattern of a SPARQL query, cannot be expressed by a regular expression.*

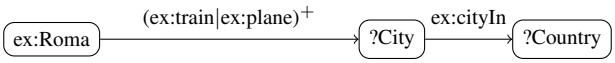

Figure 1.3: A graph pattern with regular expressions.

Both approaches are orthogonal, *i.e.,* some queries that can be expressed in one approach cannot be expressed in the other. As shown in Figure 1.2, a query whose homomorphic image in the database is not a path cannot be expressed by a regular expression, while RDF semantics does not allow expressing paths of undetermined length. Furthermore, regular expressions provide a simple way to capture additional information along paths that may not be provided by SPARQL graph patterns, but they are not powerful enough as a query language.

To overcome this limitation, an approach that combines the advantages of both SPARQL and path queries is herein investigated. This combined approach, in which the arcs of the graph patterns may be labeled with regular expression, supports path queries (see Figure 1.3).

1.2 Main Contributions

In order to formally define that language, we first introduce Path RDF (PRDF) as an extension of RDF in which arcs of the graphs can be labeled by regular expression patterns [Alkhateeb *et al.*, 2005; Alkhateeb *et al.*, 2007]. Because we want to ground the definition of our language on the semantics of RDF, and we want to leave the door open to further extensions, we define the semantics of PRDF on top of RDF semantics and we provide a sound and complete algorithm for checking if a PRDF graph is entailed by some RDF graph.

Example 1.2.1 *The PRDF graph of Figure 1.3 when used as a query finds the name of each city and its country such that the city is reachable from Roma by a sequence of trains and planes.*

PRDF graphs are then used to define a basic extension to SPARQL, called PSPARQL, that replaces RDF graph patterns used in SPARQL by PRDF graph patterns, *i.e.,* graph patterns with regular expression patterns. We present the syntax and the semantics of PSPARQL. We provide algorithms, which are sound and complete for evaluating PSPARQL graph patterns over RDF graphs.

Example 1.2.2 *The following PSPARQL query:*

```
SELECT ?City
WHERE {
        ex:Paris (ex:train|ex:plane)+ ?City .
        ?City ex:capitalOf ?Country .
      }
ORDER BY Asc(?City)
```

returns, in an increasing order, the set of capital cities reachable from Paris by a sequence of trains or planes.

For added expressivity to PSPARQL to allow specifying properties on the nodes that belong to a path defined by a regular expression, "for example all stops must be capital cities.", we have extended PRDF. More precisely, we have defined the CPRDF (for Constrained Path RDF) language that extends the syntax and the semantics of PRDF to handle constraints on paths.

Example 1.2.3 *The graph represented in figure 1.4, where const* =]`ALL ?Stop`] : {{`?Stop ex:capitalOf ?Country .`} `UNION` {`?Stop ex:population ?Pop . FILTER (?Pop > 200000)`}}*, is a CPRDF graph.*

We have also characterized answers to a query reduced to a CPRDF graph by a kind of graph homomorphism (a particular map). This property was sufficient to extend the PSPARQL query language to CPSPARQL, combining the expressiveness of both SPARQL and CPRDF.

Example 1.2.4 *The following CPSPARQL query:*

```
SELECT ?City
WHERE {
        CONSTRAINT const ]ALL ?Stop]: {{ ?Stop ex:capitalOf ?Country. }
                                          UNION
                                       { ?Stop ex:population ?Pop .
                                         FILTER (?Pop > 200000)
                                       }
                                      }
        ex:Paris (ex:train|ex:plane)+%const% ?City .
        ?City ex:capitalOf ?Country .
      }
```

whose graph pattern is the CPRDF graph of Example 1.2.3, could be used for finding the names of cities and countries such that each city is reachable from Roma by a path (a sequence of trains or planes) whose nodes are capital cities or have population size greater than 200000.

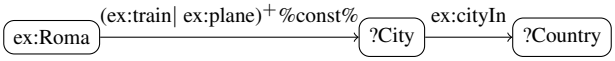

Figure 1.4: A graph pattern with constrained regular expressions.

We have implemented an evaluator for answering PSPARQL or CPSPARQL queries. The evaluator is provided with two main parsers:

– a parser for RDF graphs written in Turtle language, and

– a parser for queries written according to the CPSPARQL syntax, which is compatible with SPARQL syntax (see `http://psparql.inrialpes.fr`).

1.3 Thesis outline

To provide the necessary background, we begin in **Chapter 2** with an introduction to the RDF language. We first recall the RDF graphs over which all types of queries in this dissertation are to be evaluated, presents its semantics which will be used for defining the semantics of our extensions, and provide an inference mechanism based on graph homomorphism that can be used for checking the RDF consequences and RDF querying answering. The second chapter of the background, **Chapter 3**, discusses the current query languages for the semantic web in general and for RDF in particular, and highlights the main differences between them and our proposal.

In the research part, we provide our contribution which is presented in several chapters. **Chapter 4** presents a general graph framework that supports path expressions in RDF knowledge bases. Its syntax is a natural extension of RDF syntax, and its semantics is defined based on RDF semantics. A path-based graph homomorphism is provided to be used for querying RDF graphs. We instantiate this model to regular expressions in **Chapter 5** providing an extension to SPARQL, called PSPARQL, that covers the limitation of SPARQL in expressing paths. PSPARQL also serves as the basis for defining in **Chapter 6** a new generation, called CPSPARQL, that further extends (P)SPARQL by allowing, for example, complex constraints on nodes and edges of traversed paths. **Chapter 7** presents possible extensions of CPSPARQL such as using path variables, express-

ing constraints on path variables, expressing similarity path-based matching, allowing nested construct queries and extending constraints used in CPSPARQL. In **Chapter 8**, we give an overview to several methods for querying RDFS graphs with SPARQL and provide a method for querying RDFS based upon rewriting SPARQL queries into PSPARQL queries.

Chapter 9 presents a concrete implementation of our extensions based on the ideas presented in Chapters 4–6, as well as several exprimental tests of the prototype.

A summary of the results of the thesis is presented in **Chapter 10**, in which we conclude with several directions for the future work.

Part I

Background

The RDF Language

2

Contents

2.1 RDF Syntax 12
2.1.1 RDF terminology 13
2.1.2 RDF graphs as triples 13
2.1.3 Graph representation of RDF triples 14
2.2 Simple RDF Semantics 15
2.2.1 Interpretations 15
2.2.2 Models 16
2.2.3 Satisfiability, validity, and consequence 17
2.3 Inference Mechanism 18
2.4 RDF Entailment: Definition and Complexity 20
2.5 RDF vs. Graph Database Models 20
2.5.1 Relational data models 20
2.5.2 Object data models 21
2.5.3 Semi-structured data models 21
2.5.4 Other graph data models 22
2.6 Conclusion 22

Introduction

The Resource description Framework (RDF) is a W3C standard language dedicated to the annotation of resources within the Semantic Web [Manola and Miller, 2004]. The atomic constructs of RDF are *statements*, which are triples (subject, predicate, object) consisting of the resource (the subject) being described, a property (the predicate), and a property value (the object).

For example, the assertion of the following RDF triples {⟨`book1 rdf:type publication`⟩, ⟨`book1 title "Ontology Matching"`⟩, ⟨`book1 author "Jérôme Euzenat"`⟩, ⟨`book1 publisher "Springer"`⟩} means that `"Jérôme Euzenat"` is an author of a book titled `"Ontology Matching"` whose publisher is `"Springer"`.

A collection of RDF statements (RDF triples) can be intuitively understood as a directed labeled graph: resources are nodes and statements are arcs (from the subject node to the object node) connecting the nodes. The language is provided with a model-theoretic semantics [Hayes, 2004], that defines the notion of consequence (or entailment) between two RDF graphs, *i.e.,* when an RDF graph is entailed by another one. Answers to an RDF query (the knowledge base and the query are RDF graphs) are determined by the consequence, and can be computed using a particular *map* (a mapping from terms of the query to terms of the knowledge base preserving constants), a *graph homomorphism* [Gutierrez *et al.*, 2004; Baget, 2005].

RDFS (RDF Schema) [Brickley and Guha, 2004] is an extension of RDF designed to describe relationships between resources and/or resources using a set of reserves words called the RDFS vocabulary. In the above example, the reserved word `rdf:type` can be used to relate instances to classes, *e.g.*, `book1` is of type `publication`.

This chapter is devoted to the presentation of *Simple RDF* without RDF/RDFS vocabulary [Brickley and Guha, 2004]. We first recall (Section 2.1) its abstract syntax [Carroll and Klyne, 2004], its semantics (Section 2.2, using the notions of simple interpretations, models, simple entailment of [Hayes, 2004]), then Section 2.3 uses homomorphisms to characterize simple RDF entailment (as done in [Baget, 2005] for a graph-theoretic encoding of RDF, and in [Gutierrez *et al.*, 2004] for a database encoding), instead of the equivalent interpolation lemma of [Hayes, 2004]. Section 2.4 introduces the RDF entailment problem and its complexity. In Section 2.5, we compare RDF data model with database models, and concentrate in those that are based upon the graph structure.

2.1 RDF Syntax

RDF can be expressed in a variety of formats including RDF/XML [Beckett, 2004], Turtle [Beckett, 2006], etc. We use here its abstract syntax (triple format), which is sufficient for illustrating our proposal. To define the syntax of RDF, we need to

introduce the *terminology* over which RDF graphs are constructed.

2.1.1 RDF terminology

The RDF *terminology* $\mathcal{T}$ is the union of three pairwise disjoint infinite sets of *terms* [Hayes, 2004]: the set $\mathcal{U}$ of *urirefs*[1], the set $\mathcal{L}$ of *literals* (itself partitioned into two sets, the set $\mathcal{L}_p$ of *plain literals* and the set $\mathcal{L}_t$ of *typed literals*), and the set $\mathcal{B}$ of *variables*. The set $\mathcal{V} = \mathcal{U} \cup \mathcal{L}$ of *names* is called the *vocabulary*. From now on, we use different notations for the elements of these sets: a variable will be prefixed by `?` (like `?b1`), a literal will be between quotation marks (like `"27"`), and the rest will be urirefs (like `foaf:Person` — `foaf:`[2] is a name space prefix used for representing personal information — `ex:friend` or simply `friend`).

2.1.2 RDF graphs as triples

RDF graphs are usually constructed over the set of urirefs, blanks, and literals [Carroll and Klyne, 2004]. "Blanks" is a vocabulary specific to RDF. Because we want to stress the compatibility of the RDF structure with classical logic, we will use the term *variable* instead. The specificity of a blank with regard to variables is their quantification. Indeed, a blank in RDF is an existentially quantified variable. We prefer to retain this classical interpretation which is useful when an RDF graph is put in a different context. In the SPARQL query language, variables and blanks have different behaviors in complex cases. For example, a blank shared in different simple patterns of a group query pattern has a local scope which is easier to describe as changing the quantification scope of a variable than changing a blank into a variable. So, for the purpose of this thesis and without loss of generality, we have chosen to follow [Perez *et al.*, 2006] to not distinguish between variables and blanks, and speak of variables instead.

Definition 2.1.1 (RDF graph) *An* RDF triple *is an element of* $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times \mathcal{T}$. *An* RDF graph *is a finite set of RDF triples.*

Excluding variables as predicates and literals as subject was an unnecessary restriction in the RDF design, that has been relaxed in many RDF extensions. These constraints simplifies the syntax specification, and relaxing them neither changes

[1] An uri (uniform resource identifier) generalizes url (uniform resource locater) for identifying not only web pages but any resource (human, book, an author property). An uriref is a uri with a fragment (*e.g.* `http://www.example.org/homepage.html#section1`).

[2] `http://xmlns.com/foaf/spec/`

RDF semantics nor the computational properties of reasoning. In consequence, we adopt such an extension introduced in [Horst, 2005] and called *generalized RDF graphs*, or simply GRDF graphs.

Definition 2.1.2 (GRDF graph) *A* GRDF triple *is an element of* $\mathcal{T} \times (\mathcal{U} \cup \mathcal{B}) \times \mathcal{T}$. *A GRDF graph is a finite set of GRDF triples.*

Example 2.1.3 *The following set of triples represents a GRDF graph:*

```
{
   ⟨ ?b1    foaf:name    "Faisal" ⟩,
   ⟨ ?b1   ex:daughter   ?b2 ⟩,
   ⟨ ?b2       ?b4       ?b3 ⟩,
   ⟨ ?b3   foaf:knows    ?b1 ⟩,
   ⟨ ?b3    foaf:name    ?name ⟩
}
```

Intuitively, this graph means that there exists an entity named (`foaf:name`) `"Faisal"` that has a daughter (`ex:daughter`) that has some relation with another entity whose name is non determined, and that knows (`foaf:knows`) the entity named `"Faisal"`.

Notations If $\langle s, p, o\rangle$ is a GRDF triple, s is called its *subject*, p its *predicate*, and o its *object*. We denote by $subj(G)$ the set $\{s \mid \langle s, p, o\rangle \in G\}$ the set of elements appearing as a subject in a triple of a GRDF graph G. $pred(G)$ and $obj(G)$ are defined in the same way for predicates and objects. We call $nodes(G)$ the *nodes* of G, the set of elements appearing either as subject or object in a triple of G, i.e., $subj(G) \cup obj(G)$. A *term* of G is an element of $term(G) = subj(G) \cup pred(G) \cup obj(G)$. If $\mathcal{Y} \subseteq \mathcal{T}$ is a set of terms, we denote $\mathcal{Y} \cap term(G)$ by $\mathcal{Y}(G)$. For instance, $\mathcal{V}(G)$ is the set of names appearing in G.

A *ground* GRDF graph G is a GRDF graph with no variables, *i.e.,* $term(G) \subseteq \mathcal{V}$.

2.1.3 Graph representation of RDF triples

A *simple GRDF graph* can be represented graphically as a directed labeled graph[3] (N, E, γ, λ) where the set of nodes N is the set of terms appearing as a subject

[3]In fact as a directed labeled multigraph since multiple arcs with different labels may exists between two given nodes.

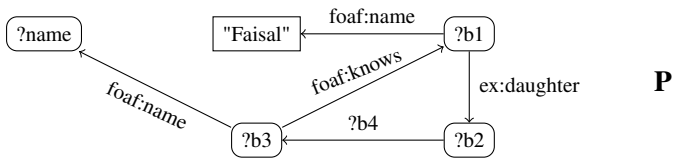

Figure 2.1: A GRDF graph.

or object in at least one triple of G, the set of arcs E is the set of triples of G, γ associates to each arc a pair of nodes (its extremities) $\gamma(e) = \langle\gamma_1(e), \gamma_2(e)\rangle$ where $\gamma_1(e)$ is the source of the arc e and $\gamma_2(e)$ its target; finally, λ labels the nodes and the arcs of the graph: if s is a node of N, *i.e.,* a term, then $\lambda(s) = s$, and if e is an arc of E, *i.e.,* a triple (s, p, o), then $\lambda(e) = p$. When drawing such graphs, the nodes resulting from literals are represented by rectangles while the others are represented by rectangles with rounded corners. In what follows, we do not distinguish between the two views of the RDF syntax (as sets of triples or directed labeled graphs). We will then speak interchangeably about their nodes, their arcs, or the triples which make them up.

For example, the GRDF triples given in Example 2.1.3 can be represented graphically as shown in Figure 2.1.

2.2 Simple RDF Semantics

[Hayes, 2004] introduces several semantics for RDF graphs. In this section, we present only the *simple semantics* without RDF/RDFS vocabulary [Brickley and Guha, 2004]. The definitions of interpretations, models, satisfiability, and entailment correspond to the *simple interpretations*, *simple models*, *simple satisfiability*, and *simple entailments* of [Hayes, 2004]. It should be noted that RDF and RDFS consequences (or entailments) can be polynomially reduced to simple entailment via RDF or RDFS rules [Baget, 2003; Horst, 2005] (see Section 8.2).

2.2.1 Interpretations

An interpretation describes possible way(s) the world might be in order to determine the truth-value of any ground RDF graph. It does this by specifying for each uriref, what is its denotation? In addition, if it is used to indicate a property, what values that property has for each thing in the universe?

Interpretations that assign particular meanings to some names in a given vocabulary will be named from that vocabulary, *e.g.* RDFS interpretations (see Section 8.1). An interpretation with no particular extra conditions on a vocabulary (including the RDF vocabulary itself) will be simply called an interpretation.

Definition 2.2.1 (Interpretation of a vocabulary) *Let $V \subseteq \mathcal{V} = \mathcal{U} \cup \mathcal{L}$ be a vocabulary. An* interpretation *of V is a tuple $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ where:*

- *I_R is a set of* resources *that contains $V \cap \mathcal{L}$;*
- *$I_P \subseteq I_R$ is a set of properties;*
- *$I_{EXT} : I_P \rightarrow 2^{I_R \times I_R}$ associates to each property a set of pairs of resources called the* extension *of the property;*
- *the interpretation function $\iota : V \rightarrow I_R$ associates to each name in V a resource of I_R, if $v \in \mathcal{L}$, then $\iota(v) = v$.*

2.2.2 Models

By providing RDF with formal semantics, [Hayes, 2004] expresses the conditions under which an RDF graph truly describes a particular world (*i.e.,* an interpretation is a model for the graph). The usual notions of validity, satisfiability and consequence are entirely determined by these conditions.

Intuitively, a ground triple $\langle s, p, o \rangle$ in a GRDF graph will be true under the interpretation I if p is interpreted as a property (for example, r_p), s and o are interpreted as resources (for example, r_s and r_o, respectively), and the pair of resources $\langle r_s, r_o \rangle$ belongs to the extension of the property r_p. A triple $\langle s, p, ?b \rangle$ with the variable $?b \in \mathcal{B}$ would be true under I if there exists a resource r_b such that the pair $\langle r_s, r_b \rangle$ belongs to the extension r_p. When interpreting a variable node, an arbitrary resource can be chosen. To ensure that a variable always is interpreted by the same resource, extensions of the interpretation function is defined as follow.

Definition 2.2.2 (Extension to variables) *Let $I = (I_R, I_P, I_{EXT}, \iota)$ be an interpretation of a vocabulary $V \subseteq \mathcal{V}$, and $B \subseteq \mathcal{B}$ a set of variables. An extension of ι to B is a mapping $\iota' : \mathcal{V} \cup B \rightarrow I_R$ such that $\forall x \in V$, $\iota'(x) = \iota(x)$.*

An interpretation I is a model of GRDF graph G if all triples are true under I.

Definition 2.2.3 (Model of a GRDF graph) *Let $V \subseteq \mathcal{V}$ be a vocabulary, and G be a GRDF graph such that every name appearing in G is also in V ($\mathcal{V}(G) \subseteq V$).*

An interpretation $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ *of* V *is a* model *of* G *iff there exists an extension* ι' *that extends* ι *to* $\mathcal{B}(G)$ *such that for each triple* $\langle s, p, o \rangle$ *of* G, $\iota'(p) \in I_P$ *and* $\langle \iota'(s), \iota'(o) \rangle \in I_{EXT}(\iota'(p))$. *The mapping* ι' *is called a* proof *of* G *in* I.

2.2.3 Satisfiability, validity, and consequence

The following definition is the standard model-theoretic definition of satisfiability validity and consequence.

Definition 2.2.4 (Satisfiability, validity, consequence) *A graph* G *is* satisfiable *iff it admits a model.* G *is* valid *iff for every interpretation* I *of a vocabulary* $V \supseteq \mathcal{V}(G)$, I *is a model of* G. *A graph* G' *is a* consequence *of a graph* G, *denoted* $G \models_{GRDF} G'$, *iff every model of* G *is also a model of* G'.

Proposition 2.2.5 (Satisfiability, validity) *Every GRDF graph is satisfiable. The only valid GRDF graph is the empty graph.*

Proof. (Satisfiability) [Baget, 2003; Horst, 2004; Baget, 2005] builds the isomorphic model of a GRDF graph G, denoted by $I_{iso}(G)$. The construction of $I_{iso}(G) = (I_R, I_P, I_{EXT}, \iota)$ can be made as follows:

(i) the set of resources in $I_{iso}(G)$ is the set of terms of G, *i.e.,* $I_R = term(G)$;

(ii) the set of properties in $I_{iso}(G)$ is the set of predicates of G, *i.e.,* $I_P = pred(G)$;

(iii) the identity $\forall x \in \mathcal{V}(G)$, $\iota(x) = x$;

(iv) $\forall p \in I_P$, $I_{EXT}(p) = \{\langle s, o \rangle \in I_R \times I_R \mid \langle s, p, o \rangle \in G\}$.

Let us prove that $I_{iso}(G)$ is a model of G. Consider the extension ι' of ι to $\mathcal{B}(G)$ defined by $\forall x \in term(G)$, $\iota'(x) = \iota(x) = x$. The condition of Definition 2.2.3 immediately follows from the construction of $I_{iso}(G)$. Note that ι is a bijection between $term(G)$ and I_R.

(Validity) a non empty GRDF graph has no proof in an interpretation in which all properties are interpreted by I_{EXT} as an empty set.

2.3 Inference Mechanism

SIMPLE RDF ENTAILMENT [Hayes, 2004] can be characterized as a kind of graph homomorphism. A *graph homomorphism* from an RDF graph H into an RDF graph G, as defined in [Baget, 2005; Gutierrez *et al.*, 2004], is a mapping π from the nodes of H into the nodes of G preserving the arc structure, *i.e.,* for each node $x \in H$, if $\lambda(x) \in \mathcal{U} \cup \mathcal{L}$ then $\lambda(\pi(x)) = \lambda(x)$; and each arc $x \xrightarrow{p} y$ is mapped to $\pi(x) \xrightarrow{\pi(p)} \pi(y)$. This definition is similar to the projection used to characterize entailment of conceptual graphs (CGs) [Mugnier and Chein, 1992] (*cf.* [Corby *et al.*, 2000] for precise relationship between RDF and CGs). We modify this definition to the one that maps $term(H)$ into $term(G)$. Maps are used to ensure that a variable always mapped to the same term, as done for extensions to interpretations.

Definition 2.3.1 (Map) *Let $V_1 \subseteq \mathcal{T}$, and $V_2 \subseteq \mathcal{T}$ be two sets of terms. A* map *from V_1 to V_2 is a mapping $\mu : V_1 \rightarrow V_2$ such that $\forall x \in (V_1 \cap \mathcal{V})$, $\mu(x) = x$.*

The *map* defined in [Gutierrez *et al.*, 2004; Perez *et al.*, 2006] is a particular case of Definition 2.3.1. An RDF homomorphism is a map preserving the arc structure.

Definition 2.3.2 (GRDF homomorphism) *Let G and H be two GRDF graphs. A GRDF homomorphism from H into G is a map π from $term(H)$ to $term(G)$ such that $\forall \langle s, p, o\rangle \in H$, $\langle \pi(s), \pi(p), \pi(o)\rangle \in G$.*

The definition of GRDF homomorphisms (Definition 2.3.2) is similar to the *map* defined in [Gutierrez *et al.*, 2004] for RDF graphs. [Gutierrez *et al.*, 2004] provides without proof an equivalence theorem (Theorem 3) between RDF entailment and maps. A proof is provided in [Baget, 2005] also for RDF graphs, but the homomorphism involved is a mapping from nodes to nodes, and not from terms to terms. In RDF, the two definitions are equivalent. However, the terms-to-terms version is necessary to extend the theorem of RDF (Theorem 2.3.4) to the PRDF graphs studied in Chapter 4. The proof of Theorem 2.3.4 will be a particular case of the proof of Theorem 4.3.5 for PRDF graphs.

Example 2.3.3 (GRDF homomorphism) *Figure 2.2 shows two GRDF graphs Q and G (note that the graph Q is the graph P of Figure 2.1, to which the following triple is added $\langle$`?b3`, `foaf:mbox`, `?mbox`$\rangle$. The map π_1 defined by $\{$(`"Faisal"`, `"Faisal"`), (`?b1`, `?c1`), (`?name`, `"Natasha"`), (`?b2`, `?c2`), (`?b4`, `ex:friend`),*

(?mbox, "natasha@yahoo.com"), (?b3, ex:Person1)} is a GRDF homomorphism from Q into G. And the map π_2 defined by {("Faisal", "Faisal"), (?b1, ?c1), (?name, "Deema"), (?b3, ex:Person2), (?b4, ex:friend), (?b2, ?c2)} is a GRDF homomorphism from P into G. Note that π_2 cannot be extended to a GRDF homomorphism from Q into G since there is no mailbox for "Deema" in G.

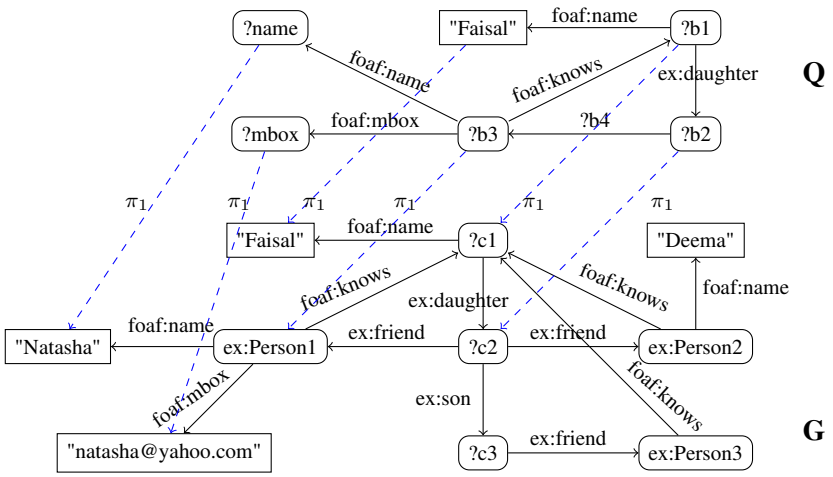

Figure 2.2: A GRDF homomorphism from Q into G.

Theorem 2.3.4 *Let G and H be two GRDF graphs, then $G \models_{GRDF} H$ if and only if there is a GRDF homomorphism from H into G.*

The proof of this theorem is an immediate consequence of the proof of Theorem 4.3.5, since each GRDF graph is a PRDF graph. Moreover, any PRDF homomorphism between GRDF graphs is a GRDF homomorphism and, by Proposition 4.2.3, PRDF entailment applied to GRDF graphs is equivalent to GRDF entailment.

This equivalence between the semantic notion of entailment and the syntactic notion of homomorphism is the ground by which a correct and complete query answering procedure can be designed. More precisely, the set of answers to a GRDF graph query Q over an RDF knowledge base G are the set of RDF homomorphisms from Q into G which, by Theorem 2.3.4, correspond to RDF consequence. For a more complex query, which is basically built on top of GRDF graphs, then the answers are constructed from the set of RDF homomorphisms from its GRDF graphs into the RDF knowledge base(s) (see Section 3.3.2).

2.4 RDF Entailment: Definition and Complexity

The decision problem associated to simple RDF semantics is called SIMPLE RDF ENTAILMENT, and is defined as follows:

SIMPLE (G)RDF ENTAILMENT
Instance: two GRDF graphs G and H.
Question: Does $G \models_{\text{GRDF}} H$?

SIMPLE (G)RDF ENTAILMENT is an NP-complete problem for RDF graphs [Gutierrez *et al.*, 2004]. For GRDF graphs, its complexity remains unchanged [Perez *et al.*, 2006]. Polynomial subclasses of the problem have been exhibited based upon the structure or labeling of the query:

- when the query is ground [Horst, 2004], or more generally if it has a bounded number of variables,
- when the query is a tree or admits a bounded decompositions into a tree, according to the methods in [Gottlob *et al.*, 1999] as shown in [Baget, 2005].

2.5 RDF vs. Graph Database Models

In order to compare RDF query languages with those of database, we first need to identify the differences in the underlying data models.

In this section, we provide a brief presentation of the RDF data model with some of the database models, and stress on those that are based on a graph model. See [Angles and Gutierrez, 2008] for a survey of database models and [Kerschberg *et al.*, 1976] for a taxonomy of data models.

2.5.1 Relational data models

The relational data model is introduced in [Codd, 1983] to highlight the concept level of abstraction by separating the physical and logical levels. It is a simple model based on the notions of sets and relations with a defined algebra and logic. SQL is its standard query and manipulation language.

The main differences with RDF are: the relational model is that the data have a predefined structure with simple record-type, and the schema is fixed and difficult to be extended. The same differences between RDF and the object data models are also applied to the relational data model (see the following subsection).

2.5.2 Object data models

These models are based on the object-oriented programing paradigms [Kim, 1990], representing data as a collection of objects interacting among them by methods.

The main differences between object oriented data models and RDF are: RDF resources can occur as edge labels or node labels; no strong typing in RDF (*i.e.,* classes do not define object types); properties may be refined respecting only the domain and range constraints; RDF resources can be typed of different classes, which are not necessarily pairwise related by specialization, *i.e.,* the instances of a class may have associated quite different properties such that there is no other class on which the union of these properties is defined.

Among object oriented data models are: O2 [Lécluse *et al.*, 1988] based on a graph structure; and Good [Gyssens *et al.*, 1990] that has a transparent graph-based manipulation and representation of data.

2.5.3 Semi-structured data models

These models are oriented to model semi-structured data [Buneman, 1997; Abiteboul, 1997]. They deal with data whose structure is irregular, implicit, and partial, and with schema contained in the data.

One of these models is OEM (Object Exchange Model) [Papakonstantinou *et al.*, 1995]. It aims to express data in a standard way to solve the information exchange problem. This model is based on objects that have unique identifiers, and property value that can be simple types or references to objects. However, labels in the OEM model can not occur in both nodes (objects) and edges (properties), and OEM is schemaless while RDF may be coupled with RDFS. Moreover, nodes in RDF can be also blanks.

Another data model is XML data model [Bray *et al.*, 2006]. However, RDF has substantial differences from the XML data model [Bray *et al.*, 2006]. XML has an ordered-tree like structure against the graph structure of RDF. Also, information about data in XML is part of the data while RDF expresses explicitly information about data using relation betweens entities. In addition, we can not distinguish in RDF between entity (or node) labels and relation labels, and RDF resources may have irregular structures due to multiple classification.

2.5.4 Other graph data models

The Functional Data Model [Shipman, 1981] is one of models that considers an implicit structure of graphs for the data, aiming to provide a "conceptually natural" database interface. A different approach is the Logical Data Model proposed in [Kuper and Vardi, 1993], where an explicit graph model is considered for representing data. In this model, there are three types of nodes (namely basic, composition and collection nodes), all of which can be modeled in RDF. Among the models that have explicit graph data model are: G-Base [Kunii, 1987] representing complex structures of knowledge; Gram [Amann and Scholl, 1992] representing hypertext data; GraphDB [Gting, 1994] modeling graphs in object oriented databases; and Gras [Kiesel *et al.*, 1996]. They have no direct applicability of a graph model to RDF since RDF resources can occur as edge or node labels. Solving this problem requires an intermediate model to be defined, *e.g.* bipartite graphs [Hayes and Gutierrez, 1996].

2.6 Conclusion

Nowadays, more resources are annotated via RDF due to its simple data model, formal semantics, and a sound and complete inference mechanism. RDF itself can be used as a query language for an RDF knowledge base using RDF consequence. Nonetheless, the use of consequence is still limited for answering queries. In particular, answering those that contain complex relations requires complex constructs. It is impossible, for example, to answer the query "find the names and addresses, if they exist, of persons who either work on query languages or ontology matching" using a simple consequence test.

Therefore the need for added expressivity in queries has led to define several query languages on top of graph patterns that are basically RDF and more precisely GRDF graphs. The focus of the next chapter is then to give an overview of some languages that have been designed or can be used for querying RDF graphs, and discusses the main differences between them in terms of expressiveness and limitations.

Querying RDF Graphs 3

Contents

3.1 (Semi)-Structured Query Languages 24
3.2 RDF Query Languages 26
3.3 The SPARQL Query Language 27
3.3.1 SPARQL syntax 27
3.3.2 Formal semantics: answers to SPARQL queries 31
3.4 Extensions to SPARQL 33
3.5 Work on SPARQL 35
3.6 Comparison with other Query Languages 36
3.7 Conclusion 37

Introduction

A query language can be understood as an inference mechanism for manipulating and inferencing data from valid instances of the data model.

This chapter surveys the languages that can be used for querying RDF graphs. In particular, Section 3.1 reviews some of the well-know graph query languages used for querying structured or semi-structured data bases. Section 3.2 presents some of RDF query languages. Section 3.3 details the SPARQL query languages as well as a semantic query framework, which will be the basis for our proposal. In Section 3.4, we present some extensions of SPARQL and stress in the strongly related ones. In Section 3.6, we compare (C)PSPARQL to some of the existing query languages. Finally, we discuss some work on SPARQL in Section 3.5.

3.1 (Semi)-Structured Query Languages

Query languages for structured graph data base models can be used for querying RDF viewing RDF data as a graph that may contain transitive or repetitive patterns of relations. Among them, G [Cruz *et al.*, 1987] and its extension G+ [Cruz *et al.*, 1988] are two languages for querying structured databases. A simple G+ query has two elements, a query graph that specifies the pattern to be matched and a summary graph that defines graphically how the answers are to be structured and then presented to the user

Example 3.1.1 *Given a graph that represents relations between people, the G+ query of Figure 3.1 finds pairs of people who share a common ancestor.*

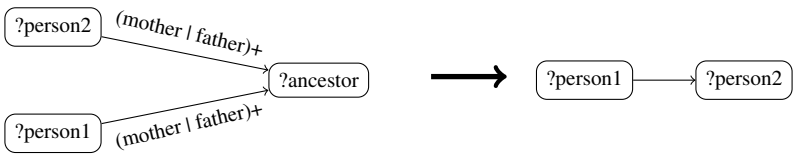

Figure 3.1: A G+ query to find common ancestor.

The left hand side of the bold arrow is the pattern to be matched in the knowledge base while the right hand side is the summary graph.

Graphlog — a visual query language which has been proven equivalent to linear Datalog [Consens and Mendelzon, 1990] — extends G+ by combining it with the Datalog notation. It has been designed for querying hypertext. A Graphlog query is only a graph pattern containing a distinguished edge or arc (*i.e.,* it is a restructuring edge, which corresponds to the summary graph in G+).

Example 3.1.2 *Figure 3.2 shows a Graphlog query: dashed lines represent edge labels with the positive closure, a crossed dashed line represents a negated label (e.g.* `!descendant+` *between* `?person2` *and* `?person3`*),* `person` *is a unary predicate, and finally a bold line represents a distinguished edge that must be labeled with a positive label. The effects of this query is to find all instances of the pattern that occur in the database, i.e., finding descendant of* `?person1` *which are not descendant of* `?person2`*. Then, for each one of them, define a virtual link represented by the distinguished edge.*

These query languages (namely G, G+ and Graphlog) support only graphical queries similar to PRDF queries. In contrast to PRDF, they are limited to finding

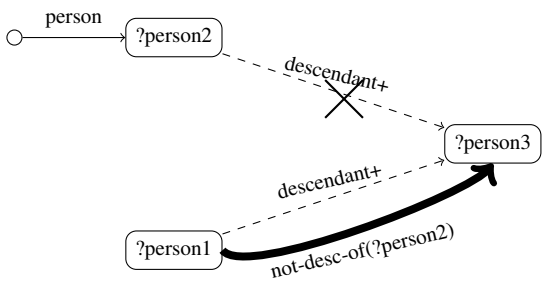

Figure 3.2: A Graphlog query.

simple paths (cycle-free paths). The main problem with finding only simple paths, is that there are situations in which answers to such queries are all non simple, *e.g.* if the only paths matching a regular expression pattern have cycles (see the example of non-simple paths in [Anyanwu *et al.*, 2007]). In addition, the complexity of finding simple paths problem is NP-complete even without variables in regular expressions [Wood, 1988]. Moreover, they do not provide complex functionalities, for example, for filtering, ordering, projection, union of graph patterns, optional graph patterns and other useful features (see SPARQL features and examples below).

Lorel [Abiteboul *et al.*, 1997] is an OEM-based language for querying semi-structured documents. It uses regular expression patterns for traversing object hierarchy paths, restricted to simple path semantics (or acyclic paths). UnQL [Buneman *et al.*, 1996] is a language closely related to Lorel for querying semi-structured data. It is based on a data model similar to OEM [Buneman *et al.*, 1995]. A particular aspect of the language is that it allows some form of restructuring even for cyclic structures. A traverse construct allows one to transform a database graph while traversing it, *e.g.* by replacing all labels A by the label A_0. This powerful operation combines tree rewriting techniques with some control obtained by a guided traversal of the graph. For instance, one could specify that the replacement occurs only if a particular edge, say B, is encountered on the way from the root. STRUQL [Fernandez *et al.*, 1997], a query language for a web-site management system, incorporates regular expressions and has precisely the same expressive power as stratified linear Datalog. It became clear that query languages for semi-structured data or that are based on object oriented models are not well suited for RDF as discussed in Section 2.5, and also as stated in [Karvounarakis *et al.*, 2002]. WebSQL [Mendelzon *et al.*, 1997], incorporates regular expressions for

querying distributed collection of documents connected by hypertext links. It has a cost based query evaluation mechanism, *i.e.,* it evaluates how much of the network must be visited to answer a particular query. To our knowledge, none of the above query languages allow expressing constraints on internal nodes, which is allowed by CPRDF.

A Logic that incorporates a kind of constrained regular expressions has been proposed for XPath [Genevès *et al.*, 2007]. However, XPath operates on trees (not on graphs), and only defines monadic queries [Clark and DeRose, 1999]. Several works attempt to adapt PDL-like or μ-calculus based on monadic queries for querying graphs, for example [Alechina *et al.*, 2003].

3.2 RDF Query Languages

Several query languages have been proposed for RDF [Haase *et al.*, 2004]. Most of them use a query model based on *relational algebra* [Codd, 1970], where RDF graphs are viewed as a collection of triples and the queries are triple-based formulas expressed over a single relation. In spite of the benefits gained from the existing relational database systems such as indexing mechanisms, underlying storage of triples as relations [Harris and Shadbolt, 2005], query optimization techniques, and others; relational queries cannot express recursive relations and even the most simple form, the transitive closure of a relation [Aho and Ullman, 1979], directly inherited from the graph nature of RDF triples.

There are many real-world applications, inside and outside the domain of the semantic web, requiring data representation that are inherently recursive. For that reason, there are several attempts to extend relational algebra to express complex query modeling. Outside the domain of the semantic web, we mention [Agrawal, 1988] that extends the relational algebra to represent transitive closure and [Jagadish, 1989] to represent query hierarchies. In the domain of RDF, some query languages such as RQL [Karvounarakis *et al.*, 2002] attempts to combine the relational algebra with some special class hierarchies. It supports a form of transitive expressions over RDFS transitive properties (*i.e.,* subPropertyOf and subClassOf) for navigating through class and property hierarchies. Versa [Olson and Ogbuji, 2002], RxPath [Souzis, 2004], PRDF [Alkhateeb *et al.*, 2005; Alkhateeb, 2007] and [Matono *et al.*, 2005] are all path-based query languages for RDF that are well suited for graph traversal but do not support SQL-like functionalities. WILBUR [Lassila, 2002] is a toolkit that incorporates path expressions for navigation in RDF

graphs. [Zhang and Yoshikawa, 2008] discusses the usage of a Concise Bounded Description (CBD) of an RDF graph, which is defined as a subgraph consisting of those statements which together constitute a focused body of knowledge about a given resource (or node) in a given RDF graph. It also defines a Dynamic version (DCBD) of CBD as well as proposes a query language for RDF called DCBD-Query, which mainly addresses the problem of finding meaningful (shortest) paths with respect to DCBD.

SQL-like query languages for RDF include SeRQL [Broekstra, 2003], RDQL [Seaborne, 2004] and its current successor – a W3C recommendation – SPARQL [Prud'hommeaux and Seaborne, 2008]. Since it is defined by the W3C's Data Access Working Group (DAWG) and becomes the most popular query language for RDF, we chose to build our work on SPARQL and avoid reinventing another query language for RDF. So, SPARQL will be presented below in more details than the other languages.

3.3 The SPARQL Query Language

There has been early proposals for specific RDF query languages, such as RDQL [Seaborne, 2004], RQL [Karvounarakis *et al.*, 2002] or SeRQL [Broekstra, 2003]. In 2004, the W3C launched the Data Access Working Group for designing an RDF query language, called SPARQL, from these early attempts [Prud'hommeaux and Seaborne, 2008]. SPARQL query answering is characterized by defining maps from GRDF graphs used as query patterns of the query to the RDF knowledge base [Perez *et al.*, 2006].

3.3.1 SPARQL syntax

SPARQL graph patterns

The heart of SPARQL queries is graph patterns. Informally, a *graph pattern* can be one of the following (*cf.* [Prud'hommeaux and Seaborne, 2008] for more details):

- a **triple pattern**: a triple pattern corresponds in RDF to a GRDF triple;
- a **basic graph pattern**: a set of triple patterns (or a GRDF graph) is called a basic graph patterns;
- a **union of graph patterns**: we use the keyword `UNION` in SPARQL to represent alternatives;

- an **optional graph pattern**: SPARQL allows optional results to be returned determined by the keyword `OPT`;
- a **constraint**: constraints in SPARQL are boolean-valued expressions that limit the number of answers to be returned. They can be defined using the keyword `FILTER`. As atomic `FILTER` expressions, SPARQL allows unary predicates like `BOUND`; binary (in)equality predicates ($=$ and $!=$); comparison operators like $<$; data type conversion and string functions which will be omitted here. Complex `FILTER` expressions can be built using !, || and &&;
- a **group graph pattern**: is a graph pattern grouped inside { and }, and determines the scope of SPARQL constructs like `FILTER` and variable nodes;

Definition 3.3.1 (SPARQL graph pattern) *A* SPARQL graph pattern *is defined inductively in the following way:*

- *every GRDF graph is a basic SPARQL graph pattern;*
- *if P_1, P_2 are SPARQL graph patterns and C is a SPARQL constraint, then $\{P_1\}$, (P_1 `AND` P_2), (P_1 `UNION` P_2), (P_1 `OPT` P_2), and (P_1 `FILTER` C) are SPARQL graph patterns.*

Example 3.3.2 *The following graph pattern:*

```
{ ?person foaf:knows "Faisal" . }
```

is a basic graph pattern that can be used in a query for finding persons who know Faisal.

```
{
   { ?person ex:liveIn ex:France . }
      UNION
   { ?person ex:hasNationality ex:French . }
}
```

is a union of two basic graph patterns that searches the persons who either live in France or have a French nationality.

The following graph pattern

```
{    ?person foaf:knows "Faisal" .
       OPT
   { ?person foaf:mbox ?mbox . }
}
```

contains an optional basic graph pattern searching the mail boxes, if they exist, of persons who know Faisal.

```
{
    ?person ex:liveIn ex:France .
    ?person ex:hasAge ?age .
    FILTER ( ?age < 40 ) .
}
```

the constraint in this graph pattern limits the answers to the persons who live in France whose ages are less than 40.

```
{ { ?person foaf:knows "Faisal" . }
  {
    ?person ex:liveIn ex:France .
    ?person ex:hasAge ?age .
    FILTER ( ?age < 40 ) .
  }
}
```

is a graph pattern of two group graph patterns. The scope of the constraint in this graph pattern is the second group graph pattern. So, it is applied only to the persons who live in France.

SPARQL query

A SELECT *SPARQL query* is expressed using a form resembling the SQL SELECT query:

$$\texttt{SELECT}\ \vec{B}\ \texttt{FROM}\ u\ \texttt{WHERE}\ P$$

where u is the URL of an RDF graph G, P is a SPARQL graph pattern and $\vec{B}$ is a tuple of variables appearing in P. Intuitively, an answer to a SPARQL query is an instantiation π of the variables of $\vec{B}$ by the terms of the RDF graph G such that π is a restriction of a proof that P is a consequence of G.

SPARQL provides several result forms other than SELECT that can be used for formating the query results. For example, CONSTRUCT that can be used for building an RDF graph from the set of answers, ASK that returns `TRUE` if there is a answer to a given query and `FALSE` otherwise, and DESCRIBE that can be used for describing a resource RDF graph. The following example queries give an insight of these query forms.

Example 3.3.3 *The following ASK query:*

```
ASK
WHERE { ?person foaf:names "Faisal" .
        ?person ex:hasChild ?child  .
      }
```

returns `TRUE` *if a person named Faisal has at least one child,* `FALSE` *otherwise.*

The following CONSTRUCT query:

```
CONSTRUCT { ?son1 ex:brother ?son2 .}
WHERE {     ?person foaf:names "Faisal" .
            ?son1 ex:sonOf ?person .
            ?son2 ex:sonOf ?person .
            FILTER ( ?son1 != ?son2 ) .
      }
```

constructs the RDF graph (containing the brotherhood relation) by substituting for each located answer the values of the variables $?son1$ *and* $?son2$*.*

The following query:

```
DESCRIBE <example.org/person1>
```

returns a description of the resource identified by the given uriref, i.e., returns the set of triples involving this uriref.

SPARQL uses post-filtering clauses which allow, for example, to order (ORDER BY clause), or to limit (LIMIT and/or OFFSET clauses) the answers of a query. The reader is referred to the SPARQL specification [Prud'hommeaux and Seaborne, 2008] for more details or to [Perez *et al.*, 2006] for formal semantics of SPARQL queries.

Example 3.3.4 *The following SPARQL query:*

```
SELECT  ?name
WHERE  {
        ?person ex:liveIn ex:France .
        ?person foaf:name ?name .
       }
ORDER BY ?name
LIMIT 10
OFFSET 5
```

returns the names of persons who live in France limited to maximum 10 *persons, ordered by their names, and starting from the* 5*th answer.*

Since the graph patterns in the SPARQL query language are shared by all SPARQL query forms and that our proposal is based upon extending these graph patterns, we illustrate our extension using the SELECT ... FROM ... WHERE ... queries. Our extension can then be applied to other query forms.

3.3.2 Formal semantics: answers to SPARQL queries

[Perez *et al.*, 2006] gives an alternate characterization of query answering, which relies upon the correspondence between maps from GRDF graph of the query graph patterns to the RDF knowledge base and GRDF entailment. Then, SPARQL query constructs are defined through algebraic operations on maps. In the following, we recall this characterization.

If μ is a map, then the domain of μ, denoted by $dom(\mu)$, is the subset of $\mathcal{T}$ where μ is defined. If P is a graph pattern, then $\mu(P)$ is the graph pattern obtained by the substitution of $\mu(b)$ to each variable $b \in \mathcal{B}(P)$. Two maps μ_1 and μ_2 are *compatible* when $\forall x \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(x) = \mu_2(x)$. If μ_1 and μ_2 are two compatible maps, then we denote by $\mu = \mu_1 \oplus \mu_2 : T_1 \cup T_2 \rightarrow \mathcal{T}$ the map defined by: $\forall x \in T_1, \mu(x) = \mu_1(x)$ and $\forall x \in T_2, \mu(x) = \mu_2(x)$. Analogously to [Perez *et al.*, 2006], we define the *join* of two sets of maps Ω_1 and Ω_2 as follows:

- *(join)*[1] $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \oplus \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible $\}$;
- *(difference)* $\Omega_1 \backslash \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1$ and μ_2 are not compatible$\}$.

Definition 3.3.5 (Answer to a SPARQL graph pattern) *Let G be an RDF graph and P be a SPARQL graph pattern. The set $\mathcal{S}(P, G)$ of answers of P in G is defined inductively in the following way:*

1. *if P is a GRDF graph, $\mathcal{S}(P, G) = \{\mu \mid \mu$ is an RDF homomorphism from P into $G\}$;*
2. *if $P = (P_1 \texttt{ AND } P_2)$, $\mathcal{S}(P, G) = \mathcal{S}(P_1, G) \bowtie \mathcal{S}(P_2, G)$;*
3. *if $P = (P_1 \texttt{ UNION } P_2)$, $\mathcal{S}(P, G) = \mathcal{S}(P_1, G) \cup \mathcal{S}(P_2, G)$;*
4. *if $P = (P_1 \texttt{ OPT } P_2)$, $\mathcal{S}(P, G) = (\mathcal{S}(P_1, G) \bowtie \mathcal{S}(P_2, G)) \cup (\mathcal{S}(P_1, G) \setminus \mathcal{S}(P_2, G))$;*
5. *if $P = (P_1 \texttt{ FILTER } C)$, $\mathcal{S}(P, G) = \{\mu \in \mathcal{S}(P_1, G) \mid \mu(C) = \top\}$.*

The semantics of SPARQL `FILTER` expressions is defined as follows: given a map μ and a SPARQL constraint C, we say that μ satisfies C (denoted by $\mu(C) = \top$), if:

- $C = \texttt{BOUND}(x)$ with $x \in dom(\mu)$;
- $C = (x = c)$ with $x \in dom(\mu)$ and $\mu(x) = c$;
- $C = (x = y)$ with $x, y \in dom(\mu)$ and $\mu(x) = \mu(y)$;

[1] [Polleres, 2007] defines join maps of unbound variables.

- $C = (x!= c)$ with $x \in dom(\mu)$ and $\mu(x)!= c$;
- $C = (x!= y)$ with $x, y \in dom(\mu)$ and $\mu(x)!= \mu(y)$;
- $C = (x < c)$ with $x \in dom(\mu)$ and $\mu(x) < c$;
- $C = (x < y)$ with $x, y \in dom(\mu)$ and $\mu(x) < \mu(y)$;
- $C =!C_1$ with $\mu(C_1) = \bot$ (μ does not satisfy C_1);
- $C = (C_1||C_2)$ with $\mu(C_1) = \top$ or $\mu(C_2) = \top$;
- $C = (C_1\&\&C_2)$ with $\mu(C_1) = \top$ and $\mu(C_2) = \top$;

Let $Q =$`SELECT` $\vec{B}$ `FROM` u `WHERE` P be a SPARQL query, G be the RDF graph identified by the URL u, and Ω is the set of maps of P in G. Then the answers of the query Q are the instantiation of elements of Ω to $\vec{B}$. That is, for each map π of Ω, the answer of Q associated to π is $\{(x, y) \mid x \in \vec{B}$ and $y = \pi(x)$ if $\pi(x)$ is defined, `null` otherwise$\}$.

Proposition 3.3.6 *Let $Q =$`SELECT` $\vec{B}$ `FROM` u `WHERE` P be a SPARQL query, P be a GRDF graph and G be the (G)RDF graph identified by the URI u, then the answers to Q are the images of variables in $\vec{B}$ by an RDF homomorphism π from P into G such that $G \models_{RDF} \pi(P)$.*

This property is a straightforward consequence of Definition 3.3.5. It is based on the fact that the answers to Q are the restrictions to $\vec{B}$ of the set of RDF homomorphisms from P into G which, by Theorem 2.3.4, corresponds to RDF-entailment.

Example 3.3.7 *Consider the following SPARQL query Q:*

```
SELECT ?name ?mbox
FROM   <http://example.org/index1.ttl>
WHERE  {
            ?b1 foaf:name "Faisal" .
            ?b1 ex:daughter ?b2 .
            ?b2 ?b4 ?b3 .
            ?b3 foaf:knows ?b1 .
            ?b3 foaf:name ?name .
      OPT {
            ?b2 foaf:mbox ?mbox .
          }
       }
```

such that the RDF graph identified by the uriref of the FROM clause is the graph G of Figure 2.2. This query contains two basic graph patterns: the optional GRDF

pattern with only one triple represented by the optional clause and the other part is the GRDF graph P of Figure 2.1. We construct the answer to the query by taking the join of homomorphisms from P into G and the homomorphisms from the optional triple into G; i.e., the homomorphisms from Q (see Figure 2.2) into G (e.g. the homomorphism π_1 of Example 2.3.3), and the homomorphisms from P into G that cannot be extended to include the optional triple, e.g. the homomorphism π_2 of Example 2.3.3. There are therefore two answers to the query:

?name	?mbox
"Deema"	null
"Natasha"	"natasha@yahoo.com"

To end this section, we note that simple (C)PSPARQL queries (*i.e.,* without recursive operators and path variables) can be expressed into SPARQL (see examples in Section 5.2.2).

3.4 Extensions to SPARQL

Corese [Corby *et al.*, 2004] is a semantic web search engine based on conceptual graphs that offers functionalities for querying RDF graphs. At the time of writing, it supports only fixed path length queries and no other path expressions such as variable-length paths or constraints on internal nodes, though this seems to be planned.

Two extensions of SPARQL, which are closely similar to PSPARQL [Alkhateeb, 2007], have been recently defined based on our initial proposal [Alkhateeb *et al.*, 2005]: SPARQLeR and SPARQ2L.

SPARQLeR [Kochut and Janik, 2007] extends SPARQL by allowing query graph patterns involving path variables. Each path variable is used to capture simple (*i.e.,* acyclic) paths in RDF graphs, and is matched against any arbitrary composition of RDF triples between given two nodes. This extension offers good functionalities like testing the length of paths and testing if a given node is in the found paths. Since SPARQLeR is not defined with a formal semantics, its use of path variables in the subject position is unclear, in particular, when they are not bound. Even when this is the case, multiple uses of same path variable several times is not fully defined: it is not specified which path is to be returned or if is it enforced to be the same. The effects of paths variables in the DISTINCT clause are not treated either. Finally, several problems are raised in the evaluation of graph

patterns of such extension. In particular, the strategy of obtaining paths and then filtering them is inefficient since it can generate a large number of paths.

Example 3.4.1 *The following SPARQLeR query:*

```
SELECT %path
WHERE  {
            <r> %path <s> .
            FILTER ( length(%path) < 10 ).
       }
```

matches any path of length less than 10 *between the resources* `<r>` *and* `<s>`*. The path variable* `%path` *is bound to the matched path.*

SPARQ2L [Anyanwu *et al.*, 2007] also allows using path variables in graph patterns and offers good features like constraints in nodes and edges, *i.e.,* testing the presence or absence of nodes and/or edges; constraints in paths, *e.g.* simple or non-simple paths, presence of a pattern in a path. This extension is also not described semantically. One can only try to guess what is the intuitive semantics of the constructs. It seems that the algorithms are not complete with regard to their intuitive semantics, since the set of answers can be infinite in absence of constraints for using shortest or acyclic paths. Moreover, this extension suffers from generality, *i.e.,* it does not allow using more than one triple pattern having a path variable. Relaxing this restriction requires adapting radically the evaluation algorithm which otherwise is inoperative. This occurs due to the compatibility function that does not take into account the use of the same path variable in multiple triple patterns. As for SPARQLeR, the order of evaluation is very complex when using the PATHFILTER construct for filtering paths, and the result of the graph pattern depends upon constructing all paths (which may not be exhaustive due to the infinite number of paths that can be constructed for cycle RDF graphs) and then selecting those ones that match a regular pattern.

Example 3.4.2 *The following SPARQ2L query:*

```
SELECT ??path
WHERE  {
            ?x ??path ?x .
            ?z compound:name "Methionine" .
            PATHFILTER ( containsAny(??path,?z)).
       }
```

finds any feedback loop (i.e., non-simple path) that involves the compound Methionine.

In both cases, the proposal seems to add expressivity to PSPARQL, in particular due to the use of path variables. However, the lack of a clearly defined semantics raises questions about what should be the returned answers and this does not allow to assess the correctness and completeness of the proposed procedures. Moreover, the constraints in these two languages are simple, *i.e.,* restricted to testing the length of paths and testing if a given node is in the resulting path (to be elaborated on in the sequel).

A recent extension of SPARQL, called nSPARQL, to a restricted fragment of RDFS is proposed in [Arenas *et al.*, 2008]. This extension allows using nested regular expressions, *i.e.,* regular expressions extended with branching axis borrowed form XPath. The authors presented a formal syntax and semantics of their proposal. As shown in Chapter 8, regular expressions in SPARQL (as in the case of (C)PSPARQL) have the ability of capturing the semantics of the used RDFS fragment. In particular, (C)PSPARQL can express all the examples provided in [Arenas *et al.*, 2008] for demonstrating the expressivity of the proposed language. On the one hand, nSPARQL has axis for navigating on nodes and edges. On the other hand, CPSPARQL has constraints on traversed edges and nodes. It may be useful to put the two extensions together.

Other extensions to SPARQL include: SPARQL-DL [Sirin and Parsia, 2007] that extends SPARQL to support Description Logic semantic queries, SPARQL++ [Polleres *et al.*, 2007] extending SPARQL with external functions and aggregates which serves as a basis for declaratively describing ontology mappings, and iSPARQL [Kiefer *et al.*, 2007] extending SPARQL to allow for similarity joins which employ several different similarity measures.

3.5 Work on SPARQL

[Cyganiak, 2005] presents a relational model of SPARQL, in which relational algebra operators (join, left outer join, projection, selection, etc.) are used to model SPARQL SELECT clauses. The authors propose a translation system between SPARQL and SQL to make a correspondence between SPARQL queries and relational algebra queries over a single relation. [Harris and Shadbolt, 2005] presents an implementation of SPARQL queries in a relational database engine, in which relational algebra operators similar to [Cyganiak, 2005] are used. [de Bruijn *et al.*, 2005] addresses the definition of mapping for SPARQL from a logical point of view. [Franconi and Tessaris, 2005], in which we can find a preliminary for-

malization of the semantics of SPARQL, defines an answer set to a basic graph pattern query using partial functions. The authors use high level operators (Join, Optional, etc.) from sets of mappings to sets of mappings, but currently they do not have formal definitions for them, stating only their types. [Polleres, 2007] provides translations from SPARQL to Datalog with negation as failure, some useful extensions of SPARQL, like *set difference and nested queries*, are proposed. Finally, [Perez *et al.*, 2006] presents the semantics of SPARQL using traditional algebra, and gives complexity bounds for evaluating SPARQL queries. The authors use the graph pattern facility to capture the core semantics and complexities of the language, and discussed their benefits. We followed their framework to define the answer set to (C)PSPARQL queries.

[Corby and Faron-Zucker, 2007a] presents an implementations of the SPARQL query language in Corese search engine [Corby *et al.*, 2004]. In particular, it describes a graph homomorphism based algorithm for answers SPARQL queries that integrates SPARQL constraints during the search process (*i.e.,* while matching the query against RDF graphs). [Corby and Faron-Zucker, 2007b] presents a design pattern to handle contextual metadata hierarchically organized and modeled within RDF. The authors of [Corby and Faron-Zucker, 2007b] propose a syntactic extension to SPARQL to facilitate querying context hierarchies together with rewriting rules to return to standard SPARQL.

3.6 Comparison with other Query Languages

We have compared PSPARQL and CPSPARQL to other query languages based on [Haase *et al.*, 2004; Angles and Gutiérrez, 1995]. [Haase *et al.*, 2004] compares several RDF query languages using 14 distinct tests (or features). Among them were Path expression, Optional path and Recursion tests. The interpretation of these three tests is given respectively as follows: using graph patterns, optional graph patterns, and recursive expressions. To remove ambiguity with the interpretation of path or regular expressions given in this thesis, we rename the three tests to be: Graph pattern, Optional pattern, and Recursion (or Regular expression). From [Angles and Gutiérrez, 1995], we include the following features: Adjacent nodes, Adjacent edges, Fixed-length path, Degree of a node, Distance between nodes, and Diameter. We also add the following features: Regular expression variable, Constraints, Path variable, Constrained regular expression, Inverse path, and Non-simple path. We mean by "Regular expression variable" that the use of variables

in the predicates or regular expressions of graph patterns. The query languages are restricted to this feature when they allow the use of variables only in the atomic predicates. A simple path is a path whose nodes are all distinct. There were 8 query languages in the original comparison ([Haase *et al.*, 2004]) from which we choose RQL, RDQL, SeRQL, and Versa which seem to represent the most expressive languages for supporting the two types of querying paradigms (*i.e.,* path-based and relational-based models); we include G+, GraphLog, STRUQL, LOREL from [Angles and Gutiérrez, 1995]; and we add SPARQL, Corese, SPARQ2L, SPARQLeR and (C)PSPARQL.

In Table 3.1, columns represent query languages and rows represent features or queries. Moreover, we use - to denote that the feature has no support in the query language, ○ to denote that there exists a partial (restricted) support, and finally • to denote the full support of the feature.

Table 3.1 summarizes the main differences between the current SPARQL extensions, CPSPARQL and other query languages. Most of features allowed in SPARQL extensions are also supported in CPSPARQL. Note that SPARQLeR (respectively, SPARQ2L) allows using SPARQL constraints (respectively, using path constraints like ContainsANY and ContainsALL) for a posteriori filtering paths. For example, checking the existence of regular pattern in a given path, and checking the existence of a node in the path. We conjuncture that we can emulate these constraints using constrained regular expressions of CPSPARQL. CPSPARQL and SPARQ2L are the only languages that supports non-simple paths. However, the algorithms in SPARQ2L are not complete for non-simple paths, and it has no support of inverse paths (inverse regular expressions).

As we can see in Table 3.1, there are a lot of features in SPARQL and its extensions that cannot be expressed in the current languages like G+, GraphLog, and others.

3.7 Conclusion

As shown in this chapter through the use of examples, SPARQL allows to ask more sophisticated queries than the consequence test. But many types of queries remains inexpressible. The development of the SPARQL recommendation has not prevented many extensions to be proposed. We have even proposed our own extension, which is not reducible to any of the above proposals (see examples in Chapters 5 and 6). It will be detailed further on in the subsequent chapters.

	SPARQL	Corese	SPARQ2L	SPARQLeR	(P/CP)SPARQL	G+	GraphLog	STRUQL	RDQL	SeRQL	Versa	RQL	LOREL
Graph pattern	●	●	●	●	●/●	●	●	●	●	●	●	●	●
Optional pattern	●	●	●	●	●/●	-	-	-	-	●	●	○	-
Union	●	●	●	●	●/●	-	-	-	-	●	●	●	●
Constraints	●	●	●	●	●/●	-	-	-	●	●	●	●	●
Difference	●	●	●	●	●/●	-	-	-	-	●	○	●	-
Quantification	-	-	-	-	-/-	-	-	-	-	●	-	●	●
Aggregation	-	○	-	-	○/○	-	-	-	-	-	●	●	●
Reification	●	●	●	●	●/●	-	-	-	○	●	○	○	-
Collections and Containers	○	○	○	○	○/○	-	-	-	○	○	○	●	-
Namespace	●	●	●	●	●/●	-	-	-	○	●	-	●	-
Language	●	●	●	●	●/●	-	-	-	-	●	-	-	-
Lexical space	●	●	●	●	●/●	-	-	-	●	●	●	●	-
Value space	●	●	●	●	●/●	○	○	○	○	●	-	●	●
Entailment	-	●	-	○	●/●	-	-	-	○	●	-	●	-
Recursion (Regular expression)	-	-	●	●	●/●	●	●	●	-	-	●	○	●
Regular expression variable	-	-	-	-	●/●	●	●	●	-	-	○	○	●
Constrained regular expression	-	-	○	○	-/●	-	-	-	-	-	-	-	-
Fixed-length path	-	●	○	○	●/●	●	●	●	○	○	-	●	●
Path variable	-	●	●	●	●/●	-	-	-	-	-	-	-	●
Inverse Path	-	-	-	●	-/●	●	●	-	-	-	-	-	-
Non-simple path	-	-	●	-	●/●	-	-	-	-	-	-	-	-
Adjacent nodes	●	●	●	●	●/●	●	●	●	○	○	○	○	●
Adjacent edges	●	●	●	●	●/●	●	●	●	○	-	○	○	○
Degree of a node	-	-	-	-	-/-	●	●	●	-	-	-	○	-
Distance between nodes	-	-	-	-	-/-	●	●	●	-	-	-	-	-
Diameter	-	-	-	-	-/-	●	●	●	-	-	-	-	-

Table 3.1: Comparison of query languages for graphs: white circle for partial (restricted) support, a dash for no support, and full circle for full support.

Part II

Research Work

A General Graph Framework with Paths

4

Contents

4.1 PRDF Syntax 42
4.1.1 Regular languages 43
4.1.2 PRDF graphs 46
4.2 PRDF Semantics 47
4.2.1 Interpretations and models 47
4.2.2 Satisfiability and canonical models 49
4.2.3 PRDF-GRDF entailment 51
4.3 Querying RDF with PRDF Graphs 51
4.3.1 Inference mechanism: path-based homomorphisms . . 52
4.3.2 Complexity of PRDF homomorphism 56
4.4 Containment of PRDF Queries 58
4.4.1 Query containment–definition 58
4.4.2 Containment and canonical models 59
4.4.3 Query containment for restricted PRDF queries 61
4.5 Conclusion 64

Introduction

Some query languages, such as SPARQL, are based upon RDF semantics, and use the RDF consequence to define answers over RDF graphs. Such query languages, as they are edge-based, lacks the ability of expressing variable length paths. The following are examples of applications requiring recursive queries: finding the ancestors of a person having a French nationality; finding pairs of capital cities con-

nected by a sequence of flights; finding pairs of persons knowing each other (*i.e.,* having a sequence of knows relations).

To overcome this limitation, we present in this chapter an extension of RDF with regular path expressions, called PRDF (short of Path RDF). This extension will be made in general, parametrized using language generators without grounding it to a specified language. Regular expressions will be used as a running example to illustrate the extension. The primary advantage of this generality is that the soundness and completeness (Theorem 4.3.5) does not depend upon the regular language used for expressing paths. It also permits language designers to decide which fragments, or more precisely operators, to be used for expressing paths.

For this extension of RDF, we present its abstract syntax in Section 4.1 and its semantics that extends RDF model-theoretic semantics in Section 4.2. It should be noted that this extension of RDF (PRDF) is made to be used mainly for defining the PSPARQL query language (our extension of SPARQL) and not for expressing knowledge (though it can be used for that purpose). Hence, for those readers who do not want to see the semantic justification of this extension and prefer reading it in a purely syntactic way, they can skip Section 4.2 (PRDF semantics) and trust Theorem 4.3.5 for grounding semantically our proposal.

An inference mechanism for answering PRDF graphs over RDF graphs will be presented in Section 4.3. Finally, we introduce the containment problem for PRDF graphs Section 4.4.

4.1 PRDF Syntax

In GRDF, arcs can be labeled by urirefs or variables. The PRDF language extends GRDF naturally to allow using path expressions as labels for arcs, *i.e.,* as predicate, in PRDF graphs. Each path expression encodes a set of words, called a regular language. The path expression $(\texttt{ex:train} \mid \texttt{ex:plane} \mid \texttt{ex:bus})^+$, for example, encodes sequences of trains, planes and buses.

So, to define the syntax of PRDF, we first need to introduce regular languages, then we use an abstract notion, a generator, to express such languages. A particular case for this set is the set of regular expressions, which will be used as a running example. The instantiation of PRDF to the set of regular expressions will be used in Chapter 5 to extend the SPARQL query language.

4.1.1 Regular languages

Words and languages

Let Σ be an alphabet. A *language* over Σ is a subset of Σ^*: its elements are sequences of elements of Σ called *words*. A (non empty) word $\langle a_1, \ldots, a_k \rangle$ is denoted by $a_1 \cdot \ldots \cdot a_k$. If $A = a_1 \cdot \ldots \cdot a_k$ et $B = b_1 \cdot \ldots \cdot b_q$ are two words over Σ, then $A \cdot B$ is the word over Σ defined by $A \cdot B = a_1 \cdot \ldots \cdot a_k \cdot b_1 \cdot \ldots \cdot b_q$. For example, if $\Sigma = \{$`ex:daughter`, `ex:son`$\}$, then $L = ($`ex:daughter` $\cup$ `ex:son`$)^*$ = $\{$`ex:daughter`, `ex:son`, `ex:daughter`$\cdot$`ex:son`, $\ldots\}$ is the regular language constructed over Σ.

One possible way to define regular languages is through the use of regular expressions as they are simple and compact for generating such languages. But, for doing the same task, one might use other means such as automaton or regular grammars. To not restrict our framework to a specific mean, we use the term generator to express a regular language.

Generators

We call a *generator* over Σ any object that can be used to specify a regular language over Σ. If R is such a generator, we note $L^*(R)$ the language specified by R (named language generated by R).

Since arcs of GRDF graphs can be only urirefs and variables, regular languages will be defined over the set of urirefs and variables, *i.e.,* $\Sigma \subseteq \mathcal{U} \cup \mathcal{B}$. The existence of a variable in the generator means that there exists something, and hence it can be replaced or mapped by any element of the alphabet, and one can define the language generated by a generator that contains variables using maps as given in the following definition. In which, the mapped value of a repeated occurrence of the same variable is ensured to be the same via maps.

Definition 4.1.1 *Let Σ be an alphabet, X be a set of variables, R be a generator over $\Sigma \cup X$, and μ be a map from $\Sigma \cup X$ to $\Sigma \cup X$. If $m = a_1 \cdot \ldots \cdot a_k \in (\Sigma \cup X)^*$, we note $\mu(m) = \mu(a_1) \cdot \ldots \cdot \mu(a_k)$, and $\mu(R)$ is the generator such that $L^*(\mu(R))$= $\{\mu(m) \mid m \in L^*(R)\}$.*

For example, if $\Sigma \cup X = \{$`ex:daughter`,`?X`$\}$, R be a generator over $\Sigma \cup X$, and $\mu = \{$`?X`$\leftarrow$`ex:friend`$\}$, then $L^*(\mu(R)) = ($`ex:daughter`, `ex:friend`$)^*$ is the language generated by $\mu(R)$.

In what follows, we use $\mathcal{R}(\Sigma)$ to denote an abstract infinite set of generators constructed over Σ.

Example: regular expression patterns

Regular expressions are the usual way for expressing path queries [Cruz *et al.*, 1987; Cruz *et al.*, 1988; Buneman *et al.*, 1996; Abiteboul *et al.*, 1997; de Moor and David, 2003; Liu *et al.*, 2004]. They can be used for defining regular languages over Σ.

Definition 4.1.2 (Regular expression) *Let Σ be an alphabet, the set $\mathcal{R}(\Sigma)$ of regular expressions is inductively defined by:*

- $\forall a \in \Sigma$, $a \in \mathcal{R}(\Sigma)$ *and* $!a \in \mathcal{R}(\Sigma)$*;*
- $\Sigma \in \mathcal{R}(\Sigma)$*;*
- $\epsilon \in \mathcal{R}(\Sigma)$*;*
- *If* $A \in \mathcal{R}(\Sigma)$ *and* $B \in \mathcal{R}(\Sigma)$ *then* $A|B$, $A \cdot B$, A^*, $A^+ \in \mathcal{R}(\Sigma)$.

such that $!a$ is the complement of a over Σ, $A|B$ denotes the disjunction of A and B, $A \cdot B$ the concatenation of A and B, A^* the Kleene closure, and A^+ the positive closure.

We have restricted regular expressions to atomic negation in order to have a reasonable time complexity in the query language that we are building, and to avoid its application to variables which have no meaning. However, the semantics, soundness and completeness results as well as the algorithms defined throughout this thesis still work with non-atomic regular expressions [Alkhateeb *et al.*, 2007].

More general forms of regular expressions are the ones that include variables, we call them *regular expression patterns*. Their combined power and simplicity contribute to their wide use in different fields. For example, in [de Moor and David, 2003], in which they are called *universal regular expressions*, they are used for compiler optimizations. In [Liu *et al.*, 2004], they are called *parametric regular expressions*, and are used for program analysis and model checking. The use of variables in regular expression patterns is different from the use of variables in Unix ("regular expressions with back referencing" in [Aho, 1980]). A variable appearing in a regular expression pattern matches any symbol of the alphabet or any variable, while a variable in regular expressions with back referencing can match strings. Matching strings with regular expressions with back referencing has been shown to be NP-complete [Aho, 1980].

The use of such patterns is necessary to generalize SPARQL that allows the use of variables in the predicate position of basic graph patterns.

Definition 4.1.3 (Regular expression pattern) *Let Σ be an alphabet, X be a set of variables, the set $\mathcal{RE}(\Sigma, X)$ of regular expression patterns is inductively defined by:*

- $\forall a \in \Sigma$, *then* $a \in \mathcal{RE}(\Sigma, X)$ *and* $!a \in \mathcal{R}(\Sigma, X)$;
- $\forall x \in X$, $x \in \mathcal{RE}(\Sigma, X)$;
- $\# \in \mathcal{RE}(\Sigma, X)$;
- $\Sigma \in \mathcal{RE}(\Sigma, X)$;
- $\epsilon \in \mathcal{RE}(\Sigma, X)$;
- *If* $A \in \mathcal{RE}(\Sigma, X)$ *and* $B \in \mathcal{RE}(\Sigma, X)$ *then* $A|B$, $A \cdot B$, A^*, $A^+ \in \mathcal{RE}(\Sigma, X)$.

With the absence of maps, the language generated by a regular expression pattern R, denoted by $L^*(R)$, is given in the following definition.

Definition 4.1.4 (Language defined by a regular expression pattern) *Let Σ be an alphabet, X be a set of variables, and $R, R' \in \mathcal{RE}(\Sigma, X)$ be regular expression patterns. $L^*(R)$ is the set of words of $(\Sigma \cup X)^*$ defined by:*

$$
\begin{aligned}
L^*(\epsilon) &= \{\epsilon\};\\
L^*(a) &= \{a\};\\
L^*(!a) &= \Sigma \setminus \{a\};\\
L^*(x) &= \Sigma \cup X;\\
L^*(\#) &= \Sigma \cup X;\\
L^*(\Sigma) &= \Sigma;\\
L^*(!R) &= \Sigma^* \setminus L^*(R_1);\\
L^*(R \mid R') &= \{w \mid w \in L^*(R) \cup L^*(R')\};\\
L^*(R \cdot R') &= \{w \cdot w' \mid w \in L^*(R) \text{ and } w' \in L^*(R')\};\\
L^*(R^+) &= \{w_1 \cdot \ldots \cdot w_k \mid \forall i \in [1 \ldots k], w_i \in L^*(R)\};\\
L^*(R^*) &= \{\epsilon\} \cup L^*(R^+).
\end{aligned}
$$

With regard to a more traditional definition of the language generated by a regular expression, our definition ranges over $\Sigma \cup X$. This is necessary because variables may match variables in GRDF graphs. In the context of PRDF this also pre-

serves the opportunity to define an order between PRDF[$\mathcal{R}$] graphs, *i.e.,* query containment. Note that the difference between # and a variable is that each occurrence of #[1] can be mapped to a different term while all occurrences of the same variable is mapped to the same term. For example, $L^*(?x) = \{u, u \cdot \ldots \cdot u \mid u \in (\Sigma \cup X)\}$ while $L^*(\#) = \{u_1, u_1 \cdot \ldots \cdot u_n \mid u_i \in (\Sigma \cup X)\}$.

4.1.2 PRDF graphs

Since arcs in GRDF graphs are labeled by the elements of $\mathcal{U} \cup \mathcal{B}$, path queries will be defined by generators over $\Sigma = \mathcal{U} \cup \mathcal{B}$.

Definition 4.1.5 (PRDF graph) *A* PRDF[$\mathcal{R}$] triple *is an element of* $\mathcal{T} \times \mathcal{R}(\mathcal{U}, \mathcal{B}) \times \mathcal{T}$. *A PRDF*[$\mathcal{R}$] *graph is a set of PRDF*[$\mathcal{R}$] *triples.*

Note that all PRDF[$\mathcal{R}$] graphs with atomic predicates are not necessarily RDF graphs. They can be a generalization of RDF graphs with blanks as predicates, as called *generalized RDF graphs* [Horst, 2005]. A PRDF[$\mathcal{R}$] graph can be represented graphically as a directed labeled graph whose arcs are labeled by elements of $\mathcal{R}(\mathcal{U}, \mathcal{B})$.

The set of terms in a PRDF[$\mathcal{R}$] graph is the set of all elements appearing as subjects and objects including all atomic elements in each language generator.

Notations

Let R be a generator, $u \in \mathcal{U}(R)$ if $u \in U$ and U is the smallest set such that $R \in \mathcal{R}(U, \mathcal{B})$ (*i.e.,* $\mathcal{U}(R)$ is the set of urirefs appearing in R). In the same way, $b \in \mathcal{B}(R)$ if $b \in B$ and B is the smallest set such that $R \in \mathcal{R}(\mathcal{U}, B)$ (*i.e.,* $\mathcal{B}(R)$ is the set of blanks appearing in R). Let G be a PRDF[$\mathcal{R}$] graph, $pred(G)$ is the set of generators appearing as a predicate in a triple of G. Let $\mathcal{UB}(\mathcal{R}) = \mathcal{U}(R) \cup \mathcal{B}(R)$, $\forall R \in pred(G)$. Then $term(G) = subj(G) \cup \mathcal{UB}(\mathcal{R}) \cup obj(G)$.

Example 4.1.6 *For example, the following PRDF*[$\mathcal{RE}$] *graph represented graphically by the graph* P *of Figure 4.2:*

[1] # can be replaced by a more elegant rdfs property, `rdfs:anyRelation`. This new property can be interpreted by $I_{EXT}(\iota(\texttt{rdfs:anyrelation})) = I_R \; X \; I_R$

```
{
   ⟨ ?b1          foaf:name          "Faisal" ⟩,
   ⟨ ?b1   (ex:daughter|ex:son)+·?b5  ?person ⟩,
   ⟨ ?person      foaf:knows         ?b1 ⟩,
   ⟨ ?person      foaf:name          ?name ⟩,
   ⟨ ?person      foaf:mbox          ?mbox ⟩
}
```

when used as a query searches among the relatives of Faisal's descendants, the names and email addresses of people who know Faisal. Recall that $\mathcal{RE}$ is the set of regular expression patterns.

4.2 PRDF Semantics

The PRDF semantics extends the RDF semantics to allow expressing paths of arbitrary length.

4.2.1 Interpretations and models

Since the terminology of RDF is the one used for PRDF, RDF interpretations remain unchanged in the case of PRDF. However, an RDF interpretation must satisfy specific conditions to be a model for a PRDF[$\mathcal{R}$] graph. These conditions are the transposition of the classical path semantics within RDF semantics.

Definition 4.2.1 (Support of a generator) *Let $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ be an interpretation of a vocabulary $V = U \cup L$, ι' be an extension of ι to $B \subseteq \mathcal{B}$, and $R \in \mathcal{R}(U, B)$. Let $w = a_1 \cdot \ldots \cdot a_k$ be a word of $L^*(R)$. A sequence $(r_0, \ldots, r_k)$ of resources of I_R is called a proof of w in I according to ι' iff one of the following conditions holds:*

(i) w is the empty word and $r_i = r_j$ $(0 \leq i, j \leq k)$; or

(ii) $\langle r_{i-1}, r_i \rangle \in I_{EXT}(\iota'(a_i))$ $(\forall 1 \leq i \leq k)$, otherwise.

Instead of considering paths in RDF graphs, Definition 4.2.1 considers paths in the interpretations of PRDF[$\mathcal{R}$] graphs, *i.e.,* paths are now relating resources. This definition is the semantic substitute for the satisfaction of a regular expression pattern by two nodes (Definition 4.3.1). It has the same function: ensuring that

variables have only one image. This is achieved by the "extension to variables" (ι') which plays the same role as μ in Definition 4.3.1.

It is used in the following definition of PRDF models in which it replaces the direct correspondence that exists in RDF between a relation and its interpretation (see Definition 2.2.3), by a correspondence between a generator (for example, a regular expression pattern) and a sequence of relation interpretations. This allows to match variable length paths (for regular expression patterns, *e.g.* r^+).

Definition 4.2.2 (Model of a PRDF graph) *Let G be a PRDF$[\mathcal{R}]$ graph, and $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ be an interpretation of a vocabulary $V \supseteq \mathcal{V}(G)$. I is a PRDF model of G if and only if there exists an extension ι' of ι to $\mathcal{B}(G)$ such that for every triple $\langle s, R, o\rangle \in G$, there exists a sequence $T = (\iota'(s) = r_0, \ldots, r_k = \iota'(o))$ of resources of I_R and a word $w \in L^*(R)$ such that T is a proof of w in I according to ι'. (We also say that $\langle \iota'(s), \iota'(o)\rangle$ supports R in ι').*

This definition extends the definition of RDF models (Definition 2.2.3), and they are equivalent when all generators R are reduced to atomic terms, *i.e.,* urirefs or variables. Moreover, GRDF graphs are PRDF graphs with predicates restricted to atomic terms.

Proposition 4.2.3 *If G is a PRDF$[\mathcal{R}]$ graph with $pred(G) \subseteq \mathcal{U} \cup \mathcal{B}$, i.e., G is a GRDF graph, and I be an interpretation of a vocabulary $V \supseteq \mathcal{V}(G)$, then I is an RDF model of G (Definition 2.2.3) iff I is a PRDF model of G (Definition 4.2.2).*

Proof. We prove both directions of the proposition.
($\Rightarrow$) Suppose that $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ is an RDF model of G, then there exists an extension ι' of ι to $\mathcal{B}(G)$ such that $\forall\langle s, p, o\rangle \in G$, $\langle \iota'(s), \iota'(o)\rangle \in I_{EXT}(\iota'(p))$ (Definition 2.2.3). Since $pred(G) \subseteq \mathcal{U} \cup \mathcal{B}$, $\langle \iota'(s), \iota'(o)\rangle$ supports p in ι' (Definition 4.2.1) (with a word $w = p$), *i.e.,* I is also a PRDF model (Definition 4.2.2).
($\Leftarrow$) Suppose that $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ is a PRDF model of G, then there exists an extension ι' of ι to $\mathcal{B}(G)$ such that $\forall\langle s, p, o\rangle \in G$, $\langle \iota'(s), \iota'(o)\rangle$ supports p in ι' (Definition 4.2.2). Since $pred(G) \subseteq \mathcal{U} \cup \mathcal{B}$, $\epsilon \notin L^*(p)$. So there there exists a word of length $n = 1$ where $w \in L^*(p)$, $w = p$, and a sequence of resources of IR $\iota'(s) = r_0$, $\iota'(o) = r_1$ such that $\langle r_0, r_1\rangle \in I_{EXT}(\iota'(w))$ (Definition 4.2.1). So $\forall\langle s, p, o\rangle \in G$, $\langle \iota'(s), \iota'(o)\rangle \in I_{EXT}(\iota'(p))$ (by replacing r_0 with $\iota'(s)$, r_1 with $\iota'(o)$, and w with p). So I is also an RDF model (Definition 2.2.3).

Due to the use of the disjunction and negation operators in regular expressions, we may have a model of a given PRDF[$\mathcal{RE}$] graph that does interpret all its terms. As an example, consider the interpretation $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ defined by:
- $I_R = \{Paris, Lyon, train\}$;
- $I_P = \{train\}$;
- ι(ex:Paris) $= Paris$, ι(ex:Lyon) $= Lyon$, ι(ex:train) $= train$, and $I_{EXT}(train) = \{\langle Paris, Lyon\rangle\}$.

There is no interpretation of ex:plane in I, but it is a model of the graph defined by $\{\langle$ex:Paris (ex:train|ex:plane) ex:Lyon$\rangle\}$.

Definition 4.2.4 (Satisfiability and consequence) *A PRDF[$\mathcal{R}$] graph G is* satisfiable *iff it admits a model. A PRDF[$\mathcal{R}$] graph G' is a* consequence *of a PRDF[$\mathcal{R}$] graph G, noted $G \models_{PRDF} G'$, iff every model of G is also a model of G'.*

4.2.2 Satisfiability and canonical models

In this subsection, we give conditions under which a model is considered as a *canonical* model. Then, we prove that each PRDF graph is satisfiable by building such a model.

Definition 4.2.5 (Canonical Model of a PRDF graph) *Let G be a PRDF[$\mathcal{R}$] graph, $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ be an interpretation of a vocabulary $V \supseteq \mathcal{V}(G)$, and ι' be an extension of ι to $\mathcal{B}(G)$. I is called an ι'-canonical model if:*

- *I contains one proof for each $\langle s, R, o\rangle \in G$ of a word $w \in L^*(R)$ in I according to ι', i.e., there exists $T = (\iota'(s) = r_0, \ldots, r_k = \iota'(o))$ of resources of I_R and a word $w \in L^*(R)$ such that T is a proof of w in I according to ι'.*
- *Each resource $r_i \in I_R$ occurs exactly once as a first element in an extension of a property and exactly once as a second element of another property unless $r_i = \iota'(n)$ for some node $n \in nodes(G)$.*

Example 4.2.6 *The interpretation $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ defined by:*

- $I_R = \{Paris, Lyon, train, plane\}$
- $I_P = \{train, plane\}$
- ι(ex:Paris) $= Paris$, ι(ex:Lyon) $= Lyon$, ι(ex:train) $= train$, ι(ex:plane) $= plane$, $I_{EXT}(plane) = \{\langle Paris, Lyon\rangle, \langle Lyon, Grenoble\rangle\}$ and $I_{EXT}(train\ ) = \{\langle Paris, Lyon\rangle\}$

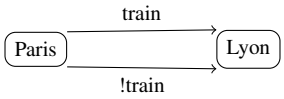

Figure 4.1: A PRDF graph with negation.

is not a canonical model of the PRDF graph $\{\langle$`ex:Paris` $($`ex:train`$|$`ex:plane`$)^+$ `ex:Grenoble`$\rangle\}$ *since the intermediate resource* $Lyon$ *belongs to the extensions of* $plane$ *and* $train$*, which violates the definition of canonical models. Nonetheless, if we remove the pair* $\langle Paris, Lyon\rangle$ *from the extension of* $plane$*, then we have a canonical model of the given graph.*

Proposition 4.2.7 (Satisfiability) *Each PRDF*$[\mathcal{R}]$ *graph* G *is satisfiable*

Proof. Let G be a PRDF$[\mathcal{R}]$ graph. To prove that G is satisfiable, we build a canonical model as follows:

1. Build a graph G' by replacing each triple $\langle s, R, p\rangle$ in G by a set of triples $\{\langle s, p_1, v_1\rangle, \ldots, \langle v_{n-1}, p_n, o\rangle\}$ such that $p_1 \cdot \ldots \cdot p_n$ is an arbitrary word in the language generated by R, and $v_{i's}$ are all new distinct variables.

2. G' is satisfiable since it admits a model (constructing the isomorphic model of G' see Proposition 2.2.5). Hence, every PRDF$[\mathcal{R}]$ graph G is satisfiable.

As in the case of (G)RDF, every PRDF$[\mathcal{R}]$ graph is satisfiable (if we consider only simple semantics). This can hurt the intuition since the graph G of Figure 4.1 admits a model due to the interpretation of path negation which differs from its interpretation in first-order logic. The triple $\langle$`ex:Paris !ex:train ex:Lyon`$\rangle$ is read as "Paris and Lyon are in a relation other than train" and not as "Paris and Lyon are not related by train". If we consider the second interpretation and replace Lyon with a variable, then we can never find a city which is related and not related by train at the same time.

Definition 4.2.8 *Let* $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ *be an interpretation of a vocabulary* V*,* G *and* H *be two PRDF*$[\mathcal{R}]$ *graphs such that* $\mathcal{V}(G) \subseteq V$ *and* $\mathcal{V}(H) \subseteq V$*, and* ι' *be a canonical model of* G*. An extension* ι'' *of* ι *to* $\mathcal{B}(H)$ *is called an* ι'*-model of* H *if:*

– ι'' *is a model of* H*;*
– $\forall n_1 \in nodes(H)$, $\exists n_2 \in nodes(G)$ *with* $\iota''(n_1) = \iota'(n_2)$*; and*
– $\forall b \in (\mathcal{B}(H) \cap \mathcal{B}(G))$, $\iota''(b) = \iota'(b)$.

The second item is necessary when the two graphs G and H have different variable names, and this item can be omitted by variable renaming.

The canonical consequence (entailment) between PRDF graphs is defined as in the usual way, and we use $\models^c_{\text{PRDF}}$ to denote such consequence.

Definition 4.2.9 (Canonical consequence) *Let G and H be two PRDF$[\mathcal{R}]$ graphs of a vocabulary V, then G canonically entails H, denoted by $\models^c_{PRDF}$, iff every ι'-canonical model of G is also ι'-model of H.*

4.2.3 PRDF-GRDF entailment

For the purpose of defining a query language, we need to deal with the PRDF-GRDF ENTAILMENT problem:

$\mathcal{R}$-PRDF-GRDF ENTAILMENT
Instance: a GRDF graph G and a PRDF$[\mathcal{R}]$ graph H.
Question: Does $G \models_{\text{PRDF}} H$?

This problem is at least NP-hard, since it contains SIMPLE RDF ENTAILMENT, an NP-complete problem. However, when the entailed graph, *i.e.,* the query, is *ground*, this problem can be decided in NLOGSPACE.

Theorem 4.2.10 *Let G be a GRDF graph and H be a* ground *PRDF$[\mathcal{R}]$ graph, then $\mathcal{RE}$-*PRDF-GRDF ENTAILMENT *is in* NLOGSPACE.

The following section shows the complexity of the latter problem through the equivalence between $\mathcal{R}$-PRDF-GRDF ENTAILMENT and $\mathcal{R}$-PRDF-GRDF HOMOMORPHISM.

4.3 Querying RDF with PRDF Graphs

This section presents a particular homomorphism for checking if a PRDF graph is a consequence of an RDF graph. It will be then used for answering PRDF graphs over RDF graphs.

4.3.1 Inference mechanism: path-based homomorphisms

PRDF can be used as a stand alone language for querying (G)RDF knowledge bases. An answer to a PRDF query Q over a (G)RDF knowledge base will be a particular map from Q into G, we called it a PRDF homomorphism.

PRDF homomorphisms extend RDF homomorphisms to deal with nodes connected with regular language generators (for example, regular expression patterns), that can be mapped to nodes connected by paths.

Definition 4.3.1 (Path word) *Let G be a GRDF graph of a vocabulary $V = U \cup B$, and $R \in \mathcal{R}(\mathcal{U}, \mathcal{B})$ be a generator such that $\mathcal{U}(R) \subseteq V$. Let $\mu : \mathcal{B}(R) \rightarrow V$ be a map from the variables of R to V, and $w = a_1 \cdot \ldots \cdot a_k$ be a word of $L^*(R)$. A sequence $(x_0, \ldots, x_k)$ of nodes of G is called a* path *of w in G according to μ iff $\forall 1 \leq i, j \leq k$ one of the following conditions holds:*

- *w is the empty word and $x_i = x_j$; or*
- *$\langle x_{i-1}, \mu(a_i), x_i \rangle \in G$, otherwise.*

We also say that $\langle x_0, x_k \rangle$ satisfies w in G according to μ.

A path of nodes $(x_0, \ldots, x_k)$ is said to be *simple* if all nodes are distinct (*i.e.,* each x_i occurs once in the path).

Language generators (*e.g.* regular expression patterns) can be used alone as queries. An answer to a generator R in an RDF graph G will be a triple $\langle x_0, x_k, \mu \rangle$ such that $\mu : \mathcal{U}(R) \rightarrow \mathcal{V}(G)$ is a map from the variables of R into terms of G and $\langle x_0, x_k \rangle$ is a pair of nodes of G that satisfies a word $w \in L^*(R)$ in G according to μ.

Example 4.3.2 *Consider the RDF graph G of Figure 2.2, and the regular expression pattern $R = (\texttt{ex:son}|\texttt{ex:daughter})^+ \cdot \texttt{?b5}$. Intuitively, this regular expression pattern encodes the paths from the entity x to the entity y such that y has a relation, by any predicate, of a descendant of x. The answers to R are:*

$$\begin{array}{lll}
\langle\ \texttt{ex:c1} & \texttt{ex:c3} & \{(\texttt{?b5},\texttt{ex:son})\}\rangle, \\
\langle\ \texttt{ex:c1} & \texttt{ex:person1} & \{(\texttt{?b5},\texttt{ex:friend})\}\rangle, \\
\langle\ \texttt{ex:c1} & \texttt{ex:person2} & \{(\texttt{?b5},\texttt{ex:friend})\}\rangle, \\
\langle\ \texttt{ex:c1} & \texttt{ex:person3} & \{(\texttt{?b5},\texttt{ex:friend})\}\rangle,
\end{array}$$

Definition 4.3.3 (PRDF homomorphism) *Let G be a (G)RDF graph and H be a PRDF$[\mathcal{R}]$ graph. A* PRDF homomorphism *from H into G is a map $\pi : \mathcal{T}(H) \rightarrow$*

$\mathcal{T}(G)$ that preserves the paths, i.e., $\forall\langle s, R, o\rangle \in H$, there exists a sequence $T = (\pi(s), \ldots, \pi(o))$ of nodes of G and a word $w \in L^*(R)$ such that T is a path of w in G according to π.

Example 4.3.4 *Figure 4.2 shows a PRDF homomorphism from the PRDF graph P into the RDF graph G. Note that the path satisfying the regular expression pattern of P is one of those given in Example 4.3.2.*

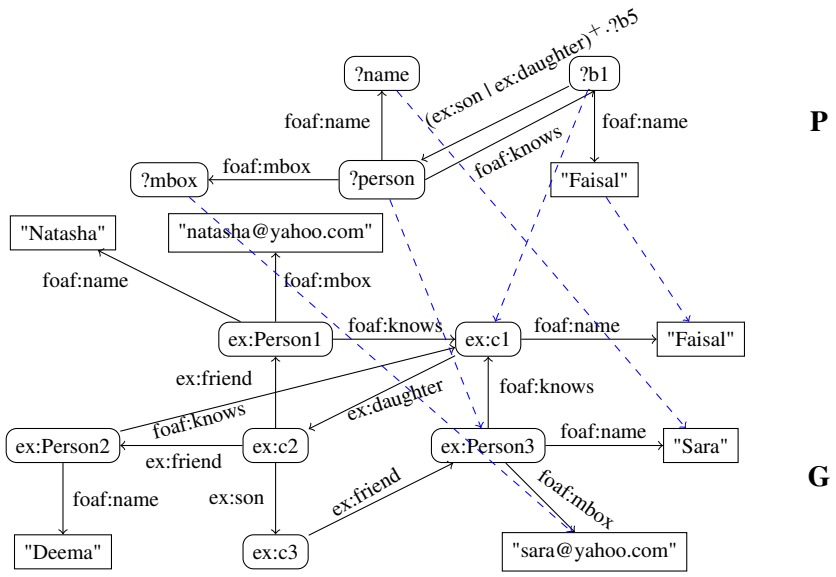

Figure 4.2: A PRDF homomorphism from a PRDF graph to a GRDF graph represented in dashed lines.

The existence of a PRDF homomorphism is exactly what is needed for deciding entailment between GRDF and PRDF$[\mathcal{R}]$ graphs.

Theorem 4.3.5 *Let G be a GRDF graph, and H be a PRDF$[\mathcal{R}]$ graph, then there is a PRDF homomorphism from H into G iff $G \models_{PRDF} H$.*

We have proven Theorem 4.3.5 via a transformation to hypergraphs following the proof framework in [Baget, 2005]. Since this requires a long introduction to hypergraphs, we prefer here to give a simple direct proof to Theorem 4.3.5.

Proof. We prove both directions of the theorem.

($\Rightarrow$) For the if-part, we suppose that there exists a PRDF homomorphism from H into G, $\pi : term(H) \rightarrow term(G)$. We want to prove that $G \models_{\text{PRDF}} H$, *i.e.,* that

every model of G is a model of H. Consider the interpretation I of a vocabulary $V = U \cup L$.

If I is a model of G, then there exists an extension I' of I to $\mathcal{B}(G)$ such that $\forall \langle s, p, o\rangle \in G$, $\langle I'(s), I'(o)\rangle \in I_{EXT}(I'(p))$ (Definition 2.2.3). We want to prove that I is also a model of H, *i.e.,* that there exists an extension I'' of I to $\mathcal{B}(H)$ such that $\forall \langle s, R, o\rangle \in H$, $\langle I''(s), I''(o)\rangle$ supports R in I''.

Let us define the map $I'' = (I' \circ \pi)$, and show that I'' verifies the following properties:

1. I is an interpretation of $\mathcal{V}(H)$.

2. I'' is an extension to variables of H, *i.e.,* $\forall x \in \mathcal{V}(H)$, $I''(x) = I(x)$ (Definition 2.2.2).

3. I'' satisfies the conditions of PRDF models (Definition 4.2.2), *i.e.,* , for every triple $\langle s, R, o\rangle \in H$, the pair of resources $\langle I''(s), I''(o)\rangle$ supports R in I''.

Now, we prove the satisfaction of these properties:

1. Since each term $x \in \mathcal{V}(H)$ is mapped by π to a term $x \in \mathcal{V}(G)$ and I interprets all $x \in \mathcal{V}(G)$, I interprets all $x \in \mathcal{V}(H)$.

2. $\forall x \in \mathcal{V}(H)$, $I''(x) = (I' \circ \pi)(x)$ (definition of I''). $I''(x) = I'(x)$ (since $\pi(x) = x$ by Definition 4.3.3). Hence, $I''(x) = I(x)$ (Definition 2.2.2).

3. It remains to prove that for every triple $\langle s, R, o\rangle \in H$, the pair of resources $\langle I''(\pi(s)), I''(\pi(o))\rangle$ supports R in I'' (by Definition 4.2.1):

   (i) If the empty word $\epsilon \in L^*(R)$ and $\pi(s) = \pi(o) = y$ ($y \in term(G)$, Definition 4.3.3), then $I''(s) = (I' \circ \pi)(s) = I'(y)$, and $I''(o) = (I' \circ \pi)(o) = I'(y)$. So $I''(s) = I''(o) = I'(y)$. Hence, $\langle I''(s), I''(o)\rangle$ supports R in I'' (Definition 4.2.2).

   (ii) If $\exists \langle n_0, p_1, n_1\rangle, \ldots, \langle n_{k-1}, p_k, n_k\rangle$ in G such that $n_0 = \pi(s)$, $n_k = \pi(o)$, and $p_1 \cdot \ldots \cdot p_k \in L^*(\pi(R))$ (*cf.* Definition 4.3.3). It follows that $\langle I'(\pi(s)), I'(n_1)\rangle \in I_{EXT}(I'(p_1))$, $\ldots$, $\langle I'(n_{k-1}), I'(\pi(o))\rangle \in I_{EXT}(I'\ (p_k))$ (Definition 2.2.3). So the two resources $\langle I'(\pi(s)), I'(\pi(o))\rangle$ supports $\pi(R)$ in I'. $\langle I'(\pi(s)), I'(\pi(o))\rangle$ supports $\pi(R)$ in I'' (since $I'' = (I' \circ \pi)$, we have $\forall x \in term(H)$, $I''(x) = I'(\pi(x))$ and $\pi(x) \in term(G)$. Moreover, we can choose every variable b appearing

in H to be interpreted by the resource of $\pi(b)$). Hence, $\langle I''(s), I''(o)\rangle$ supports R in I'' (since for every word $w \in \pi(R), w \in R$).

($\Leftarrow$) Suppose that $G \models_{\text{PRDF}} H$. We want prove that there is a PRDF homomorphism from H into G. Every model of G is also a model of H. In particular, the isomorphic model $I_{iso} = \langle I_R, I_P, I_{EXT}, \iota\rangle$ of G, where there exists a bijection ι between $term(G)$ and I_R (*cf.* Proposition 2.2.5). ι is an extension of I_{iso} to $\mathcal{B}(G)$ such that $\forall\langle s, p, o\rangle \in G$, $\langle \iota(s), \iota(o)\rangle \in I_{EXT}(\iota(p))$ (Definition 2.2.3). Since I_{iso} is a model of H, there exists an extension I' of ISO to $\mathcal{B}(H)$ such that $\forall\langle s, R, o\rangle$, $\langle I'(s), I'(o)\rangle$ supports R in I' (Definition 4.2.2). Let us consider the function $\pi = (\iota^{-1} \circ I')$. To prove that π is a PRDF homomorphism from H into G, we must prove that:

1. π is a map from $term(H)$ into $term(G)$;
2. $\forall x \in \mathcal{V}(H)$, $\pi(x) = x$;
3. $\forall\langle s, R, o\rangle \in H$, either
   (i) the empty word $\epsilon \in L^*(R)$ and $\pi(s) = \pi(o)$; or
   (ii) $\exists\langle n_0, p_1, n_1\rangle, \ldots, \langle n_{k-1}, p_k, n_k\rangle$ in G such that $n_0 = \pi(s)$, $n_k = \pi(o)$, and $p_1 \cdot \ldots \cdot p_k \in L^*(\pi(R))$.

1. Since I' is a map from $term(H)$ into I_R and ι^{-1} is a map from I_R into $term(G)$, $\pi = (\iota^{-1} \circ I')$ is clearly a map from $term(H)$ into $term(G)$ ($term(H) \xrightarrow{I'} I_R \xrightarrow{\iota^{-1}} term(G)$).

2. $\forall x \in \mathcal{V}(H)$, $I'(x) = \iota(x)$ (Definition 2.2.2 and Proposition 2.2.5). $\forall x \in \mathcal{V}(H)$, $(\iota^{-1} \circ I')(x) = (\iota^{-1} \circ \iota)(x) = x$.

(3i) If $\epsilon \in L^*(R)$ and $I'(s) = I'(o) = r \in I_R$ (Definition 4.2.1), then $\pi(s) = (\iota^{-1} \circ I')(s) = \iota^{-1}(r)$, and $\pi(o) = (\iota^{-1} \circ I')(o) = \iota^{-1}(r)$. So $\pi(s) = \pi(o) = \iota^{-1}(r)$.

(3ii) If there exists a word of length $n \geq 1$ such that $w = a_1 \cdot \ldots \cdot a_n$ where $w \in L^*(R)$ and $a_i \in U \cup \mathcal{B}(G)$ ($1 \leq i \leq k$), and there exists a sequence of resources of I_R $I'(s) = r_0, \ldots, r_k = I'(o)$ such that $\langle r_{i-1}, r_i\rangle \in I_{EXT}(I'(a_i))$, $1 \leq i \leq k$ (Definition 4.2.1). It follows that $\langle n_{i-1}, p_i, n_i\rangle \in G$ with $n_i = \iota^{-1}(r_i)$, and $p_i = (\iota^{-1} \circ I')(a_i)$ (construction of $I_{iso}(G)$ of Proposition 2.2.5). So $(\iota^{-1} \circ I')(s) = \iota^{-1}(r_0) = n_0$, $(\iota^{-1} \circ I')(o) = \iota^{-1}(r_k) = n_k$, and $p_1 \cdot \ldots \cdot p_k \in L^*((\iota^{-1} \circ I')(R))$.

4.3.2 Complexity of PRDF homomorphism

The definition of PRDF homomorphism is parameterized by the language generator $\mathcal{R}$ and subject to its satisfaction checking. To study the complexity of checking the existence of a PRDF homomorphism, we need first to associate to the path checking the decision problem called $\mathcal{R}$-PATH SATISFIABILITY, and defined as follows:

$\mathcal{R}$-PATH SATISFIABILITY
Instance: A GRDF graph G, two nodes x_0, x_k of G, and a generator $R \in \mathcal{R}(U, B)$, where $U \supseteq \mathcal{V}(G)$.
Question: Is there a map μ from $U \cup B$ to $term(G)$, a sequence $T = (x_0, \ldots, x_k)$ of nodes of G and a word $w \in L^*(R)$ such that T is a path of w in G according to π (*i.e.,* the pair $\langle x_0, x_k \rangle$ satisfies $L^*(\mu(R))$)?

$\mathcal{R}$-PRDF-GRDF HOMOMORPHISM
Instance: A PRDF[$\mathcal{R}$] graph H and a GRDF graph G.
Question: Is there a PRDF homomorphism from H into G?

The problem is at least NP-hard, since it contains SIMPLE RDF ENTAILMENT which is an NP-complete problem. Moreover, any solution can be checked by checking as many times as there is edges in the query an instance of the $\mathcal{R}$-PATH SATISFIABILITY problem. Hence, if $\mathcal{R}$-PATH SATISFIABILITY is in NP then $\mathcal{R}$-PRDF-GRDF HOMOMORPHISM is NP-complete.

Proposition 4.3.6 $\mathcal{RE}$-PATH SATISFIABILITY *in which* $B = \emptyset$ *(*$R \in \mathcal{RE}(U, B)$ *is a regular expression that does not contain variables) is in* NLOGSPACE *in* G *and* R*.*

Proof. The labels of paths between x_0 and x_k form a regular language P_{x_0,x_k} [Yannakakis, 1990]. So, construct a non-deterministic finite automaton A_G accepting the regular language P_{x_0,x_k} with initial state x_0 and final state x_k (G can be transformed to an equivalent NDFA in NLOGSPACE). Constructing a NDFA M accepting $L^*(R)$, the language generated by R, can be done in NLOGSPACE. Constructing the product automaton $\mathcal{P}$, that is, the intersection of A_G and M, can be done in NLOGSPACE in $|A_G| + |M|$. Checking if the pairs $\langle x_0, x_k \rangle$ satisfies $L^*(R)$ is equivalent to checking whether $L^*(\mathcal{P})$ is not empty, and each of these operations can be done in NLOGSPACE in $|\mathcal{P}|$ [Mendelzon and Wood, 1995;

Alechina *et al.*, 2003] (with the fact that the class of LOGSPACE transformations is closed under compositions [Balcazar *et al.*, 1988]). An automaton for the intersection of $L^*(R)$ with M is constructed by taking the product of the automaton for the two languages. That is, the states of the product automaton are of the form $\langle s, u\rangle$ such that s is a state of M and u is a node of G; and there exists a transition on letter a (respectively, letter b) from a state $\langle s, u\rangle$ to another state $\langle t, v\rangle$ if M has a transition on a (respectively, on letter $!a$[2]) from s to t and $\langle u, a, v\rangle \in G$ (respectively, $\langle u, b, v\rangle \in G$ and $b \neq a$). The construction is similar to the one presented in [Yannakakis, 1990] without atomic negation.

When regular expressions do not contain variables, there is no need to guess a map and the problem is reduced to the following decision problem [Mendelzon and Wood, 1995; Alechina *et al.*, 2003]:
$\mathcal{R}$-REGULAR PATH [Mendelzon and Wood, 1995]
Instance: A directed labeled graph G, two nodes x_0, x_k of G, and a regular expression pattern $R \in \mathcal{RE}(U)$.
Question: Does the pair $\langle x, y\rangle$ satisfies $L^*(R)$?

Proposition 4.3.7 $\mathcal{RE}$-PATH SATISFIABILITY *is in* NP.

Proof. $\mathcal{RE}$-PATH SATISFIABILITY is in NP, since each variable in the regular expression pattern R can be mapped (assigned) to p terms, where p denotes the number of terms appearing as predicates in G. If the number of variables in R is v, then there are (p^v) possible assignments (mappings) in all. Once an assignment of terms to variables is fixed, the problem is reduced to $\mathcal{RE}$-PATH SATISFIABILITY, where $\Sigma \subseteq \mathcal{U}$, which is in NLOGSPACE.

It follows that a non-deterministic algorithm needs to guess a map μ and check in NLOGSPACE if the pair $\langle x_0, x_k\rangle$ satisfies $L^*(\mu(R))$.

Theorem 4.3.8 *Let G be a GRDF graph and H be a* ground *PRDF*$[\mathcal{RE}]$ *graph, then* $\mathcal{RE}$-PRDF-GRDF HOMOMORPHSIM *is in* NLOGSPACE.

Proof. If H is ground, for each node x in H, $\pi(x)$ is determined in G. Then it remains to verify independently, for each triple $\langle s, R, o\rangle$ in H, if $\langle \pi(s), \pi(o)\rangle$=$\langle s, o\rangle$

[2] $!a$ is an atomic negation, *i.e.,* a negated uriref.

satisfies $\pi(R) = R$. Since each of these operations corresponds to the case of PATH SATISFIABILITY, in which $\Sigma \subseteq \mathcal{U}$ and $X = \emptyset$, the complexity of each of them is NLOGSPACE (see Proposition 4.3.6) (Since H is ground, R does not contain variables). So, the total time is also NLOGSPACE. Given the equivalence between PRDG-GRDF ENTAILMENT and checking the existence of PRDF homomorphism (Theorem 4.3.5), PRDF-GRDF ENTAILMENT is thus in NLOGSPACE.

From this result and the equivalence of $\mathcal{RE}$-PRDF-GRDF ENTAILMENT and $\mathcal{RE}$-PRDF-GRDF HOMOMORPHISM (Theorem 4.3.5), we conclude that the $\mathcal{RE}$-PRDF-GRDF ENTAILMENT problem is in NLOGSPACE.

4.4 Containment of PRDF Queries

A fundamental form of reasoning on queries is checking containment, *i.e.,* checking whether the answer to one query is a subset of answers of another one. It is useful in several contexts such as query optimizations, information integration, knowledge base verification, etc.

We introduce in this section the notion of query containment. Then we characterize the containment problem of PRDF graphs, and show the decidability of the problem. Finally, we provide a particular case in which the problem is NP-complete.

4.4.1 Query containment–definition

Informally, the problem of query containment is the problem of testing whether if answers to one query are all answers to another one. Let us use $\mathcal{S}(Q, G)$ to denote the set of answers of the query Q over the knowledge base G. This problem can be defined as follows:

Definition 4.4.1 (Query Containment) *Let Q and Q' be two queries. We say that Q is contained in Q', denoted by $Q \sqsubseteq Q'$, if and only if for all RDF knowledge base G then $\mathcal{S}(Q, G) \subseteq \mathcal{S}(Q', G)$. Q and Q' are equivalent, denoted by $Q \equiv Q'$, if $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$.*

We are interested sometimes in returning the values of a subset of the set of variables appearing in the query, and the following simple form could be used:

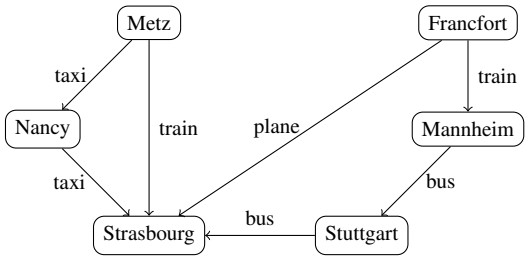

Figure 4.3: An RDF graph.

$$Q(\vec{X}) : -P$$

where P is a graph pattern to be matched against the knowledge base (for example, a PRDF graph), and $\vec{X}$ is a vector of variables, which is a subset of that appearing in P.

Example 4.4.2 *Consider the following two PRDF queries:*

```
Q1(?X,?Z) :- { (?X train+ ?Z) }
Q2(?X,?Z) :- { (?Y ?T+ ?Z), (?X (train+ | train.bus*) ?Z),
               (?Y train ?Z) }
```

For example, on the RDF graph G of Figure 4.3, the set of answers to the two queries are as follows:

$\mathcal{S}$(Q1,G) = { ⟨Metz,Strasbourg⟩,⟨Francfort,Mannheim⟩ }
$\mathcal{S}$(Q2,G) = { ⟨Metz,Strasbourg⟩,⟨Francfort,Strasbourg⟩
⟨Francfort,Mannheim⟩ }

4.4.2 Containment and canonical models

We show first that the entailment between two PRDF queries is not sufficient to guarantee the containment between them. More precisely, given two PRDF queries Q_1 and Q_2 such that $Q_1 \models_{\text{PRDF}} Q_2$, then $Q_1 \sqsubseteq Q_2$ does not necessarily hold. Consider the following two PRDF queries:

```
Q1 :- { (?X bus ?Y) }
Q2 :- { (?X bus+ ?Y)}
```

Q_1 searches the set of pairs of cities connected by a direct bus while Q_2 searches all pairs of cities connected by a sequence of buses. In terms of models, these queries are equivalent. In other words, all models of Q_1 are also models of Q_2, *i.e.,* $Q_1 \models_{\text{PRDF}} Q_2$ and $Q_2 \models_{\text{PRDF}} Q_1$ hold since the PRDF[$\mathcal{RE}$] graphs $\{\langle$`?X bus ?Y`$\rangle\}$ and $\{\langle$`?X bus+ ?Y`$\rangle\}$ have the same models. However, in terms of answers, we have $Q_1 \sqsubseteq Q_2$, but $Q_2 \not\sqsubseteq Q_1$ since (`Mannheim`, `Strasbourg`) is an answer of Q_2 in the RDF graph of Figure 4.3 but is not an answer of Q_1. This example shows that there must exist extra semantic conditions to guarantee the containment.

Theorem 4.4.3 (Containment and Entailment) *Let Q and Q' be two PRDF queries such that $Q \sqsubseteq Q'$. Then it may exist a PRDF query Q'' such that $Q'' \models_{PRDF} Q$, $Q \models_{PRDF} Q''$, and $Q'' \not\sqsubseteq Q'$.*

It is enough to give such a counter example as a proof of this theorem. Consider the Q_1 and Q_2 of the previous example, and the following PRDF query:

```
Q3 :- { (?X (bus | (bus.bus)) ?Y) }
```

searching the set of pairs of cities connected by exactly one or two trains. It is clear that $Q_1 \sqsubseteq Q_3$, Q_1 and Q_2 are semantically equivalent (*i.e.,* $Q_1 \models_{\text{PRDF}} Q_2$ and $Q_2 \models_{\text{PRDF}} Q_1$), but $Q_2 \not\sqsubseteq Q_3$. Nonetheless, if we consider only canonical models, then we have $Q_1 \models^c_{\text{PRDF}} Q_2$, but not the vice-versa.

Theorem 4.4.4 (Containment and Canonical Models) *Let Q and Q' be two queries. Then $Q \not\sqsubseteq Q'$, if and only if there exists an interpretation $I = \langle IR, IP, I_{EXT}, \iota\rangle$ and an extension ι' of ι to $\mathcal{B}(Q)$ such that (i) I is an ι'-canonical model of Q, (ii) does not exist an extension ι'' of ι to $\mathcal{B}(Q')$ such that ι'' is an ι'-model of Q'.*

Proof. For the if-part, it is sufficient to give a counterexample (see below). For the only-if-part, we have for each ι'-canonical model of Q, there exists always an extension ι'' such that ι'' is an ι'-model of Q'. This means that any canonical knowledge base G obtained by constructing a given ι'-canonical model of Q, there exists a map (*i.e.,* a PRDF homomorphism) from Q' into G. There exists therefore an answer to Q' in G. Hence, any answer of Q is also an answer of Q'. If this is not the case, we consider it as a counterexample and $Q \not\sqsubseteq Q'$.

Example 4.4.5 *Consider for example the interpretation $I = (I_R, I_P, I_{EXT}, \iota)$ defined by:*
- $I_R = \{Paris, Lyon, Grenoble, bus\}$;
- $I_P = \{bus\}$;
- $\iota(\texttt{bus}) = bus$, $I_{EXT}(bus) = \{(Grenoble, Lyon), (Lyon, Paris)\}$.

The existence of an extension ι' of ι defined by ($\iota'(\texttt{?X}) = Grenoble$ and $\iota'(\texttt{?Y}) = Paris$) such that I is an ι'-canonical model of Q_2, and there is no such an extension ι'' of ι with ι'' is an ι'-model of Q_1 shows that $Q_2 \not\models^c_{PRDF} Q_1$ and thus $Q_2 \not\sqsubseteq Q_1$.

Since we can associate to every canonical model a canonical GRDF knowledge base using Proposition 2.2.5, we can use the framework of [Florescu *et al.*, 1998] (see also [Calvanese *et al.*, 2000a]) for testing the containment of PRDF[$\mathcal{RE}$] graphs with simple semantics. In this framework, we find an EXPSPACE-complete algorithm based on canonical graphs (for us, canonical GRDF graphs) for testing the containment of conjunctive regular path queries (respectively, conjunctive regular path queries with inverse).

If we consider the RDF(S) vocabulary (see Chapter 8), then canonical models and canonical graphs must satisfy the RDF(S) conditions. In the same way, we can define the canonical entailment and containment using, for example, RDF(S) canonical models and RDF(S) canonical graphs.

Example 4.4.6 *Given the following two PRDF [$\mathcal{RE}$] queries:*

```
Q1 :- { (train subPropertyOf Transport),
        (Paris train+ ?Y)
      }
Q2 :- { (Paris transport+ ?Y)}
```

with simple semantics, we have $Q_1 \not\sqsubseteq Q_2$. However, if we consider RDF(S) semantics, then we have $Q_1 \sqsubseteq Q_2$.

4.4.3 Query containment for restricted PRDF queries

We study in this section a particular case of PRDF queries, *i.e.,* queries with *restricted* PRDF[$\mathcal{R}$] graphs, where a PRDF[$\mathcal{R}$] graph is restricted if each of its predicates represents a finite word. Then we show that the query containment in this case is NP-complete by reducing the problem to the containment of GRDF graphs. Let us first define formally *restricted* PRDF[$\mathcal{R}$] graphs.

Definition 4.4.7 (Restricted PRDF graph) *Let G be a PRDF$[\mathcal{R}]$ graph. We say that G is* restricted *if for each $\langle s, R, o\rangle \in G$, $R \in (\mathcal{U} \cup \mathcal{B})^k$, where $k \in \mathbb{N}\backslash\{0\}$.*

Example 4.4.8 *The following PRDF query is restricted since its body is a restricted PRDF $[\mathcal{RE}]$ graph, i.e., each predicate represents a word of length 2:*

```
Q1 :- { (Paris train.plane ?Y),
        (?Y plane.train Paris)
      }
```

Each restricted PRDF$[\mathcal{R}]$ graph can be normalized, and the result of the process will be a semantically equivalent GRDF graph.

Definition 4.4.9 (Normal graph) *Let G be a restricted PRDF$[\mathcal{R}]$ graph. Then the normal graph of G, denoted by normal(G), is the graph obtained by replacing each triple $\langle s, R, o\rangle \in G$, by $\langle s, a_1, x_1\rangle, \ldots, \langle x_{n-1}, a_k, o\rangle$ where $R = a_1 \cdot \ldots \cdot a_k$, and $x_{i's}$ are all new distinct variables.*

Example 4.4.10 *The PRDF $[\mathcal{RE}]$ query of Example 4.4.8 can be normalized to the following one:*

```
Q1 :- { (Paris train ?newVar1),
        (?newVar1 plane ?Y),
        (?Y plane ?newVar2),
        (?newVar2 train Paris)
      }
```

Theorem 4.4.11 *Let G and H be two restricted PRDF$[\mathcal{R}]$ queries. $G \sqsubseteq H$ iff normal$(G) \sqsubseteq$ normal(H).*

Proof. To prove this theorem, we show that for every restricted PRDF$[\mathcal{R}]$ graph G and normal(G) are canonically equivalent (*i.e.,* all canonical models of G are also canonical models of normal(G)). Let us consider a canonical model $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ of G, and show that I is also a canonical model of normal(G). Since I is a canonical model of G, there exists an extension ι' of ι to $\mathcal{B}(G)$ such that for every triple $\langle s, R = a_i \cdot \ldots \cdot a_k, o\rangle$, $\langle \iota'(s), \iota'(o)\rangle$ supports $a_i \cdot \ldots \cdot a_k$ in I according to ι'. That is, there exists a sequence $(r_0 = \iota'(s), \ldots, r_k = \iota'(o))$ of resources of I_R such that $\langle r_{i-1}, r_i\rangle \in I_{EXT}(\iota'(a_i))$, $\forall 1 \leq i, j \leq k$. normal$(G)$ contains, for every triple $\langle s, R, o\rangle \in G$, the following triples $\langle s, a_1, x_1\rangle, \ldots, \langle x_{n-1}, a_k, o\rangle$,

where $x_{i's}$ are all new distinct variables. Choose $x_{i's}$ to be interpreted by r_i in ι', $\forall 1 \leq i \leq k$. See that ι' is also a canonical model of normal(G).

On the other way, normal(G) contains, for every triple $\langle s, R, o\rangle \in G$, the following triples $\langle x_0 = s, a_1, x_1\rangle, \ldots, \langle x_{n-1}, a_k, x_k = o\rangle$, where $x_{i's}$ are all new distinct variables, $\forall 1 \leq i \leq k$. If I is a canonical model of normal(G), then there exists an extension ι' of ι to the variables normal(G) such that for every triple $\langle x_{i-1}, a_i, x_i\rangle$, $\langle \iota'(s), \iota'(o)\rangle \in I_{EXT}(\iota'(a_i))$. See that the sequence of resources $(\iota'(x_0) = \iota'(s), \iota'(x_1, \ldots, \iota'(x_{k-1}), \iota'(x_k) = \iota'(o))$ is a proof of $R = a_i \cdot \ldots \cdot a_k$ in I according to ι'. Hence, I is also a canonical model of G.

This result is not only applied to simple semantics but also to other semantics such that RDF(S) and OWL semantics. More precisely, using the process of normalizing restricted path queries, we can use the deductive algorithm for testing the containment of RDF(S) graph patterns of [Serfiotis *et al.*, 2005] including restricted path queries with RDFS semantics.

Example 4.4.12 *Given the following restricted PRDF* $[\mathcal{RE}]$ *queries with RDFS vocabulary:*

```
Q1 :- { (Paris transport.transport ?City) }

Q2 :- { (Paris train.plane ?City),
        (train subPropertyOf transport),
        (plane subPropertyOf transport),
      }
```

by applying the normalization process, we have:

```
Q1 :- { (Paris transport ?NewVar),
        (?NewVar transport ?City)
      }

Q2 :- { (Paris train ?NewVar),
        (?NewVar plane ?City),
        (train subPropertyOf transport),
        (plane subPropertyOf transport),
      }
```

by applying the deductive algorithm of [Serfiotis et al.*, 2005] to* Q_2*, we have:*

```
Q2 :- { (Paris train NewVar),
        (?NewVar plane ?City),
        (Paris transport NewVar),
        (?NewVar transport ?City),
        (train subPropertyOf transport),
        (plane subPropertyOf transport),
        ...
      }
```

Since there exists a containment mapping from Q_1 into Q_2 (according to [Serfiotis et al.*, 2005]), we have $Q_2 \sqsubseteq Q_1$.*

4.5 Conclusion

We have proposed in this chapter an extension of RDF, called PRDF[$\mathcal{R}$]. The language generators $\mathcal{R}$ in PRDF[$\mathcal{R}$] graphs are used to generate regular languages (*i.e.,* a set of words), and thus allow encoding variable length paths in graphs since each path labels form a word. The set of regular expressions has been used as a demonstration example to instantiate this extension, *i.e.,* PRDF[$\mathcal{RE}$]. The originality of our proposal lies in our adaptation of RDF model-theoretic semantics to take into account regular expression patterns, and the extension of the semantics to non-simple paths. This provides polynomial classes of the satisfiability problem of regular expressions, *e.g.* when they do not contain variables, and thus we solved the problem of simple paths proposed in (Example 4.1, [Mendelzon and Wood, 1995]).

In the following chapter, we will use PRDF[$\mathcal{RE}$] to generalize the SPARQL query language to have the PSPARQL extension. The inference mechanism, PRDF homomorphism, defined in this chapter will be exploited to construct answers to PSPARQL queries.

The PSPARQL Query Language 5

Contents

5.1 PSPARQL Syntax 66
5.1.1 PSPARQL graph patterns 66
5.1.2 PSPARQL query 67
5.2 Formal Semantics of PSPARQL 67
5.2.1 Answers to PSPARQL graph patterns 68
5.2.2 Answers to PSPARQL queries 69
5.3 Translation from PSPARQL to SPARQL 70
5.4 Algorithms for PSPARQL Query Evaluation 72
5.4.1 Triple-by-triple evaluation 72
5.4.2 A backtrack algorithm for calculating PRDF homomorphisms 77
5.5 Complexity of Evaluating PSPARQL Graph Patterns . . . 82
5.6 Conclusion 83

Introduction

As we mentioned before, SPARQL as an edge-based query language suffers from the ability of expressing paths. PSPARQL (stands for Path SPARQL) basically extends SPARQL with regular expression patterns (*i.e.,* using PRDF graphs as basic graph patterns) to overcome this limitation providing a wider range of querying paradigms [Alkhateeb *et al.*, 2008b].

We think that query languages for querying semantically defined languages like RDF should be defined semantically. This ensures the correct interpretation of the knowledge base to be queried, *e.g.* guaranteeing that querying two semantically

equivalent graphs will yield the same result. It also preserves the opportunity to extend this language beyond what can be defined through mappings, *e.g.* querying modulo an OWL ontology. Hence, we ground the definition of answers to a PSPARQL query by consequences (*i.e.,* PRDF-GRDF entailments). More precisely, we have proven in Chapter 4 that a GRDF graph G contains an answer to a PRDF graph H (G entails H) if and only if there exists a PRDF homomorphism (which is a particular map) from H into G. Then, PSPARQL query constructs are defined through algebraic operations on PRDF homomorphisms.

This chapter is dedicated to the presentation of PSPARQL. Section 5.1 presents its syntax, which is built on top of PRDF in the same way that SPARQL is built on top of RDF. Section 5.2 defines the answers to a given PSPARQL query following the framework of [Perez *et al.*, 2006] followed by the algorithms for calculating these answers in Section 5.4. Finally, Section 5.5 presents the complexity study of evaluating PSPARQL graph patterns.

5.1 PSPARQL Syntax

The only difference between the syntax of SPARQL and that of PSPARQL, is basic graph patterns. In SPARQL, they are GRDF graphs while in PSPARQL they are PRDF graphs instantiated to regular expression patterns. This means that PSPARQL keeps the compatibility with SPARQL queries since PRDF graph patterns reduced to atomic terms are GRDF graph patterns.

5.1.1 PSPARQL graph patterns

PSPARQL graph patterns are built on basic graph patterns which are PRDF[$\mathcal{RE}$] graphs, where $\mathcal{RE}$ denotes the set of regular expression patterns constructed over the set of urirefs and the set of variables ($\mathcal{U} \cup \mathcal{B}$).

Definition 5.1.1 (PSPARQL graph patterns) *A* PSPARQL graph pattern *is defined inductively in the following way:*

- *every PRDF[$\mathcal{RE}$] graph is a PSPARQL graph pattern;*
- *if P_1, P_2 are PSPARQL graph patterns and C is a SPARQL constraint, then (P_1 `AND` P_2), (P_1 `UNION` P_2), (P_1 `OPT` P_2), and (P_1 `FILTER` C) are PSPARQL graph patterns.*

Example 5.1.2 *The following PSPARQL graph pattern P*

```
{ { ex:Paris (ex:train|ex:plane)+ ?City . }
  {
    { ?City ex:capitalOf ?Country . }
      UNION
    { ?City ex:populationSize ?Population .
      FILTER (?Population > 200000)
    }
  }
}
```

consists of the following basic graph patterns (i.e., PRDF graphs) and constraint:
$P = (P_1$ `AND` $(P_2$ `UNION` $(P_3$ `FILTER` $C)))$, *where*
$P_1 = \{$ `ex:Paris (ex:train|ex:plane)+ ?City .` $\}$ *that finds cities reachable from Paris by a sequence of trains or planes;*
$P_2 = \{$ `?City ex:capitalOf ?Country .` $\}$ *that finds capital cities together with their countries;*
$P_3 = \{$ `?City ex:populationSize ?Population.` $\}$ *that finds cities and their population size;*
$C =$ `Filter (?Population > 20000)` *is a constraint that restricts the values of the variable* `?Population` *to be greater than* 200000.

As PSPARQL introduces PRDF[$\mathcal{RE}$] graphs, we give in Table 5.1 the necessary modifications to the SPARQL grammar [Prud'hommeaux and Seaborne, 2008] in the extended Backus-Naur form, where the production rule [21'] replaces [21] in SPARQL, and all other rules are added to SPARQL grammar to have a complete grammar for PSPARQL Appendix A (see also `psparql.inrialpes.fr`).

5.1.2 PSPARQL query

A *PSPARQL query* is of the form `SELECT` $\vec{B}$ `FROM` u `WHERE` P. The only difference with a SPARQL query is that, this time, P is a PSPARQL graph pattern, *i.e.,* a PRDF[$\mathcal{RE}$] graph. The use of variables in PRDF regular expression patterns is a generalization of the use of variables as predicates in the basic graph patterns of SPARQL.

5.2 Formal Semantics of PSPARQL

Answers to SPARQL queries are defined based on maps from GRDF graph patterns of the query into the RDF knowledge base following the framework outlined in

[21']	$\langle TriplesBlock \rangle$	::=	$\langle PathTriples1 \rangle$
		|	(‘.’ $\langle PathTriples1 \rangle$?)*
[30.1]	$\langle PathTriples1 \rangle$	::=	$\langle VarOrTerm \rangle$ $\langle PathPropLNE \rangle$
		|	$\langle PathTripleNode \rangle$ $\langle PathPropL \rangle$
[31.1]	$\langle PathPropL \rangle$	::=	$\langle PathPropLNE \rangle$?
[32.1]	$\langle PathPropLNE \rangle$	::=	$\langle PathVerb \rangle$ $\langle PathObL \rangle$ (‘;’ $\langle PathPropL \rangle$)?
[33.1]	$\langle PathObL \rangle$	::=	$\langle PathGraphNode \rangle$ (‘,’ $\langle PathObL \rangle$)?
[34.1]	$\langle PathVerb \rangle$	::=	$\langle RegularExp \rangle$
[35.1]	$\langle PathTripleNode \rangle$	::=	$\langle PathCollection \rangle$
		|	$\langle PathBNodePropL \rangle$
[36.1]	$\langle PathBNodePropL \rangle$	::=	‘[’ $\langle PathPropLNE \rangle$ ‘]’
[37.1]	$\langle PathCollection \rangle$	::=	‘(’ $\langle PathGraphNode \rangle$+ ‘)’
[38.1]	$\langle PathGraphNode \rangle$	::=	$\langle VarOrTerm \rangle$
		|	$\langle PathTripleNode \rangle$
[39.1]	$\langle RegularExp \rangle$	::=	$\langle Rexp \rangle$ ((‘|’ | ‘·’) $\langle Rexp \rangle$)*
[39.2]	$\langle Rexp \rangle$	::=	(‘+’ | ‘*’)? $\langle Atom \rangle$
[39.3]	$\langle Atom \rangle$	::=	‘!’ $\langle IRIref \rangle$
		|	$\langle VarOrIRIref \rangle$
		|	‘(’ $\langle RegularExp \rangle$ ‘)’

Table 5.1: PSPARQL graph pattern grammar.

[Perez *et al.*, 2006]. Since answers to PSPARQL queries are given using maps, the same framework can be used to define semantics of PSPARQL queries.

5.2.1 Answers to PSPARQL graph patterns

As in the case of SPARQL reduced to GRDF graphs, the answer to a query reduced to a PRDF[$\mathcal{RE}$] graph is also given by a map. The definition of an answer to a PSPARQL query will thus be identical to that given for SPARQL [Perez *et al.*, 2006], but it will use PRDF homomorphisms.

Definition 5.2.1 (Answer to PSPARQL graph patterns) *Let P be a PSPARQL graph pattern and G be an RDF graph. The set $\mathcal{S}(P, G)$ of answers of P in G is defined inductively in the following way:*

- *if P is a PRDF[$\mathcal{RE}$] graph, $\mathcal{S}(P, G) = \{\mu \mid \mu$ is a PRDF homomorphism from P into $G\}$;*
- *if $P = (P_1 \texttt{ AND } P_2)$, $\mathcal{S}(P, G) = \mathcal{S}(P_1, G) \bowtie \mathcal{S}(P_2, G)$;*
- *if $P = (P_1 \texttt{ UNION } P_2)$, $\mathcal{S}(P, G) = \mathcal{S}(P_1, G) \cup \mathcal{S}(P_2, G)$;*
- *if $P = (P_1 \texttt{ OPT } P_2)$, $\mathcal{S}(P, G) = (\mathcal{S}(P_1, G) \bowtie \mathcal{S}(P_2, G)) \cup (\mathcal{S}(P_1, G) \setminus \mathcal{S}(P_2, G))$;*

– *if* $P = (P_1 \texttt{ FILTER } C)$, $\mathcal{S}(P, G) = \{\mu \in \mathcal{S}(P_1, G) \mid \mu(C) = \top\}$.

Example 5.2.2 *According to Definition 5.2.1, the set of answers of the PSPARQL graph pattern P of Example 5.1.2 in a given RDF graph G is defined as:*

$P = (\mathcal{S}(P_1, G) \bowtie (\mathcal{S}(P_2, G) \cup (\{\mu \in \mathcal{S}(P_3, G) \mid \mu(C) = \top\})))$

In words, the set of maps (i.e., PRDF homomorphisms) from P_1 into G joined with the union of that from P_2 into G and those from P_3 into G that satisfy the constraint C.

5.2.2 Answers to PSPARQL queries

If $Q =$`SELECT` $\vec{B}$ `FROM` u `WHERE` P is a PSPARQL query, G is the GRDF graph identified by the URI u, and Ω is the set of answers of P in G, then the answers to Q are the projections of elements of Ω to $\vec{B}$, *i.e.,* for each map π of Ω, the answer to Q associated to π is $\{(x, y) \mid x \in \vec{B}$ and $y = \pi(x)$ if $\pi(x)$ is defined, `null` otherwise$\}$ otherwise.

Proposition 5.2.3 *Let $Q =$`SELECT` $\vec{B}$ `FROM` u `WHERE` P be a PSPARQL query, P be a PRDF$[\mathcal{RE}]$ graph and G be a GRDF graph identified by URI u, then the answers to Q are the images of variables in $\vec{B}$ by a PRDF homomorphism π from P into G such that $G \models_{PRDF} \pi(P)$.*

This property is a straightforward consequence of Definition 5.2.1 and Theorem 4.3.5. It is based on the fact that the answers to Q are the restrictions to $\vec{B}$ of the set of PRDF homomorphisms from P into G which, by Theorem 4.3.5, corresponds to PRDF-GRDF ENTAILMENT.

Example 5.2.4 *The following PSPARQL query that uses the graph pattern P of Example 5.1.2:*

```
SELECT ?City
WHERE { P }
ORDER BY Asc(?City)
```

returns in an ascending order the set of cities reachable from Paris by a sequence of trains and planes, which are either capital cities or have a population size greater than 200000.

5.3 Translation from PSPARQL to SPARQL

Simple PSPARQL queries (*i.e.,* with absence of the recursion operators + and * in regular expressions), could be expressed by equivalent SPARQL queries.

Example 5.3.1 *The following PSPARQL query:*

```
SELECT ?City
WHERE { ex:Paris (ex:train | ex:plane).ex:bus ?City .}
```

that searches cities connected to Paris by plane or train relations followed by a bus relation, is equivalent to the following SPARQL query:

```
SELECT ?City
WHERE {
        { {ex:Paris ex:train ?MidCity .}
           UNION
          {ex:Paris ex:plane ?MidCity .}
        }
        ?MidCity ex:bus ?City .
      }
```

Nonetheless, as shown in this example, regular expressions provide a more compact syntax. Moreover, the complexity is growing very rapidly with the size of queries while regular expressions suggest a natural and more efficient evaluation. In addition, we should pay attention when we translate from PSPARQL to SPARQL queries, in particular, for queries involving negation in regular expressions. Let us illustrate this point given the following RDF graph.

```
{ (ex:Person1 foaf:name "Faisal Alkhateeb"),
  (ex:Person1 foaf:knows "Jérôme Euzenat"),
  (ex:Person1 foaf:knows "Jean François Baget")
}
```

Suppose we want to find persons who do not know "Jérôme Euzenat". Then the following SPARQL query:

```
SELECT ?Name
WHERE {
        ?Person1 foaf:name ?Name .
        ?Person1 foaf:knows ?Person2 .
        FILTER ( ?Person2 != "Jerome Euzenat") .
      }
```

as it returns also the person named "Faisal Alkahteeb" who knows "Jean François Baget", fails to achieve the desired answers.

One solution to do that is using a trick reproducing negation as failure from Logic programming [Clark, 1978]. That is, by testing if a graph pattern is not expressed by specifying an OPTIONAL graph pattern that introduces a variable and testing to see that the variable is not bound.

```
SELECT ?Name
WHERE {
        ?Person1 foaf:name ?Name .
        OPTIONAL { ?Person1 foaf:knows ?Person2 .
                   FILTER ( ?Person2 = "Jerome Euzenat") .
                 }
        FILTER ( !BOUND ( ?Perosn2) ) .
      }
```

The same problem occurs when using variables in the predicate position. This way, the following SPARQL query:

```
SELECT ?City2
WHERE {
        ?City1 foaf:name "Paris" .
        ?City1 ?Mean ?City2 .
        FILTER ( ?Mean != ex:plane)
      }
```

fails to find cities that are not connected to Paris by a plane. Also, according to the semantics of regular expressions (Chapter 4), the following two equivalent PSPARQL queries:

```
SELECT ?City1
WHERE {
        ?City1 foaf:name "Paris" .
        ?City1 (!ex:plane)+ ?City2 .
      }
```

and

```
SELECT ?City1
WHERE {
        ?City1 foaf:name "Paris" .
        ?City1 ?Mean+ ?City2 .
        FILTER ( ?Mean != ex:plane )
      }
```

search cities connected by a transportation mean other than plane, which is the usual semantics of regular expressions. However, the following query:

```
SELECT ?City2
WHERE { ?City1 foaf:name "Paris" .
        OPTIONAL { ?City1 ?Mean+ ?City2 .
                   FILTER ( ?Mean = ex:plane ) . }
        FILTER ( !BOUND(?Mean) ) . }
```

finds cities that are not connected to Paris by a sequence of planes.

As a consequence, the negation operator in SPARQL does not always guarantee the correct interpretation of the negation operator in regular expressions. Hence, PSPARQL is more expressive than SPARQL because the use of recursion operators and that even if we can translate the rest easily, this translation does not interact well with the negation operator.

5.4 Algorithms for PSPARQL Query Evaluation

To answer a PSPARQL query Q involving PRDF[$\mathcal{RE}$] graphs as basic graph patterns, mandates to enumerate all PRDF[$\mathcal{RE}$] homomorphisms from the graph pattern(s) of Q into the data RDF graph of Q. So, we are interested in an algorithm that, given a PRDF[$\mathcal{RE}$] graph H and an RDF graph G, solves the following problems:

1. Is there a PRDF[$\mathcal{RE}$] homomorphism from H into G?

2. Exhibit, if it exists, a PRDF[$\mathcal{RE}$] homomorphism from H into G.

3. Enumerate all PRDF[$\mathcal{RE}$] homomorphisms from H into G.

Two possible methods can be used for solving these problems: a method based on evaluating the PRDF graph triple-by-triple is presented in Section 5.4.1; and a backtracking method based on the standard backtrack techniques is presented in Section 5.4.2.

5.4.1 Triple-by-triple evaluation

Given a PRDF[$\mathcal{RE}$] graph H and an RDF graph G, we can enumerate all PRDF homomorphisms from H into G by evaluating the graph H triple-by-triple and take the join of the intermediate results. This method is similar to the edge-by-edge evaluation method presented in [Cruz *et al.*, 1988].

Evaluation algorithms

[Liu *et al.*, 2004; de Moor and David, 2003] present the algorithm $Reach(G, R, v_0)$ (Algorithm 1), where G is a graph (for us, an RDF graph), R is a regular expression pattern and v_0 is a node of G. This algorithm calculates the set of triples $\langle v_0, v_k, \mu\rangle$, where v_k is a node of G and μ is a map from terms of R into terms of G such that there exists a sequence $T = (v_0, \ldots, v_k)$ of nodes of G and a word $w \in L^*(R)$ with T is a path of w in G according to μ.

The $Reach$ algorithm uses a non deterministic finite automaton (NDFA) that recognizes a language equivalent to a given regular expression pattern. It can be constructed in the usual way (*cf.* [Aho *et al.*, 1974]). It also reuses the definition of matching two regular expression patterns found in [Liu *et al.*, 2004].

Matching regular expression patterns. Let R_1 and R_2 be two regular expression patterns, then we say that R_2 *matches* R_1 under the mapping μ, denoted by $match(R_2, R_1, \mu)$, if one of the following conditions holds:

1. $R_1 = \mu(R_2)$;
2. $R_2 \in \mathcal{B}$ and $R_2 \notin dom(\mu)$;
3. $R_1, R_2 \in \mathcal{B}$ and ($\mu(R_2) = R_1$ or $R_2 \notin dom(\mu)$);
4. $R_2 = \#$;
5. $R_2 =!R_3$, and recursively, R_1 does not match R_3;
6. $R_1 = \langle e_1, \ldots, e_k\rangle$, $R_2 = \langle a_1, \ldots, a_k\rangle$, and recursively e_i matches a_i, $\forall 1 \leq i \leq k$, where e_i, a_i are the atomic elements of R_1, R_2.

For example, the regular expression pattern $(\texttt{?z} \cdot \texttt{?y})$ matches the regular expression pattern $(\texttt{ex:train} \cdot \texttt{ex:plane})$ and the result will be the mapping $\{\langle \texttt{?z}, \texttt{ex:train}\rangle, \langle \texttt{?y}, \texttt{ex:plane}\rangle\}$.

The $Reach$ algorithm is used by the algorithm $Evaluate$ (Algorithm 2), which, given an RDF graph G and a PRDF[$\mathcal{RE}$] triple $\langle x, R, y\rangle$, calculates the set of maps μ such that $\langle \mu(x), \mu(y)\rangle$ satisfies R in G with the map μ (it is said that μ satisfies $\langle x, R, y\rangle$ in G).

The results of the $Evaluate$ algorithm are used to calculate the PRDF homomorphisms of a PRDF[$\mathcal{RE}$] graph P into an RDF graph G by successive joins in the algorithm $Eval$ (Algorithm 3), whose initial call will be $Eval(P, G, \{\mu_\emptyset\})$, where $\mu_\emptyset$ is the map with the empty domain.

The $Eval$ algorithm is given for evaluating PRDF[$\mathcal{RE}$] graphs, and can be extended to evaluate PSPARQL graph patterns following the $Eval$ algorithm for evaluating SPARQL graph patterns [Perez *et al.*, 2006].

Algorithm 1: $Reach(G, R, v_0)$

Data: An RDF graph G, a regular expression R and a start node v_0 in G.
Result: $\{\langle v_0, v_k, \mu\rangle \mid$ there exists a sequence $T = (v_0, \ldots, v_k)$ of nodes of G, a map μ from terms of R into $term(G)$ and a word $w \in L^*(R)$ with T is a path of w in G according to μ.

```
begin
    Let A = ⟨S, s_0, δ, F⟩ be the NDFA of R;
    R ← {};
    W ← {};
    S(G) ← {};
    for ⟨s_0, tl, s⟩ ∈ A do
        for ⟨v_0, el, v⟩ ∈ G do
            if match(tl, el, μ_∅) then
                μ ← {⟨tl, el⟩};
                μ' = (μ ⊕ μ_i);
                W ← W ∪ {⟨v, s, μ'⟩};
    while (exists ⟨v, s, μ⟩ ∈ W) do
        R ← R ∪ {⟨v, s, μ⟩}; W ← W − {⟨v, s, μ⟩};
        for ⟨s, tl, s_1⟩ ∈ A do
            for ⟨v, el, v_1⟩ ∈ G do
                if match(tl, el, μ) then
                    μ_1 ← {⟨tl, el⟩}; μ_2 ← (μ ⊕ μ_1);
                    if (⟨v_1, s_1, μ_2⟩ ∉ R) then
                        W ← W ∪ {⟨v_1, s_1, μ_2⟩};
        if s ∈ F then
            S(G) ← S(G) ∪ {⟨v_0, v, μ⟩};
    return S(G);
end
```

Algorithm 2: $Evaluate(t, G)$.

Data: An RDF graph G, a PRDF[$\mathcal{RE}$] triple $t = (x, R, y)$.
Result: The set of maps μ satisfying t in G.
begin
  if $x \in \mathcal{U}$ **then**
    $S_G(t) \leftarrow Reach(G, R, x)$;
  else
    $S_G(t) \leftarrow \bigcup_{s \in G}\{Reach(G, \mu_p(R), s,) \mid \mu_p \leftarrow \{\langle x, s\rangle\}\}$;
  if $y \in \mathcal{V}$ **then**
    $S_G(t) \leftarrow \{(s, y, \mu) \in S_G(t)\}$
  else
    $S_G(t) \leftarrow \{(s, o, \mu') \mid (s, o, \mu) \in S_G(t)$, $(\mu, (y \leftarrow o))$ are compatible, and $\mu' \leftarrow \mu \oplus \{(y \leftarrow o)\}\}$
  return $\{\mu \mid (s, o, \mu) \in S_G(t)\}$;
end

Algorithm 3: $Eval(P, G, \Omega)$.

Data: An RDF graph G, a set of maps, a PRDF graph P.
Result: The set $\{\mu \mid \mu$ is a PRDF homomorphism from P into $G\}$.
begin
  if $P = \{t\}$ **then**
    return $\Omega \bowtie Evaluate(t, G)$;
  else
    if $P = (t \cup P')$ **then**
      return $Eval(\{t\}, G, Eval(P', G, \Omega))$;
end

Algorithmic time complexity.

The $Reach$ algorithm has worst-case time complexity $\mathcal{O}(|G| \times |R_i| \times maps \times (predicateSize + vars(R_i)))$ (the notations used in Table 5.2 are reformulated from [Liu *et al.*, 2004] and adapted to our problem). Now, for each triple $\langle x, R_i, y\rangle$ in P, the $Reach$ algorithm is called by the $Evaluate$ algorithm once if x is a constant, *i.e.,* a uriref or a literal if it is allowed in the subject position; otherwise it is called for each node in G multiplied by the number of variables in P in the subject position. So, the $Evaluate$ algorithm has overall worst-case time complexity $\mathcal{O}((vars_s(P) \times subj(G) + const_s(P)) \times |G| \times |R_i| \times maps \times (predicateSize + vars(R_i)))$, where $vars_s(P)$ (respectively, $const_s(P)$) is the number of variables (respectively, constants) appearing in the subject position in a triple of P.

Name	Meaning
$vars$	the number of variables.
$predicateSize$	the maximum predicate size appearing in G or in R.
$maps$	the number of possible maps from variables of R into terms of G that match some path in G with some path in R; the worst case is $pred(G)^{vars(R)}$.

Table 5.2: Notations for complexity analysis

This result shows an exponential complexity with respect to the number of variables in the regular expression patterns of the PRDF graph representing the query ($\mathcal{O}(pred(G)^{vars(R)})$). However, the size of the query, and in particular, the number of variables is usually considered very small with regards to the knowledge base. Hence, the number of variables in each regular expression pattern can be assumed a constant. With this assumption, the data complexity, which is defined as the complexity of query evaluation for a fixed query [Vardi, 1982], is $\mathcal{O}(|G|^2)$, *i.e.,* not much worse than the one of SPARQL [Perez *et al.*, 2006].

Though the above method is correct and complete, it is not efficient, in particular, for testing the existence of a PRDF homomorphism which is sufficient for checking if a PRDF[$\mathcal{RE}$] graph is a consequence of an RDF graph. Using this method, we need to perform the join operation for all PRDF triples to have the set of maps, *i.e.,* the set of PRDF homomorphism, while we need to test the existence of one PRDF homomorphism. Consider the PRDF graph P and the RDF graph of Figure 5.1. To test if there exists a PRDF homomorphism from P into G, we need to solve PATH SATISFIABILITY N^2 times for the regular expression pattern

R in P, where N is the number of nodes of G. However, we need to solve PATH SATISFIABILITY only once as it appears in Figure 5.1. More precisely, since the extremities of the regular expression R are variables (namely, `?b6` and `?b7`), we need to check for each pair of nodes $\langle x, y\rangle$ of G if they satisfy R in G while, in this example, `?b6` and `?b7` can be only mapped to `ex:c1` and `ex:c2`, respectively. In such a case, it is sufficient to determine whether the pair $\langle$`ex:c1`, `ex:c2`$\rangle$ satisfies R in G.

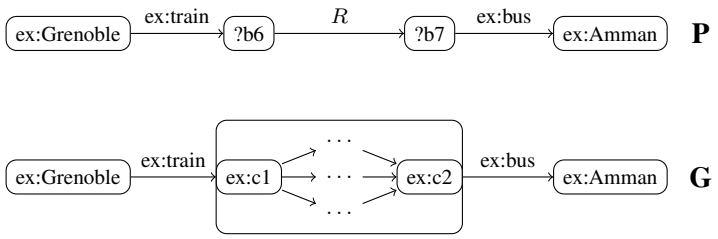

Figure 5.1: A case in which the path closure method is not efficient.

The next section presents a backtracking algorithm for calculating the set of PRDF homomorphisms of a PRDF graph into an RDF graph. This algorithm has the same worst-case time as the triple-by-triple method, but it is more efficient in practice since in some cases there is no need to traverse all the backtrack tree to find the first PRDF homomorphism.

5.4.2 A backtrack algorithm for calculating PRDF homomorphisms

An alternative method for evaluating PSPARQL graph patterns, *i.e.,* enumerating all PRDF homomorphisms from the PRDF graph of a given PSPARQL query into the data graph, is based on a backtracking technique that generates each possible map from the current one by traversing the parse tree in a depth-first manner and using the intermediate results to avoid unnecessary computations.

Algorithm 4 is a simple recursive version of the basic Backtrack algorithm [Golomb and Baumert, 1965]. The inputs to this algorithm are: a PRDF graph, an RDF graph, and a partial map, denoted by $partialProj$. $partialProj$ includes a set of pairs $\{\langle x_i, y_i\rangle\}$ such that x_i is a term of H (*i.e.,* $x_i \in term(H)$), and y_i is the image of x_i in G (*i.e.,* $y_i \in term(G)$).

The other parts of the algorithm perform as follows:

$chooseTerm(nodes(H))$ chooses a term $x \in nodes(H)$.

Algorithm 4: $Extendhomomorphism(H, G, partialProj, n)$.

Data: A PRDF graph H, an RDF graph G, and a partial map $partialProj$ from $term(H)$ to $term(G)$.
Result: Extends the partial map to PRDF homomorphisms.
if n==*nodes(H)* **then**
  return $solution\text{-}Found(partialProj)$;
$x \leftarrow chooseTerm(nodes(H))$;
for *each* $\langle y, \theta \rangle \in candidates(partialProj, x, G, H)$ **do**
  $Extendhomomorphism(H, G, partialProj \oplus \{\langle x, y \rangle\} \oplus \theta, n + 1)$;

$candidates(partialProj, x, G, H)$ calculates all possible candidate images in G for the current term x satisfying the partial map $partialProj$. It returns all sets of pairs $\langle y, \theta \rangle$ such that y is a possible image of x, and θ is the possible map from the terms of each regular expression pattern R_i appearing in a triple with x and one of the terms in $nodes(H)$ already mapped in $partialProj$. That is, if there is no term of $nodes(H)$ having a triple with x, then the possible candidate images of x are all y in $nodes(G)$ such that x can be mapped to y (*cf.* the definition of mapping Definition 2.3.1). Otherwise, there exists a set of terms $z_1, \ldots, z_k \in nodes(H)$ having a triple with x, which are already mapped in $partialProj$. In this case, $image(z_i)$ and y satisfies $\theta(R_i)$, where R_i is the regular expression pattern appearing in the predicate position of the triple between z_i and x. The order in which the two nodes $image(z_i)$ and y satisfies $\theta(R_i)$ depends on the order in which x and z_i appear in the triple, that is, if the triple is $\langle z_i, R_i, x \rangle$ then $\langle image(z_i), y \rangle$ satisfies $\theta(R_i)$ in G, otherwise $\langle y, image(z_i) \rangle$ satisfies $\theta(R_i)$ in G. θ maps the terms appearing in the regular expression patterns of H into the terms appearing along the paths in G with respect to $partialProj$, that is, θ is a possible map such that θ and $partialProj$ are compatible.

Then the algorithm takes each candidate y of the current term $x \in nodes(H)$ and the possible map θ, put y in the $image(x)$, and tries to generate the possible candidates of y with the current map $partialProj \oplus \{\langle x, y \rangle\} \oplus \theta$ (note that $partialProj$, $\{\langle x, y \rangle\}$ and θ are compatible, since the set $\langle y, \theta \rangle$ is calculated with respect to $partialProj$). This is done recursively in a depth-first manner in the call of $Extendhomomorphism(H, G, partialProj \oplus \{\langle x, y \rangle\} \oplus \theta)$. At the end of the algorithm, we have a tree that contains one level with a term from H, *i.e.,* a node from H, and one level with the possible images of that term in G. The input to

Algorithm 5: $Candidates(\mu_p, x, G, H)$.

Data: A map μ_p, an RDF graph G and a node x from a PRDF graph H.
Result: The set $\langle y, \mu \rangle$ such that y is a possible image of x in G, and μ extends μ_p to the node x.

```
begin
    preV_s ← {⟨x, R_i, z_i⟩ | ⟨x, R_i, z_i⟩ ∈ H and z_i ∈ dom(μ_p)};
    preV_o ← {⟨z_i, R_i, x⟩ | ⟨z_i, R_i, x⟩ ∈ H and z_i ∈ dom(μ_p)};
    if preV_s == ∅ and preV_o == ∅ then
        if x is variable then
            if x ∉ dom(μ_p) then
                candidates = {⟨y, μ_p⟩ | y ∈ nodes(G)};
            else
                if μ_p(x) ∈ nodes(G) then
                    candidates = ⟨μ_p(x), μ_p⟩;
                else
                    candidates = ∅;
        else
            if x ∈ nodes(G) then
                candidates = ⟨x, μ_p⟩;
            else
                candidates = ∅;
    else
        if preV_s ≠ ∅ then
            tempCands = {⟨s, μ⟩ | ⟨x, R_i, z_i⟩ ∈ preV_s and
            ⟨s, o, μ⟩ ∈ sat(μ_p, x, R_i, μ_p(z_i), G)};
            preV_s = preV_s − {⟨x, R_i, z_i⟩};
        else
            tempCands = {⟨o, μ⟩ | ⟨z_i, R_i, x⟩ ∈ preV_o and
            ⟨s, o, μ⟩ ∈ sat(μ_p, μ_p(z_i), R_i, x, G)};
            preV_o = preV_o − {⟨z_i, R_i, x⟩};
        for each ⟨x, R_i, z_i⟩ ∈ preV_s do
            candidates = {⟨s, μ'⟩ | ⟨s, μ_1⟩ ∈ tempCands,
            ⟨s, o, μ_2⟩ ∈ sat(μ_p, x, R_i, μ_p(z_i), G), μ_1, μ_2 are compatible,
            and μ' ← merge(μ_1, μ_2)};
            tempCand = candidates;
        for each ⟨z_i, R_i, x⟩ ∈ preV_o do
            candidates = {⟨o, μ'⟩ | ⟨o, μ_1⟩ ∈ tempCands,
            ⟨s, o, μ_2⟩ ∈ sat(μ_p, μ_p(z_i), R_i, x, G), μ_1, μ_2 are compatible,
            and μ' ← merge(μ_1, μ_2)};
            tempCand = candidates;
    return candidates;
end
```

Algorithm 6: $sat(\mu_p, x, R, y, G)$.

Data: An RDF graph G, a PRDF[$\mathcal{RE}$] triple (x, R, y), and a partial map μ_p.
Result: The set of triples $\langle s, o, \mu \rangle$ such that the map μ satisfies $(x, \mu_p(R), y)$ in G.

```
begin
    S ← {};
    if (x ∈ U) or (x ∈ dom(μp)) then
        if x ∈ dom(μp) then
            n ← μp(x);
        else
            n ← x;
        S ← S ∪ Reach(G, μp(R), n);
    else
        S ← ⋃_{s∈G} {Reach(G, μ'p(R), s) | μ'p ← {⟨x, s⟩} ⊕ μp)};
    if y ∈ U then
        S ← {(s, y, μ) ∈ S}
    else
        S ← {(s, o, μ') | (s, o, μ) ∈ S, (μ, (y ← o)) are compatible, and
        μ' ← μ ⊕ {(y ← o)}}
    return S;
end
```

each node of each level is the current map. Each possible path in the tree from the root to a leaf labeled by a term of G represents a possible PRDF homomorphism.

If we call $Extendhomomorphism(H, G, partialProj_{\emptyset}, n = 0)$ with the empty map $partialProj_{\emptyset}$, then at the end of the algorithm we have all PRDF homomorphisms from the PRDF graph H into the RDF graph G.

Example 5.4.1 *Let the PRDF graph H and the RDF graph G of Figure 5.2 represent a graph pattern of a PSPARQL query a data graph, respectively (we use $\overset{p}{\longleftrightarrow}$ to represent an incoming and outcoming arcs labeled with p). To enumerate the set of PRDF homomorphisms from H into G, the algorithm chooses an arbitrary term from H (assume it is* `ex:Lyon`*). Then it searches the RDF graph G to find all possible candidate images for* `ex:Lyon`*, which will be, if it presents in G, the term* `ex:Lyon`*. It found such a term, so the only candidates for* `ex:Lyon` *is* `ex:Lyon`*. Now, it chooses another term of H (suppose it is* `?W`*). Then, the algorithm calls $candidates(\{\langle$`ex:Lyon`$,$`ex:Lyon`$\rangle\}$, `?W`$, G)$. Since there exists only one triple in H containing* `?W` *and one of the terms already mapped by $partialProj$, i.e.,* `ex:Lyon`*, the possible candidate images for* `?W` *are all $\langle y, \theta \rangle$ such that the pair $\langle$*`ex:Lyon`*$,y\rangle$ satisfies the regular expression $\theta($*`?X`$^+)$*, which will be:*

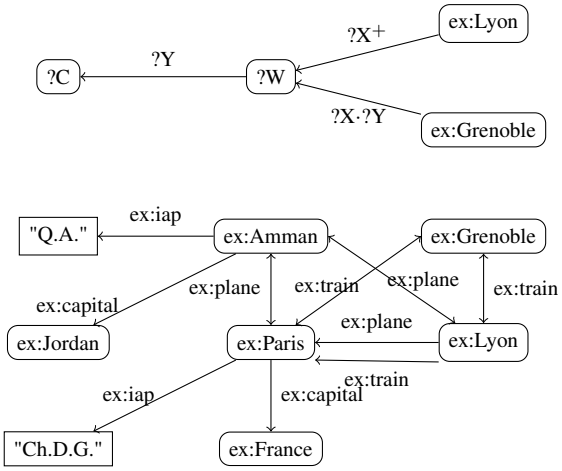

Figure 5.2: A PRDF graph H and an RDF graph G.

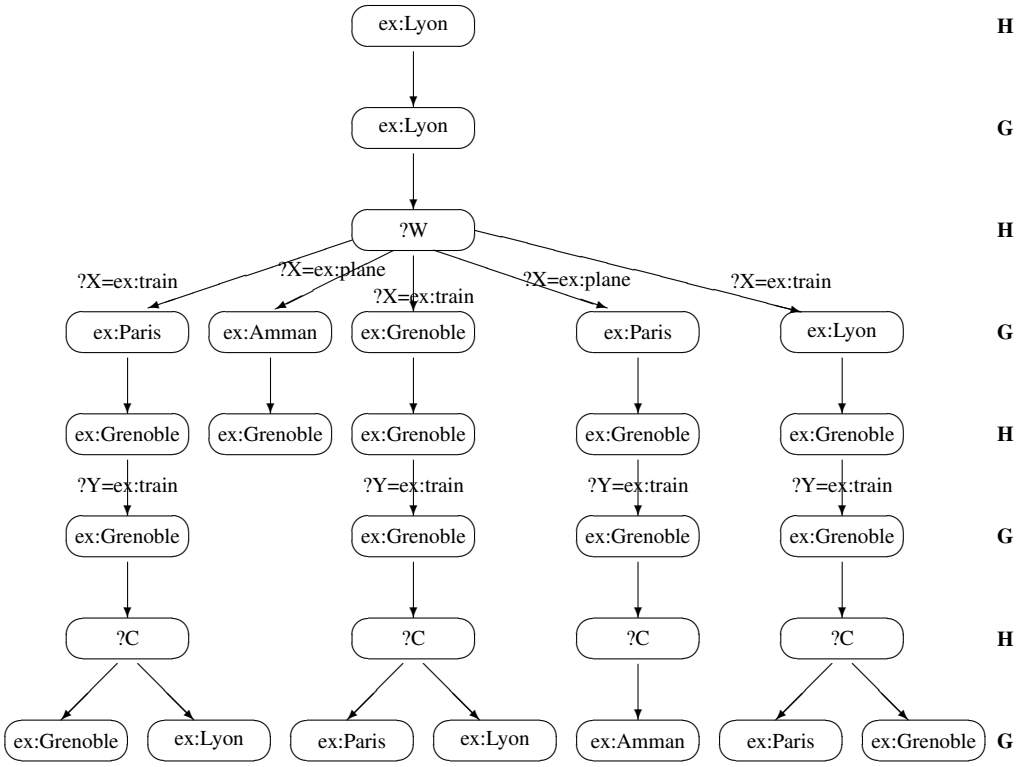

Figure 5.3: The backtracking result of Example 5.4.1.

$$
\begin{array}{lll}
\{ & \langle\ \texttt{ex:Paris}, & \theta = \{(\texttt{?X},\texttt{ex:train})\}\ \rangle, \\
 & \langle\ \texttt{ex:Paris}, & \theta = \{(\texttt{?X},\texttt{ex:plane})\}\ \rangle, \\
 & \langle\ \texttt{ex:Amman}, & \theta = \{(\texttt{?X},\texttt{ex:plane})\}\ \rangle, \\
 & \langle\ \texttt{ex:Grenoble}, & \theta = \{(\texttt{?X},\texttt{ex:train})\}\ \rangle, \\
 & \langle\ \texttt{ex:Lyon}, & \theta = \{(\texttt{?X},\texttt{ex:train})\}\ \rangle \\
\} & &
\end{array}
$$

Then the algorithm takes the possible candidates of `?W` *one by one, and try to extend* $partialProj$*. For the first candidate of* `?W`*, it calls* $Extendhomomorphism(H, G, \{\langle \texttt{ex:Lyon, ex:Lyon}\rangle, \langle \texttt{?W, ex:Paris}\rangle, \langle \texttt{?X, ex:train}\rangle\})$*. Then, it chooses another term from* $nodes(H)$ *not yet being mapped to a term of* G*, and repeats the same steps for this term. The backtrack tree of this example is given in Figure 5.3.*

One interesting feature of the backtrack algorithm is that we can stop the search process after a determined number of solutions. In the case of answering boolean queries (*e.g.* SPAQRL ASK queries) or checking the consequences, we need only to verify if there exists at least one solution.

5.5 Complexity of Evaluating PSPARQL Graph Patterns

We define the PSPARQL QUERY EVALUATION decision problems for PSPARQL in the same way as for SPARQL. This problem depends on calculating PRDF homomorphisms, and hence it is parametrized by the PRDF HOMOMORPHISM problem.

PSPARQL QUERY EVALUATION
Instance: An RDF graph G, a PSPARQL graph pattern P and a mapping μ.
Question: Does $\mu \in \mathcal{S}(P, G)$?

We have studied the PSPARQL QUERY EVALUATION problem for basic graph patterns. We have first considered ground graph patterns, which is reduced to checking if a given map is a PRDF homomorphism. So there is no need to seek such a map, and the REGULAR PATH problem is considered in this case (see Appendix). Theorem 5.5.1 shows that PSPARQL QUERY EVALUATION for ground basic graph patterns is no more difficult than REGULAR PATH (defined in Appendix).

Theorem 5.5.1 PSPARQL QUERY EVALUATION *is in* NLOGSPACE *for ground basic graph patterns and* NP*-complete for basic graph patterns.*

Proof. The first assertion (NLOGSPACE for ground PRDF graphs) follows directly from Theorem 4.2.10. For the second assertion (NP-complete), when reduced to

PRDF graphs, PSPARQL QUERY EVALUATION is equivalent to PRDF-GRDF HOMOMORPHISM (Definition 3.3.5). Indeed, PRDF-GRDF HOMOMORPHISM problem can be reduced to PSPARQL QUERY EVALUATION with the empty map μ. In such a case, PSPARQL QUERY EVALUATION is true when there exist a PRDF homomorphism between P and G. On the other way, PSPARQL QUERY EVALUATION is reduced to PRDF-GRDF HOMOMORPHISM between G and $\mu(P)$. Since PRDF-GRDF HOMOMORPHISM is NP-complete, then PSPARQL QUERY EVALUATION is NP-complete for PRDF graphs.

The complexity of PSPARQL QUERY EVALUATION for basic graph patterns is thus the same as SPARQL QUERY EVALUATION for basic graph patterns [Gutierrez *et al.*, 2004]. Since PSPARQL queries are the same as SPARQL queries with the difference of the kind of basic graph patterns and since PSPARQL QUERY EVALUATION for PRDF[$\mathcal{RE}$] graphs is in NP, our extension does not increase the complexity of SPARQL for general graph patterns, *i.e.,* PSPACE-complete [Perez *et al.*, 2006].

5.6 Conclusion

The goal of this chapter was to extend the SPARQL query language that lacks the ability of expressing variable length paths. In order to achieve this goal, we have used PRDF graphs, in which predicates are replaced by regular expression patterns, to define a novel extension to SPARQL, called PSPARQL. Then, we have provided a sound and complete inference mechanism for answering PSPARQL queries over RDF graphs as well as algorithms for calculating these answers. Finally, we proved that the problem of answering PSPARQL queries over RDF graphs remains PSPACE-complete.

The obtained language offers new and useful capabilities with respect to the SPARQL language. However, one could require more facilities in the query language. Some of these are part of our next extension: CPSPARQL.

Constrained Paths in SPARQL 6

Contents

6.1 CPSPARQL by examples 86
6.2 CPRDF: Constrained Paths in RDF 89
6.2.1 CPRDF syntax 89
6.2.2 CPRDF semantics 91
6.2.3 Inference mechanism 94
6.3 The CPSPARQL Query Language 100
6.3.1 CPSPARQL syntax 100
6.3.2 Answers to CPSPARQL queries 101
6.3.3 Formal semantics: answers to CPSPARQL queries . . 102
6.3.4 Complexity of evaluating CPSPARQL graph patterns . 103
6.4 Summary 103

Introduction

The PRDF language extends RDF with path expressions to be able to characterize paths of arbitrary length in a query. However, these queries do not allow expressing constraints on the internal nodes (*e.g.* "Does there exist a trip from town A to town B using only trains and buses such that one of the stops provides a wireless connection.").

We propose in this chapter an extension of PSPARQL, called CPSPARQL. Our definition to CPSPARQL relies on two main issues. The first one comes from the need to extend PSPARQL and thus SPARQL to allow expressing constraints on nodes of traversed paths. The second one comes form the need to enhance the

search process for finding paths that satisfy graph patterns involving path expressions. To this end, we define constraints inside path expressions allowing to reduce the search space by selecting while matching those paths matching path expressions and those nodes satisfying constraints.

In order to achieve these goals, we first define an extension to PRDF, called CPRDF (for Constrained Paths RDF). Syntactically, we define a kind of path expressions, called constrained regular expressions that extends the usual ones with constraints and the inverse operator that changes the orientation of paths. Each constrained regular expression is then used in the predicate position of CPRDF graphs to encode a set of paths such that the internal nodes in these paths satisfy its constraints. Semantically, as done for PRDF, we extend the RDF model-theoretic semantics to allow interpreting this kind of path expressions and to define the entailment between CPRDF and RDF graphs. This is necessary to define answers to CPRDF queries: there exists a solution S to a CPRDF graph P in an RDF graph G if G entails $S(P)$ with respect to this kind of entailment. This leads us to define a kind of graph homomorphism for finding answers to CPRDF graphs (as graph patterns) over RDF graphs. Then, we use CPRDF graphs to generalize SPARQL graph patterns, defining the CPSPARQL extension [Alkhateeb *et al.*, 2008a].

This chapter is divided into three parts: We start in Section 6.1 with some motivating examples which cannot be expressed by (P)SPARQL and require to constrain paths. Sections 6.2 and 6.3 present the CPRDF and CPSPARQL languages, respectively.

6.1 CPSPARQL by examples

The following example queries attempt to give an insight of CPSPARQL.

Example 6.1.1 *Consider the RDF graph G of Figure 6.1, that represents the transportation means between cities, the type of the transportation mean, and the price of tickets. For example, the existence of two triples like* ⟨`flight`, `ex:from`, `C1`⟩ *and* ⟨`flight`, `ex:to`, `C2`⟩ *means that* `C2` *is directly reachable from* `C1` *using* `flight`.

Suppose someone wants to go from Roma to a city in one of the Canary Islands. The following SPARQL query finds the name of such city with only direct trips:

```
SELECT ?City
WHERE { ?Trip ex:from ex:Roma . ?Trip ex:to ?City .
        ?City ex:cityIn ex:CanaryIslands .
```

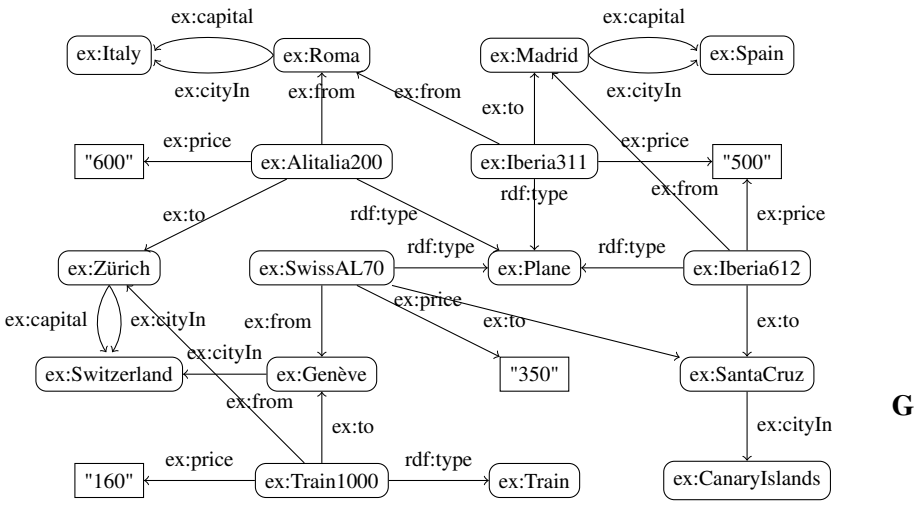

Figure 6.1: An RDF graph.

```
      }
```

Nonetheless, SPARQL cannot express indirect trips with variable length paths. We can express that using regular expressions with the following PSPARQL query:

```
SELECT ?City
WHERE { ex:Roma (ex:from-.ex:to)+ ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

Where "-" is the inverse operator. For example, given the RDF triple (`ex:Roma`, `ex:from`, `ex:flight`), we can deduce (`ex:flight`, `ex:from-`, `ex:Roma`).

Suppose that he/she wants to use only planes. This constraint cannot be emulated in SPARQL or PSPARQL. We can do that in CPSPARQL in the following way. We first define a constraint that consists of a name, interval delimiters to include or exclude path node extremities, a quantifier, and a variable to be substituted by nodes, and a graph to be matched. For example, the name of the constraint in the following query is `const1`, it is open from left and universal which ensures that all trips are of type plane. Then we use the constraint in the regular expression to require that the internal nodes in the path satisfying the regular expression must also satisfy the constraint.

```
SELECT ?City
WHERE { CONSTRAINT const1 ]ALL ?Trip]: { ?Trip rdf:type ex:Plane .
```

```
                                    }
        ex:Roma (ex:from-%const1%.ex:to)+ ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

Moreover, if the user cannot go out the European union, e.g. for visa problem, then we will require all intermediate stops to be cities in Europe.

```
SELECT ?City
WHERE { CONSTRAINT const1 ]ALL ?Trip]: { ?Trip rdf:type ex:Plane . }
        CONSTRAINT const2 ]ALL ?Stop]: { ?Stop ex:cityIn ?Country .
                                         ?Country ex:partOf ex:Europe .
                                    }
        ex:Roma (ex:from-%const1%.ex:to%const2%)+ ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

The price of each direct trip is no more than 500*:*

```
SELECT ?City
WHERE { CONSTRAINT const1 ]ALL ?Trip]: { ?Trip rdf:type ex:Plane .
                                         ?Trip ex:price ?Price .
                                         FILTER (?Price < 500)
                                    }
        CONSTRAINT const2 ]ALL ?Stop]: { ?Stop ex:cityIn ?Country .
                                         ?Country ex:partOf ex:Europe .
                                    }
        ex:Roma (ex:from-%const1%.ex:to%const2%)+ ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

Suppose we want that the price of the whole trip is no more than 1000, then we can use the `SUM` *function in the following query:*

```
SELECT ?City
WHERE { CONSTRAINT const1 SUM(?Sum1,?Price) ]ALL ?Trip]:
                                    { ?Trip rdf:type ex:Plane .
                                      ?Trip ex:price ?Price .
                                      FILTER (SUM(?Sum1,?Price) < 1000)
                                    }
        CONSTRAINT const2 ]ALL ?Stop]: { ?Stop ex:cityIn ?Country .
                                         ?Country ex:partOf ex:Europe .
                                    }
        ex:Roma (ex:from-%const1%.ex:to%const2%)+ ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

As we can see, CPSPARQL is definitely a more expressive language than (P)SPARQL. We will now present it in details.

6.2 CPRDF: Constrained Paths in RDF

In the same way PRDF extends RDF, CPRDF extends RDF and PRDF in order to express properties on nodes that belong to a regular path. For this extension, we provide an abstract syntax (by adding constraints to regular expressions) and an extension of RDF semantics. We characterize query answering (the query is a CPRDF graph, the knowledge base is an RDF graph) as a particular case of CPRDF entailment that can be computed using a kind of graph homomorphism.

6.2.1 CPRDF syntax

For the sake of simplicity and without loss of generality, we restrict the constraints in this section to be GRDF graphs. Then parametrize the CPRDF language in the way that allows us to naturally extend it to include more general constraints as done in Section 7.4.

Constraints

Definition 6.2.1 (GRDF constraint) *A* GRDF constraint *is written* $\dagger_1 Q x \dagger_2 : C$ *where* C *is a GRDF graph,* $\dagger_1$ *and* $\dagger_2$ *are one of the interval delimiters* [*and*]*,* Q *is a quantifier either* `ALL`*,* `EXISTS` *or* `EDGE`*, and* x *is a variable that occurs in a triple of* C*.*

A constraint consists of interval delimiters which are used to include or exclude the extremities of a path; a quantifier either `ALL`, `EXISTS` or `EDGE`; a variable; and a GRDF graph that must be satisfied by the internal nodes. The keyword `EDGE` can be used to indicate that the constraint will be applied to edges (or arcs) while `ALL` and `EXISTS` to indicate that the constraints will be applied to nodes. For example, the constraint defined by]`ALL ?Stop`]: {(`?Stop`, `ex:cityIn`, `?Country`), (`?Country`, `ex:partOf`, `ex:Europe`)} when applied to a regular expression R ensures that all nodes except the source extremity in a path satisfying R are cities in Europe.

In what follows, we use Φ_{GRDF} to denote the set of GRDF constraints. We divide Φ_{GRDF} into two sets, a set Φ^E_{GRDF} of edge constraints and a set Φ^N_{GRDF} of node constraints. When this restriction is not necessary, we use $\Phi = \Phi^E \cup \Phi^N$ to denote a constraint language.

Constrained regular expressions

A constrained regular expression over $(\mathcal{U}, \mathcal{B}, \Phi)$ can be used to define the language over $(\mathcal{U} \cup \mathcal{B})$.

Definition 6.2.2 (Constrained regular expression) *A* constrained regular expression *over* $(\mathcal{U}, \mathcal{B}, \Phi)$ *(denoted by* $R \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*) is defined inductively by:*

- *if* $u \in \mathcal{U}$ *and* $\psi \in \Phi^E$*, then* u*,* $u\%\psi\%$*,* $(!u)$*,* $!u\%\psi\%$*,* u^- *and* $u^-\%\psi\% \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*;*
- *if* $b \in \mathcal{B}$ *and* $\psi \in \Phi^E$*, then* b*,* $b\%\psi\% \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*;*
- *if* $\psi \in \Phi^E$*,* $\#$*,* $\#\%\psi\% \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*;*
- *if* $R \in RE(\mathcal{U}, \mathcal{B}, \Phi)$*, then* $(R^+) \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*;*
- *if* $R_1, R_2 \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*, then* $(R_1 \cdot R_2)$*, and* $(R_1|R_2)$ *are elements of* $\mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*.*
- *if* $R \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*,* $\psi \in \Phi^N$ *is a constraint, then* $R\%\psi\% \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$*.*

The inverse operator $^-$ handles only atomic expressions. It specifies the orientation of arcs in the paths retrieved (*i.e.,* it inverses the matching of arcs). Edge constraints are applied to atomic regular expressions while node constraints are applied to any regular expression. Moreover, the constraints are not necessarily grouped together and we can have a constrained regular expression of the form $R\%\psi_1\% \ldots \%\psi_k\%$. This allows us to specify at each grouped block different constraint with or without different variable(s), which is more flexible and general than grouping all constraints in one block.

CPRDF graphs

Informally, a CPRDF[Φ] graph is a graph whose arcs are labeled with constrained regular expressions whose constraints are elements of Φ.

Definition 6.2.3 (CPRDF graph) *A CPRDF[Φ] triple is an element of* $(\mathcal{T} \times \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi) \times \mathcal{T})$*. A CPRDF[$\Phi$] graph is a set of CPRDF[Φ] triples.*

Example 6.2.4 *The following CPRDF[Φ_{GRDF}] graph H:*

```
{(?City1 ex:cityIn ex:Italy), (?City2 ex:cityIn ex:CanaryIslands),
 (?City1 (ex:from-.ex:to%]ALL ?Stop]:
                              { ?Stop ex:cityIn ?Country .
                                ?Country ex:partOf ex:Europe }%)+ ?City2)
}
```

when used as a query, finds pairs of cities (`?City1`,`?City2`), one in Italy and the other in the Canary Islands, such that `?City2` is reachable from `?City1` by passing through only cities in Europe.

6.2.2 CPRDF semantics

To be able to express the semantics of CPRDF graphs, we have first to define the language generated by a regular expression. The derivation trees used here are just a visual representation of the more usual inductive definition of derivation. The internal nodes of these trees will be used to define the semantics of constraints.

Generated language

Constraints of a given constrained regular expression has no effect on the generated regular language.

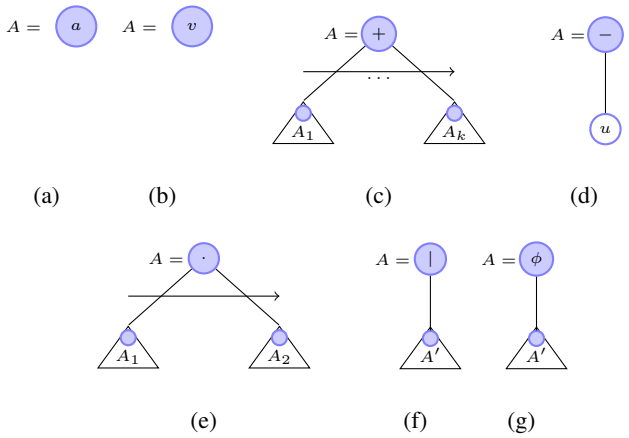

Figure 6.2: Constructing a derivation tree of a constrained regular expression.

Definition 6.2.5 (Derivation tree) *Let $R \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$ be a constrained regular expression. A rooted labeled tree with ordered subtrees A is called a derivation tree of R (denoted $A \in \mathcal{DT}(R)$) iff A can be constructed inductively in the following way:*

1. *if $R = a \in (\mathcal{B} \cup \mathcal{U})$, then A is the tree of Figure 6.2a;*
2. *if $R = (R'^{+})$ and $A_1, \ldots, A_k$ ($k \geq 1$) are a set of derivation trees of $\mathcal{DT}(R')$, then A is the tree of Figure 6.2c;*

3. *if* $R = (u^-)$*, then* A *is the tree of Figure 6.2d;*
4. *if* $R = (R_1 \cdot R_2)$*,* $A_1 \in \mathcal{DT}(R_1)$ *and* $A_2 \in \mathcal{DT}(R_2)$*, then* A *is the tree of Figure 6.2e;*
5. *if* $R = (R_1|R_2)$ *and* $A' \in \mathcal{DT}(R_1) \cup \mathcal{DT}(R_2)$*, then* A *is the tree of Figure 6.2f;*
6. *if* $R = (R'\%\psi\%)$ *and* $A' \in \mathcal{DT}(R')$*, then* A *is the tree of Figure 6.2g.*

The elements of a derivation tree are quantified using path labels in a given graph, and will be illustrated later through an example.

Definition 6.2.6 (Word) *To a derivation tree* A *we associate a unique word* $w(A)$*, obtained by concatenating the labels of the leaves of* A*, totally ordered by the depth-first exploration of* A *determined by the order of its subtrees. We use* $\rho(A, i)$ *to denote the* i^{th} *leaf of* A*, according to that order.*

The word associated to a derivation tree A of a regular expression R belongs to the language generated by R, as usually defined by $L^*(R) = \{w \in (\mathcal{U} \cup \mathcal{B})^+ \mid \exists A \in \mathcal{DT}(R), w = w(A)\}$.

Again, our definition ranges over $(\mathcal{U} \cup \mathcal{B})$ to match predicate variables in GRDF graphs.

Interpretations and models in CPRDF

A CPRDF interpretation of a vocabulary $V \subseteq \mathcal{V}$, is an RDF interpretation of V. However, an RDF interpretation must meet specific conditions to be a model for a CPRDF[Φ] graph (Definition 6.2.9). These conditions are the transposition of the classical path semantics within the RDF semantics (Definition 6.2.7); and the satisfaction of the constraints by the resources of RDF interpretations (Definition 6.2.8).

Definition 6.2.7 (Proof of a constrained regular expression) *Let* $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ *be an interpretation of a vocabulary* V*, and* $R \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$ *be a constrained regular expression such that* $\mathcal{U}(R) \subseteq V$*. Let* ι' *be an extension of* ι *to* $\mathcal{B}(R)$*, and* $w(A) = a_1 \cdot \ldots \cdot a_k$ *be a word of* $L^*(R)$*. A tuple* $(r_0, \ldots, r_k)$ *of resources of* I_R *is called a proof of* w *in* I *according to* ι' *iff* $\forall 1 \leq i \leq k$*:*

- $\langle r_i, r_{i-1} \rangle \in I_{EXT}(\iota'(a_i))$ *if* $\rho(A, i)$ *has an ancestor labeled by* $^-$*;*
- $\langle r_{i-1}, r_i \rangle \in I_{EXT}(\iota'(a_i))$*, otherwise.*

The first item of this definition handles the inverse operator ($^-$): if the ancestor of a_i is labeled by $^-$ (*i.e.,* it is equivalent to a_i^-), then we inverse the two resources that belong to the extension of the property of $\iota'(a_i)$. This definition is used for defining CPRDF models in which it replaces the direct correspondence that exists in RDF between a relation and its interpretation (see first item of Definition 6.2.9), by a correspondence between a constrained regular expression and a sequence of relation interpretations. This allows to match constrained regular expressions with variable length paths as done in Definition 4.2.1 for regular expressions.

Definition 6.2.8 (Constraint satisfaction in an interpretation) *Let $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ be an interpretation of a vocabulary V, and $\psi = \dagger_1 Qx\dagger_2 : C$ be a constraint of Φ_{GRDF}. A resource r of I_R satisfies ψ iff there exists a proof $\iota' : \mathcal{T} \rightarrow I_R$ of C such that $\iota'(x) = r$.*

In what follows, we use $z[\psi](A)$ to denote the subtree A with root node z labeled by constraint ψ. Now we are ready to define when an interpretation is a model of a CPRDF[Φ_{GRDF}] graph.

Definition 6.2.9 (Model of a CPRDF graph) *Let $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ be an interpretation of a vocabulary V, and G be a CPRDF[Φ_{GRDF}] graph such that $\mathcal{U}(G) \subseteq V$. We say that I is a* model *of G iff there exists an extension ι' of ι such that for each triple $\langle s, R, o\rangle$ of G, there exists a sequence $T = (r_0, \ldots, r_k)$ of resources of I_R ($\iota'(s) = r_0$ and $\iota'(o) = r_k$) and a word $w(A) = a_1 \cdot \ldots \cdot a_k \in L^*(R)$ such that:*

- *T is a proof of w in I according to ι';*
- *for each subtree $z[\psi = \dagger_1 Qx\dagger_2 : C](A')$ in A with $a_p \cdot \ldots \cdot a_{p+q} = w(A')$:*
  1. *if Q is* EDGE*, $q = 0$ and $\iota'(a_p)$ satisfies ψ;*
  2. *$Q\ r \in \dagger_1 r_{p-1}, \ldots, r_{p+q}\dagger_2$, r satisfies ψ; otherwise.*

It is shown in the second item of this definition that adding constraints to a CPRDF[Φ] graph reduces the number of models by selecting those ones whose resources satisfy constraints. In addition, since edge constraints are applied to only atomic regular expressions, they constrain only the preceding edge (or arc) label. This is why $q = 0$ in the first sub-item.

Proposition 6.2.10 (Satisfiability) *A CPRDF[Φ_{GRDF}] graph G is satisfiable iff $\forall (s, R, o) \in G$, $L^*(R) \neq \emptyset$.*

Proof. Let G be a CPRDF[Φ_{GRDF}] graph. To prove that G is satisfiable, we build a canonical model as follows:

1. Build a graph G' by replacing each triple $\langle s, R, p\rangle$ in G (if $|R| > 1$) by a set of triples $\langle s, p_1, v_1\rangle \ldots \langle v_{n-1}, p_n, o\rangle$ such that $p_1 \cdot \ldots \cdot p_n$ is an arbitrary word in the language generated by R, and $v_{i's}$ are all new distinct variables; and for each constraint $\psi = \dagger_1 Qx\dagger_2 : C$ in R (Q is `EXISTS` or `ALL` since $|R| > 1$), add to G' the graph C_n^x for each node n in G, where C_n^x is the graph obtained by substituting each occurrence of x by n. If $R = p\%\psi = \dagger_1 Qx\dagger_2 : C\%$, add to G' the graph C_p^x.

2. The obtained graph G' is a GRDF graph, and it is shown that each GRDF graph is satisfiable by building its isomorphic model (see Proposition 2.2.5).

6.2.3 Inference mechanism

Two conditions must be satisfied for the notion of homomorphism to cover the answers of a CPRDF[Φ] query in an RDF knowledge base (Definition 6.2.14): instead of proving an arc (a triple) of the query by an arc in the knowledge base, we prove it by a path in the knowledge base (Definition 6.2.11); and the satisfaction of the node(s) in the path of the knowledge base to the constraint(s) (Definition 6.2.13).

Definition 6.2.11 (Path word) *Let G be an RDF graph of vocabulary $V \subseteq \mathcal{V}$, and $R \in \mathcal{RE}(\mathcal{U}, \mathcal{B}, \Phi)$ be a constrained regular expression such that $\mathcal{U}(R) \subseteq V$. Let $\mu : \mathcal{B}(R) \to V$ be a map from the variables of R to V, and $w(A) = a_1 \cdot \ldots \cdot a_k$ be a word of $L^*(R)$. A sequence $(n_0, \ldots, n_k)$ of nodes of G is called a* path *of w in G according to μ iff $\forall 1 \leq i \leq k$:*

- *$\langle n_i, \mu(a_i), n_{i-1}\rangle \in G$ if $\rho(A, i)$ has an ancestor labeled by $^-$;*
- *$\langle n_{i-1}, \mu(a_i), n_i\rangle \in G$, otherwise.*

As done for the interpretation (Definition 6.2.7), the first item handles the inverse operator: if the ancestor of a_i is labeled by $^-$, then we inverse the orientation of the arc. This definition is equivalent to Definition 4.3.1 used to define path words for language generators, in which we do not handle the inverse operator.

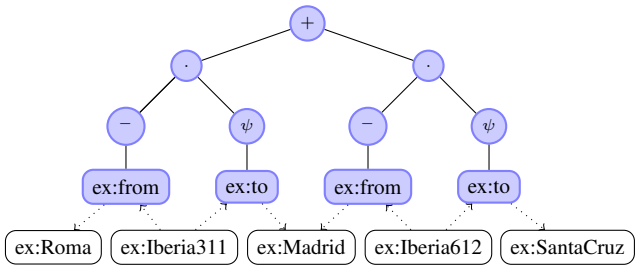

Figure 6.3: Constructing a derivation tree of a constrained regular expression.

Example 6.2.12 *Figure 6.3 shows a possible derivation tree of the constrained regular expression R =(`ex:from`$^-\cdot$`ex:to`$\%\psi\%$)$^+$ of the graph H in Example 6.2.4 with ψ =]`ALL ?Stop`]:{(`?Stop ex:cityIn ?Country`), (`?Country ex:partOf ex:Europe`)}. The nodes in white color, which correspond to the path of nodes in the RDF graph G of Figure 6.1, together with the path labels are used to quantify the elements of the tree. The sequence T=(`ex:Roma`, `ex:Iberia311`, `ex:Madrid`, `ex:Iberia612`, `ex:SantaCruz`) of nodes in the RDF graph G of Figure 6.1 is a path of the word w=(`ex:from`$^-\cdot$ `ex:to`$\cdot$`ex:from`$^-$ $\cdot$`ex:to`) $\in L^*(R)$ according to the empty map.*

The following definition gives the condition(s) when a constraint of Φ_{GRDF} is satisfied, and can be extended based on the constraints (see Section 7.4).

Definition 6.2.13 (Constraint satisfaction in a GRDF graph) *Let G be a GRDF graph, $\psi = \dagger_1 Qx\dagger_2 : C$ be a constraint of Φ_{GRDF}, and s a term of G. Then s satisfies ψ in G if there exists a GRDF homomorphism π from C into G such that $\pi(x) = s$.*

Intuitively, in CPRDF[Φ] homomorphisms, each internal node labeled by a constraint ψ of a derivation tree determines the subtree (not necessary the whole tree, since a constraint ψ may be applied to a partial part of a constrained regular expression, Definition 6.2.2) whose corresponding nodes in the knowledge base graph must satisfy ψ (see the second item of the following definition). Constraints act as filters for paths that must be traversed and select those whose nodes satisfy encountered constraints.

Definition 6.2.14 (CPRDF homomorphism) *Let H be a CPRDF[Φ] graph and G be a GRDF graph. A* CPRDF[Φ] homomorphism *from P into G is a map*

$\pi : \mathcal{T}(H) \rightarrow \mathcal{T}(G)$ such that $\forall(s, R, o) \in H$, there exists a sequence $T = (n_0, \ldots, n_k)$ of nodes of G ($\pi(s) = n_0$ and $\pi(o) = n_k$) and a word $w(A) = a_1 \cdot \ldots \cdot a_k \in L^*(R)$ such that:

- *T is a path of w in G according to π;*
- *for each subtree $z[\psi = \dagger_1 Qx\dagger_2 : C](A')$ in A with $a_p \cdot \ldots \cdot a_{p+q} = w(A')$,*
  1. *if Q is EDGE, $q = 0$ and $\pi(a_p)$[1] satisfies ψ;*
  2. *$Q\ n \in \dagger_1 n_{p-1}, \ldots, n_{p+q}\dagger_2$, n satisfies ψ; otherwise.*

The existence of a CPRDF[Φ] homomorphism is exactly what is needed for deciding entailment between RDF and CPRDF[Φ] graphs.

Theorem 6.2.15 (CPRDF-GRDF entailment) *Let G be a GRDF graph, and H be a CPRDF[Φ_{GRDF}] graph. Then $G \models_{CPRDF} H$ iff there exists a CPRDF[Φ_{GRDF}] homomorphism from H into G.*

Proof. Let G be a GRDF graph, H be a CPRDF[ϕ_{GRDF}] graph and $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ be an interpretation of a vocabulary $V = U \cup L$ such that $\mathcal{V}(G) \subseteq V$ and $\mathcal{V}(H) \subseteq V$. We prove both directions of the theorem as follows. We first add to G, for each triple $\langle s, p, o\rangle$ in G, the triple $\langle s, p^-, o\rangle$. This way we can ignore the first item of Definition 6.2.14 and Definition 6.2.9.
($\Rightarrow$) Suppose that there exists a CPRDF[Φ_{GRDF}] homomorphism from H into G, *i.e.,* $\pi : term(H) \rightarrow term(G)$. We want to prove that $G \models_{\text{CPRDF}} H$, *i.e.,* that every model of G is a model of H.

If I is a model of G, then there exists an extension ι' of ι to $\mathcal{B}(G)$ such that $\forall\langle s, p, o\rangle \in G$, $\langle \iota'(s), \iota'(o)\rangle \in I_{EXT}(\iota'(p))$ (Definition 2.2.3). We want to prove that I is also a model of H, *i.e.,* there exists an extension ι'' of ι to $\mathcal{B}(H)$ such that $\forall\langle s, R, o\rangle \in H$, $\langle \iota''(s), \iota''(o)\rangle$ supports R in I according to ι''.

Let ι'' be the map defined by:

$$\forall x \in \mathcal{T}, \iota''(x) = \begin{cases} (\iota' \circ \pi)(x) & \text{if } \pi \text{ is defined;} \\ \iota'(x) & \text{otherwise.} \end{cases}$$

.

We show that ι'' verifies the following properties:

1. I is an interpretation of $\mathcal{V}(H) \cap nodes(H)$.[2]

[1] When using the wild card #, $\pi(\#)$ is the traversed or the matched edge label.

[2] An interpretation I can be a model of a given CPRDF[Φ] graph H even it does not interpret all terms of H. This is due to the disjunction operator that occurs inside constrained regular expressions.

2. ι'' is an extension to variables of H, *i.e.,* $\forall x \in \mathcal{V}(H) \cap \mathcal{V}(G)$, $\iota''(x) = \iota(x)$.

3. ι'' satisfies the conditions of CPRDF[Φ_{GRDF}] models (Definition 6.2.9), *i.e.,* for every triple $\langle s, R, o\rangle \in H$, the pair of resources $\langle \iota''(s), \iota''(o)\rangle$ supports R in I according to ι''.

Now, we prove the satisfaction of these properties:

1. Since each term $x \in \mathcal{V}(H) \cap nodes(H)$ is mapped by π to a term $x \in \mathcal{V}(G)$ and I interprets all $x \in \mathcal{V}(G)$, I interprets all $x \in \mathcal{V}(H) \cap nodes(H)$.

2. Since π is a map (Definition 6.2.14), we have $\forall x \in \mathcal{V}(H) \cap \mathcal{V}(G)$, if π is defined, $\pi(x) = x$ (Definition 2.3.1). Hence, we have $\iota''(x) = (\iota' \circ \pi)(x) = \iota'(x) = \iota(x), \forall x \in \mathcal{V}(H) \cap \mathcal{V}(G)$.

3. It remains to prove that for every triple $\langle s, R, o\rangle \in H$, the pair of resources $\langle \iota''(\pi(s)), \iota''(\pi(o))\rangle$ supports R in ι'' (Definition 6.2.9). By the definition of CPRDF[Φ_{GRDF}] homomorphisms (Definition 6.2.14), we have:

   (i) $\forall \langle s, R, o\rangle \in H$, there exists a sequence $T = (n_0, \ldots, n_k)$ of nodes of G (with $\pi(s) = n_0$ and $\pi(o) = n_k$) and a word $w(A) = a_1 \cdot \ldots \cdot a_k \in L^*(R)$ such that T is a path of w in G according to π. From the definition of path (Definition 6.2.11), $\langle n_{i-1}, \pi(a_i), n_i\rangle \in G$ such that $n_0 = \pi(s)$, $n_k = \pi(o)$. It follows that $\langle \iota'(\pi(s)), \iota'(n_1)\rangle \in I_{EXT}(\iota'(\pi(a_1)))$, $\ldots$, $(\iota'(n_{k-1}), \iota'(\pi(o))) \in I_{EXT}(\iota'(\pi(a_k)))$ (Definition 2.2.3, GRDF models). So, by Definition 6.2.7, the sequence of resources T_r defined by $T_r = (\iota''(\pi(s)) = \iota'(n_0) = r_0, r_1, \ldots, r_{k-1}, r_k = \iota'(n_k) = \iota''(\pi(o)))$ (with $r_i = n_i$, $1 \leq i \leq k-1$) is a proof of w in I according to $(\iota' \circ \pi)$. Since $\iota'' = (\iota' \circ \pi)$, we have T_r is also a proof of w in I according to ι''.

   (ii) For each subtree $z[\psi = \dagger_1 Qx\dagger_2 : C](A')$ in A with $a_p \cdot \ldots \cdot a_{p+q} = w(A')$ and Q is `EXISTS` or `ALL`, then $Q\ n \in \dagger_1 n_{p-1}, \ldots, n_{p+q}\dagger_2$, n satisfies ψ (the same steps are applied when Q is `EDGE` but this time we take the edge label in G matched to a_p, *i.e.,* $\pi(a_p)$). By Definition 6.2.13, n satisfies ψ in G if there exists a GRDF homomorphism π_1 from C into G such that $\pi_1(x) = n$. Using Theorem 2.3.4 and Definition 2.2.3, there exists a proof $\iota_G : \mathcal{T} \rightarrow I_R$ of C such that $\iota_G(x) = \iota'(n)$. So, $Q\ r \in \dagger_1 r_{p-1}, \ldots, r_{p+q-1}\dagger_2$, r satisfies ψ (with $r_i = \iota'(n_i)$).

The conditions of CPRDF[Φ_{GRDF}] models are satisfied. Hence, every model of G is a model of H.

($\Leftarrow$) Suppose that $G \models_{\text{CPRDF}} H$. We want prove that there is a CPRDF[Φ_{GRDF}] homomorphism from H into G.

Every model of G is also a model of H. In particular, $I_{iso} = \langle I_R, I_P, I_{EXT}, \iota \rangle$ the isomorphic model of G, where there exists a bijection ι' between $term(G)$ and IR (see Proposition 2.2.5). ι' is an extension of ι to $\mathcal{B}(G)$ such that $\forall \langle s, p, o \rangle \in G$, $\langle \iota'(s), \iota'(o) \rangle \in I_{EXT}(\iota'(p))$ (Definition 2.2.3). Since I_{iso} is a model of H, there exists an extension ι'' of I_{iso} to $\mathcal{B}(H)$ such that $\forall \langle s, R, o \rangle$, $\langle \iota''(s), \iota''(o) \rangle$ supports R in ι'' (Definition 6.2.9). Let us consider the function $\pi = (\iota'^{-1} \circ \iota'')$. To prove that π is a CPRDF[Φ_{GRDF}] homomorphism from H into G, we must prove that:

1. π is a map from $term(H)$ into $term(G)$;
2. $\forall x \in \mathcal{V}(H)$, $\pi(x) = x$;
3. $\forall \langle s, R, o \rangle \in H$, the pair of nodes $(\pi(s), \pi(o))$ satisfies R in G according to π.

Let us prove these properties.

1. Since ι'' is a map from $term(H)$ into I_R and ι'^{-1} is a map from I_R into $term(G)$, $\pi = (\iota'^{-1} \circ \iota'')$ is clearly a map from $term(H)$ into $term(G)$ ($term(H) \xrightarrow{\iota''} IR \xrightarrow{\iota'^{-1}} term(G)$).
2. From the definition of an extension: $\forall x \in \mathcal{V}(H)$, $\iota''(x) = \iota(x)$. Since ι' is a bijection, $\forall x \in \mathcal{V}(H)$, $(\iota'^{-1} \circ \iota'')(x) = (\iota'^{-1} \circ \iota)(x) = x$.
3. Since ι'' is a proof of H, by definition of CPRDF[Φ_{GRDF}] models (Definition 6.2.9), we have:

   (i) For each triple $\langle s, R, o \rangle$ of H, there exists a sequence $T = (r_0, \ldots, r_n)$ of resources of I_R (with $\iota''(s) = r_0$ and $\iota''(o) = r_n$) and a word $w(A) = a_1 \cdot \ldots \cdot a_k \in L^*(R)$ such that T is a proof of w in I according to ι''. By Definition 6.2.7, $\langle r_{i-1}, r_i \rangle \in I_{EXT}(\iota''(a_i))$ with $\iota''(s) = r_0$ and $\iota''(o) = r_n$, $1 \leq i \leq k$. It follows that $\langle n_{i-1}, p_i, n_i \rangle \in G$ with $n_i = \iota'^{-1}(r_i)$, and $p_i = (\iota'^{-1} \circ \iota'')(a_i)$ (construction of $I_{iso}(G)$, see Proposition 2.2.5). We have, $(\iota'^{-1} \circ \iota'')(s) = \iota'^{-1}(r_0) = n_0$, $(\iota'^{-1} \circ \iota'')(o) = \iota'^{-1}(r_k) = n_k$, and the word w defined by

$w = p_1 \cdot \ldots \cdot p_k \in L^*((\iota'^{-1} \circ \iota'')(R))$. So the sequence of nodes T_n defined by $T_n = ((\iota'^{-1} \circ \iota'')(s) = \iota'(r_0) = n_0, n_1, \ldots, n_{k-1}, n_k = (\iota'^{-1} \circ \iota'')(o))$ is a path of w in G according to $(\iota'^{-1} \circ \iota'') = \pi$.

(ii) For each subtree $z[\psi = \dagger_1 Q x \dagger_2 : C](A')$ in A with $a_p \cdot \ldots \cdot a_{p+q} = w(A')$, then $Q\, r \in \dagger_1 r_{p-1}, \ldots, r_{p+q} \dagger_2$, r satisfies ψ(the same steps are applied when Q is EDGE but this time we take the resource associated to a_p, *i.e.,* $\iota''(a_p)$). By Definition 6.2.8, r satisfies ψ iff there exists a proof $\iota_G : \mathcal{T} \rightarrow I_R$ of G such that $\iota_G(x) = r$. Using the equivalence between GRDF homomorphism and RDF entailment (Theorem 2.3.4), there exists a GRDF homomorphism π_1 from C into G such that $\pi_1(x) = \iota'^{-1}(r) = n$.

Hence, π is a CPRDF[Φ_{GRDF}] homomorphism from H into G.

We associate to the CPRDF-GRDF entailment the following decision problem:

Φ-CPRDF-GRDF ENTAILMENT
Instance: a GRDF graph G and a CPRDF[Φ] graph H.
Question: Does $G \models_{\text{CPRDF}} H$?

Proposition 6.2.16 Φ_{GRDF}-CPRDF-GRDF ENTAILMENT *is* NP-*complete.*

Proof. Checking if $G \models_{\text{CPRDF}} G'$ is equivalent to checking the existence of a CPRDF[Φ_{GRDF}] homomorphism from G' into G (Theorem 6.2.15). So, it is sufficient to show that checking the existence of a CPRDF[Φ_{GRDF}] homomorphism from G' into G is NP-complete.

When G' does not contain constraints, *i.e.,* G' is a PRDF graph, then the problem is NP-complete (see Chapter 4). We describe an algorithm showing that adding constraints does not change this complexity as follows:

- We first add to G, for each triple $\langle s, p, o\rangle$ in G, the triple $\langle s, p^-, o\rangle$ (which can be done in polynomial time in size of G).
- Calculate all necessary homomorphisms from the graphs of constraints of G' into G a priori only one time (the problem of evaluating a union of GRDF graphs is a NP-complete [Perez *et al.*, 2006]). Suppose that $\Gamma = \{\psi_i \mid \psi_i$ is a constraint in $G'\}$, and Ω_i is the set of homomorphisms from the graph of the constraint ψ_i into G.

– Now, testing whether each node (or edge) n satisfies a given constraint ψ_i in the knowledge base is equivalent to testing if the there exists an homomorphism from the graph of ψ_i into the knowledge base, $\pi \in \Omega_i$ with $\pi(x) = n$, where x is the variable in ψ_i. The latter can be done in linear time in the size of Ω_i (if we assume that checking if $\pi(x) = n$ can be done in $\mathcal{O}(1)$, otherwise it can be in polynomial time).

Example 6.2.17 *Let us consider the CPRDF[Φ_{GRDF}] graph H of Example 6.2.4, the RDF graph G of Figure 6.1, and the map π defined by* {(`?City1`,`ex:Roma`), (`?City2`,`ex:SantaCruz`), (`ex:from`,`ex:from`), (`ex:to`,`ex:to`), (`ex:Italy`,`ex:Italy`), (`ex:cityIn`,`ex:cityIn`), (`?Country`,`ex:CanaryIslands`)}. *According to Definition 6.2.14, the sequence of nodes of Example 6.2.12 (such that the first node* `ex:Roma` *and the last node* `ex:SantaCruz` *are the images of* `?City1` *and* `?City2`*, respectively) is a path of a word of the regular expression of H according to π in G, and the stops along the path are all cities in Europe (see Figure 6.3). So, π is a CPRDF[Φ_{GRDF}] homomorphism from H into G.*

6.3 The CPSPARQL Query Language

The CPSPARQL language extends SPARQL and PSPARQL by using CPRDF graphs instead of GRDF and PRDF to define its graph patterns. Analogously to SPARQL and PSPARQL, the set of answers to a CPSPARQL query is defined inductively using the set of maps (*i.e.,* CPRDF homomorphisms) from the CPRDF graphs of the query into the RDF knowledge base.

6.3.1 CPSPARQL syntax

CPSPARQL[Φ] graph patterns are built on top of CPRDF[Φ] in the same way that SPARQL is built on top of RDF.

Definition 6.3.1 (CPSPARQL graph patterns) *A* CPSPARQL[Φ] graph pattern *is defined inductively by:*

– *every CPRDF[Φ] graph is a CPSPARQL[Φ] graph pattern;*

– *if P_1 and P_2 are two CPSPARQL[Φ] graph patterns and C is a SPARQL constraint, then (P_1* `AND` *P_2), (P_1* `UNION` *P_2), (P_1* `OPT` *P_2), and (P_1* `FILTER` *C) are CPSPARQL[Φ] graph patterns.*

CPSPARQL query. A CPSPARQL[Φ] query is of the form `SELECT` $\vec{B}$ `FROM` u `WHERE` P such that P is a CPSPARQL[Φ] graph pattern.

In CPSPARQL, we can define a constraint and give it a name using the CONSTRAINT clause (see rule [27.1] of Appendix A) such as:

```
CONSTRAINT const1 ]ALL ?Trip]: { ?Trip rdf:type ex:Plane . }
```

This allows to simplify the syntax and to reuse constraints several times.

6.3.2 Answers to CPSPARQL queries

As in the case of RDF/GRDF and RDF/PRDF, the answer to a query reduced to a CPRDF[Φ] graph is also given by a map. The definition of an answer to a CPSPARQL graph pattern will be thus identical to the one given for PSPARQL (Section 5.2) and SPARQL [Perez *et al.*, 2006], but it will use CPRDF[Φ] homomorphisms.

Definition 6.3.2 (Answers to a CPSPARQL graph pattern) *Let G be a (G)RDF graph and P be a CPSPARQL[Φ] graph pattern, then the set $\mathcal{S}(P, G)$ of answers to P in G is defined inductively by:*

- *if P is a CPRDF[Φ] graph, $\mathcal{S}(P, G) = \{\mu \mid \mu$ is a CPRDF[Φ] homomorphism from P into $G\}$;*
- *if $P = (P_1$ `AND` $P_2)$, $\mathcal{S}(P, G) = \mathcal{S}(P_1, G) \bowtie \mathcal{S}(P_2, G)$;*
- *if $P = (P_1$ `UNION` $P_2)$, $\mathcal{S}(P, G) = \mathcal{S}(P_1, G) \cup \mathcal{S}(P_2, G)$;*
- *if $P = (P_1$ `OPT` $P_2)$, $\mathcal{S}(P, G) = (\mathcal{S}(P_1, G) \bowtie \mathcal{S}(P_2, G)) \cup (\mathcal{S}(P_1, G) \setminus \mathcal{S}(P_2, G))$;*
- *if $P = (P_1$ `FILTER` $C)$, $\mathcal{S}(P, G) = \{\mu \in \mathcal{S}(P_1, G) \mid \mu(C) = \top\}$.*

The CONSTRAINT clause has no effect on the answer to a CPSPARQL graph pattern unless the defined constraints are used in the constrained regular expressions. In this case, they affect the CPRDF[Φ] homomorphisms as given in the first item of the definition.

Example 6.3.3 *In the following query:*

```
SELECT ?City
WHERE { CONSTRAINT const1 ]ALL ?Trip]: { ?Trip rdf:type ex:Plane .
                                       }
        ex:Roma (ex:from-.ex:to)+ ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

we have defined a constraint which has not been used, so it has no effects in the answer to the rest of the query. However, in the following query:

```
SELECT ?City
WHERE { CONSTRAINT const1 ]ALL ?Trip]: { ?Trip rdf:type ex:Plane .
                                       }
        ex:Roma (ex:from-%const1%.ex:to)+ ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

the defined constraint is used in the regular expression, which is equivalent to replacing the constraint by itself.

6.3.3 Formal semantics: answers to CPSPARQL queries

The definition of an answer to a CPSPARQL query is the instantiation of maps (as calculated in Definition 6.3.2) to the variables that we want to return.

Definition 6.3.4 (Answer to a CPSPARQL query) *Let $Q =$ `SELECT` $\vec{B}$ `FROM` u `WHERE` P be a CPSPARQL[Φ] query. Let G be the RDF graph identified by the URL u, and Ω the set of answers of P in G. Then the answers to the query Q are the projections of elements of Ω to $\vec{B}$, i.e., for each map π of Ω, the answer of Q associated to π is $\{(x, y) \mid x \in \vec{B}$ and $y = \pi(x)$ if $\pi(x)$ is defined, otherwise* `null`$\}$.

Proposition 6.3.5 *Let G be an RDF graph, P be a CPRDF[Φ_{GRDF}] graph and $\vec{B}$ be a tuple of variables appearing in P, an answer to the CPSPARQL[Φ_{GRDF}] query $Q =$ `SELECT` $\vec{B}$ `FROM` u `WHERE` P is a CPRDF[Φ_{GRDF}] homomorphism μ such that $G \models_{CPRDF} \mu(P)$.*

This proposition is a straightforward consequence of Definition 6.3.2. It is based on the fact that the answers to Q are the restrictions to $\vec{B}$ of the set of CPRDF[Φ_{GRDF}] homomorphisms from P into G which, by Theorem 6.2.15, corresponds to CPRDF-RDF entailment.

In CPSPARQL there are several functions that can be used for capturing the values along the paths like `SUM` for summation of values along paths, `AVG` for the average, `COUNT` for counting nodes satisfying constraints. We have already introduced the `SUM` function in Section 6.1, and we introduce the `COUNT` function in the following example.

Example 6.3.6 *The following CPSPARQL query:*

```
SELECT ?City
WHERE { CONSTRAINT const1 COUNT(?count1) [EXISTS ?Stop]:
                                         { ?Stop ex:cityIn ?Country .
                                           ?Country ex:partOf ex:Europe .
                                           FILTER (COUNT(?count1) >= 2)
                                         }
        ex:Roma (ex:from-.ex:to)+%const1% ?City .
        ?City ex:cityIn ex:CanaryIslands .
      }
```

could be used to find cities (in one of the Canary Islands) reachable from Roma by a path (a sequence of trains or planes) such that at least there are two european cities along the traversed path.

The AVG function is the sum divided by the number of nodes that satisfy the given constraint, *i.e.,* AVG=SUM/COUNT.

6.3.4 Complexity of evaluating CPSPARQL graph patterns

Since CPSPARQL queries are the same as SPARQL queries with the difference of the kind of basic graph patterns (*i.e.,* GRDF vs CPRDF[Φ_{GRDF}]) and CPSPARQL QUERY EVALUATION for CPRDF[Φ_{GRDF}] graphs is in NP, our extension does not increase the worst case complexity of SPARQL, *i.e.,* PSPACE-complete [Perez *et al.*, 2006].

6.4 Summary

Our initial proposal, the PSPARQL language, extends SPARQL with PRDF graphs to allow expressing variable length paths. Since PSPARQL and SPARQL do not allow specifying characteristics of the nodes traversed by a regular path, we have extended the PSPARQL language syntax and semantics to handle constraints, and have characterized answers to a CPRDF query in an RDF knowledge base as maps. This property was sufficient to extend the SPARQL query language to CPSPARQL, combining the expressiveness of both SPARQL and CPRDF. We have provided a sound and complete inference mechanism that can be used for answering CPSPARQL queries over RDF graphs as well as algorithms for calculating these answers.

The proposed language, CPSPARQL has several advantages. First of all, it allows expressing variable length paths which can be qualified through the use of

constraints. It may enhance efficiency, since the task of evaluating path expressions is heavyweight and exhaustive, and the use of predefined constraints inside regular expressions prunes irrelevant paths during the evaluation process and not a posteriori. The constraints in CPSPARQL are extensible (*i.e.,* it can be extended to include constraints that can be more general, as shown in Section 6.3), and partial (*i.e.,* can be applied to a part of a regular expression, see examples in Section 6.1). The use of regular expressions supports a meaningful and natural use of inverse paths through the use of inverse operator. As done for SPARQL, CPRDF graphs can be adapted and integrated in other graph-based query languages.

As it is shown along the paper, we go far beyond the trivial constraints, *i.e.,* testing simple paths and the existence of a node along the path. Extending RDF to RDFS (RDF Schema) does not change the computational properties of the language: finding consequences in RDFS is reduced polynomially to finding consequences in RDF [Hayes, 2004]. So, our work extends naturally to RDFS thanks to this reduction. Finally, we have implemented a CPSPARQL query engine that is available for both download and online test.

7 Other Possible Extensions

Contents

7.1 Path Variables 105
7.1.1 Syntax of path variables 106
7.1.2 Semantics of path variables 106
7.1.3 Distinct answers 109
7.1.4 Constraints on path variables 109
7.2 Similarity-Based Path Matching 110
7.3 Nested Queries 113
7.4 Extending Constraints in CPSPARQL 114
7.5 Conclusion 115

Introduction

The embedding of regular expressions in SPARQL opens the door to several features to be engaged. This chapter discusses some useful extensions to CPSPARQL.

7.1 Path Variables

Path variables are variables that can be used to capture paths. For example, in SPARQLeR [Kochut and Janik, 2007] and SPARQ2L [Anyanwu *et al.*, 2007], they are used instead of regular expressions to capture paths between nodes in RDF graphs. However, in [Kochut and Janik, 2007] and [Anyanwu *et al.*, 2007], they are mapped against paths of triples of arbitrary lengths and then run a post-filtering mechanism for selecting appropriate paths that match a given regular pattern. This

strategy of obtaining paths and then filtering them is inefficient since it can generate a large number of paths.

7.1.1 Syntax of path variables

In contrast to SPARQ2L and SPARQLeR, we use path variables interchangeably with regular expression patterns to express paths. Moreover, we perform a pre-filtering mechanism to paths, that is, appropriate paths that match a given regular expression are selected during the evaluation process and being mapped to a path variable. This is achieved by adding a simple syntax-surface level through the use of the *optional* `DEFINED BY` clause. Analogously to SPARQ2L, path variables in (C)PSPARQL will be prefixed by $??$ (*e.g.* $??pv1$). In what follows, we use X_P to denote an infinite set of path variables.

Definition 7.1.1 (Extended (C)PRDF graphs) *An extended (C)PRDF$[\mathcal{RE}]$ triple, denoted by (C)PRDF$_e[\mathcal{RE}]$, is a triple of $\mathcal{T} \times \mathcal{R}(\mathcal{U}, \mathcal{B}) \cup X_p \times \mathcal{T}$. An extended (C)PRDF$_e[\mathcal{RE}]$ graph is a set of (C)PRDF$_e[\mathcal{RE}]$ triples.*

Extended (C)PSPARQL$_e$ graph patterns (respectively, (C)PSPARQL$_e$ queries) are defined inductively using (C)PRDF$_e[\mathcal{RE}]$ graphs (respectively, (C)PSPARQL$_e$ graph patterns) as done in (C)PSPARQL.

7.1.2 Semantics of path variables

Informally, a triple pattern involving a path variable matches any path between the image of the subject node and the image of the object node. The use of path variables is equivalent to the use of the regular expression $(\#)^+$, with the difference that a path variable is used to match and retrieve paths. Intuitively, when we define path variables using `DEFINED BY`, then words formed along the matched paths must belong to the defined regular expression.

Definition 7.1.2 (Extended (C)PRDF homomorphsims) *Let H_e be an extended (C)PRDF$_e[\mathcal{RE}]$ graph and G be a GRDF graph. Let $(R_1, \ldots, R_n)$ ($n \geq 0$) be the set of regular expressions defined to the set of path variables ($??p_1, \ldots, ??p_n$), respectively. An extended (C)PRDF$_e$ homomorphism from H_e into G is an extended map (i.e., a map $\mu_e : \mathcal{T} \cup X_p \rightarrow \mathcal{T} \cup \mathcal{P}$ preserving constants, where $\mathcal{P}$ denotes an infinite set of paths or sequences of GRDF triples) such that:*

- *$\pi : H \rightarrow G$ is a (C)PRDF homomorphism from H into G, where H is the graph obtained by substituting each R_i to $??p_i$;*

– $\pi_e(??p_i) = p \in \mathcal{P}(G)$ *and* $w(p) \in L^*(\pi(R))$*, where* $w(p)$ *is the word along the path* p*.*

In this definition, each R_i in $(R_1, \ldots, R_n)$ corresponds to the regular expression defined by the `DEFINED BY` clause of the path variable $??p_i$, R_i is $(\#)^+$ when the path variable is used and not defined.

The domain of an extended map μ_e is the subset of $(X_p \cup \mathcal{T})$ in which μ_e is defined. An extended map μ_e is compatible with a map μ_1 if $\forall x \in dom(\mu_e) \cap dom(\mu_1)$, $\mu_e(x) = \mu_1(x)$. The operations on extended maps (like join) are defined in the usual way. The answers to (C)PSPARQL$_e$ graph patterns are constructed inductively from the extended homomorphisms of (C)PRDF$_e$ graphs.

Example 7.1.3 *Consider the following (C)PSPARQL query:*

```
DEFINED BY ??pv1 ex:Paris (ex:train | ex:plane)+ ?City2
SELECT ??pv1 ?City2
WHERE { ex:Paris ??pv1 ?City2 .
        ?City2 ex:cityIn ex:USA .
      }
```

This query searches all USA cities that are reachable from Paris by a sequence of planes and trains, and a possible path will be captured by the path variable `??pv1` *and returned together with that city. Paths must match, while the evaluation process, the regular expression* $(\texttt{ex:train}|\texttt{ex:plane})^+$ *as defined by the* `DEFINED BY` *clause.*

As a path variable can be mapped to an arbitrary-length path, then one might chose either to restrict the language to *simple* (cycle-free) semantics to have complete algorithms (*e.g.* SPARQLeR), or to design algorithms to select shortest paths (*e.g.* SPARQ2L). In our case, we do not need to enumerate all paths but instead we search the existence of paths satisfying (C)PRDF homomorphisms.

Example 7.1.4 *Consider the following (C)PSPARQL query:*

```
DEFINED BY ??pv1 ex:Paris (?Trip)+ ex:Paris
SELECT ??pv1
WHERE { ex:Paris ??pv1 ex:Paris . }
```

and the RDF graph of Figure 7.1. As it is shown in this graph, there are several cycles (going through Amman and Genève) that can generate infinite number of paths. For example, considering non-simple paths, we can generate:

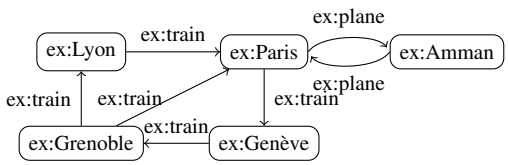

Figure 7.1: An RDF graph with cycles.

$\{\langle$`ex:Paris, ex:plane, ex:Amman, ex:plane, ex:Paris`$\rangle\}$
$\{\langle$`ex:Paris, ex:plane, ex:Amman, ex:plane, ex:Paris, ex:plane, ex:Amman, ex:plane, ex:Paris`$\rangle\}$
etc.

To overcome this problem (i.e., to cut cycles), our evaluation algorithm calculates all possible maps (or homomorphisms in the case of (C)PRDF graphs), which are finite, and those paths satisfying the calculated maps (i.e., visited paths) are mapped to the path variable.

To this end, we can go from Paris to Amman with a map $\{($`?Trip,ex:plane`$)\}$*, then we can return to Paris since the map and/or the state are different from the first visit to Paris. A possible answer therefore is:*

`??pv1`$\rightarrow \{\langle$`ex:Paris,ex:plane,ex:Amman,ex:plane,ex:Paris`$\rangle\}$

A second answer is to go from Paris to Genève through Grenoble, and then Paris with a map $\{($`?Trip,ex:train`$)\}$ *(we can take Paris since the map is different from the first answer):*

`??pv1`$\rightarrow \{\langle$`ex:Paris,ex:train,ex:Genève,ex:train,ex:Grenoble, ex:train,ex:Paris`$\rangle\}$

Now, we can also go from Paris to Genève, through Grenoble, Lyon and then Paris. However, we cannot take this path since Paris is already visited with the same map and state (second answer). The same thing, when we arrive at Genève or Amman for the second time, we cut the cycles since they are already visited with the same map and/or state.

Consider also the following (C)PSPARQL query that we use for illustrating non-simple paths:

```
DEFINED BY ??pv1 ex:Paris (ex:train.ex:plane)+ ?City
SELECT *
WHERE { ex:Paris ??pv1 ?City . }
```

In simple paths, nodes must not be visited more than once. If we consider simple paths in this example, then we cannot retrieve Amman since we cannot go

through the path Paris, Genève, Grenoble, Paris, and then Amman (Paris has been visited twice).

As SPARQL allows the use of DISTINCT key word to ensure disjoint solutions (*i.e.,* no duplicate solutions), one might need to define disjointness solutions involving path variables.

7.1.3 Distinct answers

Informally, two paths are considered the same if they have the same set of triples except the subject of the first triple and the object of the last one can be different. In the same way, two maps (answers) are considered the same if for each variable in one map then the same variable in the second map is binded to the same RDF term, and for each path variable in one map then the same path variable in the second one is binded to the same path. Note that the two solutions are assumed to have the same length.

Definition 7.1.5 (Distinct answers with path variables) *Two maps (answers)* μ_1 *and* μ_2 *are considered to be* distinct *if for each variable* $x \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(x) \neq \mu_2(x)$ *and for each path variable* $??pv \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(??pv) = \mu_2(??pv)$. *Where* $\mu_1(??pv) = \mu_2(??pv)$ *if the two path variables are assigned to the same path of triples except the subject of the first triple and the object of the last triple in the path.*

7.1.4 Constraints on path variables

Some forms of constraints on path variables have been introduced in SPARQ2L and SPARQLeR such as testing whether the path assigned to the path variables contains a node or even a set of nodes, restricting the length of the path, if the path labels match a given regular expression, and others. However, there are missing constraints that can be applied to path variables including but not limited to the following ones.

- $containsOrdered$: $X_P \times 2^{\mathcal{U}} \rightarrow$ boolean;
- $matches$: $X_P \times X_P \rightarrow$ boolean;
- $contains$: $X_P \times X_P \rightarrow$ boolean;
- $isInverse$: $X_P \times X_P \rightarrow$ boolean.

Informally, $containsOrdered(??P_1, \{u_1, u_2, u_3, \ldots, u_m\})$ returns true if the set of nodes are appearing in the path assigned to the path variable $??P_1$ with the same given order, false otherwise. $matches(??P_1, ??P_2)$ returns true if the two path variables $??P_1$ and $??P_2$ are assigned to the same path, false otherwise. $contains(??P_1, ??P_2)$ returns true if the path assigned to $??P_1$ contains the path assigned to $??P_2$. $isInverse(??P_1, ??P_2)$ returns true if the path assigned to $??P_1$ is the inverse of the path assigned to $??P_2$. Note that $containsOrdered$ is a variation of $containsALL$ of SPARQ2L which does not restrict the nodes to be in the given order.

Definition 7.1.6 *Let μ be a map and C be a built-in constraint, then we define the satisfaction relation of μ to C, denoted $\mu(C) = \top$, as follows:*

- *Let $C = containsOrdered(??P_1, \{u_1, u_2, \ldots, u_m\})$, then $\mu(C) = \top$ if $\mu(??P_1) = \langle s, p_1, n_1\rangle \ldots \langle n_k, p_k, o\rangle$ such that:*
  - *$\{u_1, \ldots, u_m\} \subseteq \{n_1, \ldots, n_k\}$; and*
  - *for all $u_i = n_j$ and $u_p = n_r$ either ($i < p$ and $j < r$) or ($i > p$ and $j > r$).*
- *Let $C = matches(??P_1, ??P_2)$, then $\mu(C) = \top$ if $\mu(??P_1) = \mu(??P_2)$, i.e., $\mu(??P_1) = \langle s, p_1, n_1\rangle \ldots \langle n_k, p_k, o\rangle$ and $\mu(??P_2) = \langle x, p_1, n_1\rangle \ldots \langle n_k, p_k, y\rangle$.*
- *Let $C = contains(??P_1, ??P_2)$, then $\mu(C) = \top$ if $\mu(??P_2) \subseteq \mu(??P_1)$.*
- *Let $C = isInverse(??P_1, ??P_2)$, then $\mu(C) = \top$ if $\mu(??P_1) = \langle s, p_1, n_1\rangle \ldots \langle n_k, p_k, o\rangle$ and $\mu(??P_2) = \langle x, p_k, n_k\rangle \ldots \langle n_1, p_1, y\rangle$.*

Example 7.1.7 *The following query:*

```
ASK
WHERE {
       ex:Roma ??P1 ex:Paris .
       ex:Paris ??P2 ex:Roma .
       FILTER (isInverse(??P1,??P2) && (length(??P1)<5)) .
      }
```

returns true if there exists a path from Roma to Paris with length less than 5, which is the inverse of the path from Paris to Roma.

7.2 Similarity-Based Path Matching

Basically, similarity-based query answering is the process of finding *similar or imprecise* answers that match the query. Usually, finding similar answers is achieved

through *query mediation* (or sometimes called query rewriting or transformation) [Papakonstantinou and Vassalos, 1999; Calvanese *et al.*, 2000b]. We present here a new approach for finding similar answers, in particular, for finding similar paths. Before proceeding, let us give a scenario example illustrating the idea behind this approach.

For example, suppose one wants to find USA cities that are reachable from Paris by a path whose predicates are similar to `Vehicle` (or `Transport`). We can express this request by the following query.

```
SELECT ?City1
WHERE {
        ex:Paris (# % SIMILAR(?Pred, ex:Vehicle, 0.7) % )+ ?City2 .
        ?City2 ex:cityIn ex:USA .
      }
```

The constraint `SIMILAR`$(?Pred,$ `ex:Vehicle`$, 0.7)$ indicates that each predicate in the path to be traversed must be similar to `Vehicle`. More precisely, we assign each traversed predicate p to the variable $?Pred$. Then, the constraint is satisfied if the value returned from the similarity measure `SIMILAR`$(p,$ `ex:Vehicle`, $0.7)$ (until now, we use cosynonym as a default similarity measure and we plan to use other similarity measures in the `SIMILAR` qualifier) is greater than the thresholding value 0.7 (default value is 0.5 if it is not specified). The same query also could be alternatively expressed using the constraints on edge as given in the following query.

```
SELECT ?City1
WHERE {
        CONSTRAINT const1 [EDGE ?P]:
                                { ?S ?P ?O .
                                  SIMILAR(?P, ex:Vehicle, 0.7)
                                }
        ex:Paris (# % const1 % )+ ?City2 .
        ?City2 ex:cityIn ex:USA .
      }
```

This approach is different from the one in [Kiefer *et al.*, 2007] wherein a new extension to SPARQL, called iSPARQL, is proposed by allowing for similarity joins measures. The novelty of our approach relies upon allowing similarity-based path matching, and applying it to CPSPARQL.

There are many ways to assess the similarity between entities (or terms) [Euzenat and Shvaiko, 2007]. The most common way amounts to defining a measure of this similarity.

Definition 7.2.1 *A similarity $\alpha : o \times o \rightarrow \mathbb{R}$ is a function from a pair of entities to a real number expressing the similarity between them such that:*

$$\forall x, y \in o, \alpha(x, y) \geq 0 \quad \textit{(positiveness)}$$
$$\forall x, y, z \in o, \alpha(x, x) \geq \alpha(y, z) \quad \textit{(maximality)}$$
$$\forall x, y \in o, \alpha(x, y) = \alpha(y, x) \quad \textit{(symmetry)}$$

Several techniques (or methods) could be used for assessing the similarity measure or relation between entities [Euzenat and Shvaiko, 2007]. We rely upon those that are based on using external resources like WordNet[1]. WordNet is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. WordNet also provides relations such as an hypernym (superconcept/subconcept) structure, meronym (part of) relation, etc.

Definition 7.2.2 (Partially ordered synonym resource) *A partially ordered synonym resource $\mathcal{P}$ over a set of words W, is a triple $\langle E, \leq, \beta \rangle$, such that $E \subseteq 2^W$ is a set of synsets, $\leq$ is the hypernym relation between synsets and β is a function from synsets to their definition (a text that is considered here as a bag of words i W). For a term t, $\mathcal{P}(t)$ denoted the set of synsets associated with t.*

Simple measures can be defined based on synonymous relation of WordNet. We consider the cosynonym similarity measure since it is simple and not a strict measure, *i.e.,* it allows calculating similarity with respect to non synonymous objects. Of course, more elaborated measures could be used [Euzenat and Shvaiko, 2007] such as Resnik semantic similarity [Resnik, 1995].

Definition 7.2.3 (Cosynonym similarity) *Given two terms s and t and a synonym resource $\mathcal{P}$, the cosynonym is a similarity $\alpha : \mathbb{S} \times \mathbb{S} \rightarrow [0\ 1]$ such that:*

$$\alpha(s, t) = \frac{|\mathcal{P}(s) \cap \mathcal{P}(t)|}{|\mathcal{P}(s) \cup \mathcal{P}(t)|}$$

Note that we ignore the uriref namespaces when calculating the similarity even if they are different. For example, if `ex1:car` and `ex2:bus` are two terms, only `car` and `bus` are considered.

[1] `wordnet.princeton.edu/`

Similarity-based path matching feature may enhance the search process by selecting similar paths and ignoring meaningless and useless ones. In particular, a path variable without a pre-defined regular expression can match any path regardless to its arc labels (*i.e.,* the word obtained by concatenating path labels), and this may yield a large number of answers to be returned for a query involving path variables. Using the similarity path matching feature with, for example, path variables selects only paths whose labels are similar to the given property.

7.3 Nested Queries

The idea of allowing nested queries in SPARQL is not new. For instance, [Polleres, 2007] suggests a simple form of nested queries, *i.e.,* boolean queries (ASK queries) with an empty result form, that can be used within FILTER expressions. We present another simple but useful form of nested queries: CONSTRUCT queries that are allowed to be used within the FORM clause. This extension is useful in complex modeling, for example, when one wants to query the result of another query with some modification made to the RDF knowledge base if for example some criterion satisfied. The following example illustrates the usefulness of this extension.

Example 7.3.1 *Consider the RDF graph of Figure 7.2. Suppose we want to find all cities reachable from Amman with a sequence (or a path) of flights such that the arrival time of each flight is always before the departure time of the next one in the path. This request cannot be expressed by a query in CPSPARQL. However, we can express it using a nested query in the following way:*

```
SELECT ?City2
FROM   CONSTRUCT { ?Flight3 ex:reachable ?Flight4 }
       FROM  u1
       WHERE {
              ?Flight3 ex:arrival ?Arrival .
              ?Flight3 ex:to ?City .
              ?Flight4 ex:departure ?Departure .
              ?Flight4 ex:from ?City .
              FILTER (?Arrival  < ?Departure )
             }

FROM   u1
WHERE  {
        ?Flight1 (ex:reachable)+ ?Flight2 .
        ?Flight1 ex:from ex:Amman .
        ?Flight2 ex:to ?City2 .
       }
```

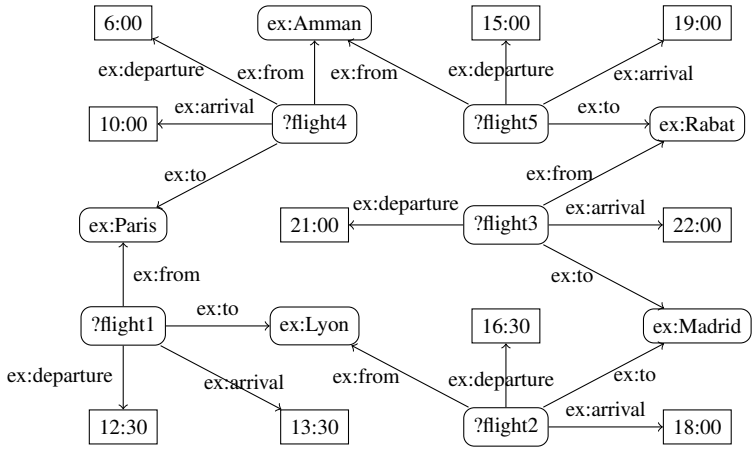

Figure 7.2: An RDF graph representing flights time table.

In which, we first calculate a new graph containing the reachable information between source and destination nodes whenever there are two flights such that the destination of the first flight is the source of the second one, and the arrival time of the first one is before the departure time of the second one. Then, we can query the constructed graph together with the initial one using the reachable information to find the reachable cities from Amman.

7.4 Extending Constraints in CPSPARQL

The parametrization of CPSPARQL[Φ] by Φ allows us to extend naturally its graph patterns to more general constraints. For example, if Φ_{PSPARQL} denotes the set of all possible PSPARQL graph patterns, then a CPRDF[Φ_{PSPARQL}] graph could be a CPSPARQL[Φ_{PSPARQL}] graph pattern.

If CPSPARQL graph patterns are constructed over CPRDF[Φ_{PSPARQL}] graphs, then we need only to extend Definition 6.2.13 in the following way: let G be a graph, P be a PSPARQL graph pattern, $\psi = \dagger_1 Qx \dagger_2 : P$ be a constraint, and s a term of G. We say that s satisfies ψ in G if there exists a map $\mu \in \mathcal{S}(P, G)$ such that $\mu(x) = s$. The definition of CPRDF[Φ] homomorphism (Definition 6.2.14) and first item of Definition 6.3.2 remain unchanged.

Example 7.4.1 *The following CPSPARQL[$\Phi_{PSPARQL}$] query:*

```
SELECT ?City
```

```
WHERE { CONSTRAINT const1 ]ALL ?Stop]:
                                { ?Stop ex:population ?Pop .
                                  FILTER (?Pop > 10000) .
                                }
        ex:Paris (ex:train | ex:plane)+ %const1% ?City .
      }
```

returns cities connected to Paris by a sequence of trains or planes such that all intermediate cities along the traversed paths have population size more than 10000.

The following CPSPARQL[$\Phi_{PSPARQL}$] query:

```
SELECT ?City
WHERE { CONSTRAINT const1 ]ALL ?Stop]:
                           {{ ?Stop ex:population ?Pop .
                              FILTER (?Pop > 10000) .
                              }
                              UNION
                            { ?stop ex:capitalOf ?Country .}
                           }
        ex:Paris (ex:train | ex:plane)+ %const1% ?City .
      }
```

returns cities connected to Paris by a sequence of trains or planes such that all intermediate cities along the traversed paths either are capital cities or have population size more than 10000.

7.5 Conclusion

This chapter has presented several extensions which we found useful for real applications of both SPARQL and CPSPARQL. Some of them are original (similarity path-based matching, nested construct queries and extending constraints in CPSPARQL); some of them have already been proposed in the literature (path variables and constraints on path variables), but we have provided other ways to use them or extended their functionalities. More precisely, we have used path variables differently from the existing query languages in a way that permits to match paths to a given regular expression while doing the search process and provided new forms of post-filtering constraints on these path variables.

8 Querying RDFS Graphs

Contents

8.1 RDF(S) 119
8.1.1 RDF(S) vocabulary 119
8.1.2 RDF(S) semantics 119
8.2 RDF(S) Closure and Query Answering 121
8.3 RDF(S) Entailment and Query Rewriting 123
8.3.1 FROM SPARQL/RDFS to PSPARQL/RDF 123
8.3.2 FROM PSPARQL/RDFS to CPSPARQL/RDF 125
8.3.3 Example: putting all together 128
8.4 Conclusion 129

Introduction

RDF [Manola and Miller, 2004] and its extension RDFS (RDF Schema) [Brickley and Guha, 2004] together with OWL [McGuinness and van Harmelen, 2004] form the three formal logics recommended by W3C for representing data in semantic web. The focus in this chapter however will be only in RDF and RDFS languages that extend the simple RDF language presented in Chapter 2. The two extensions are defined in the same way:

- they consider a particular set of urirefs of the vocabulary prefixed by `rdf:` and `rdfs:`, respectively.
- They add additional constraints to the resources associated to these terms in the interpretation.

In adding new constraints to RDF(S) interpretations, RDF(S) documents may have less models, and thus more consequences. It is possible for example, in RDF, to deduce ⟨`ex:author rdf:type rdf:Property`⟩ from ⟨`ex:person1 ex:author "Alkhateeb"`⟩; in RDFS, to deduce ⟨`ex:document1 rdf:type ex:Biography`⟩ from {⟨`ex:document1 rdf:type rdf:Autobiography`⟩, ⟨`ex:Autobiography rdfs:subClassOf ex:Biography`⟩}.

One possible approach for querying an RDF(S) graph G in a sound and complete way is by computing the so-called *closure* graph of G, then evaluating the query over the closure graph.

Another possible approach [Muñoz *et al.*, 2007], consists of searching paths between RDF(S) vocabularies. This approach gives a more efficient algorithm ($\mathcal{O}(nlogn)$ time complexity) for checking only the entailment between ground RDF(S) (or more precisely, restricted RDF(S)) graphs than using the closure operation. This algorithmic result also directly follows from our polynomial result of path satisfiability checking and the PRDF homomorphism [Alkhateeb *et al.*, 2007] of ground graphs. Despite its usefulness in many applications (*e.g.* boolean queries), the proposed algorithm cannot be used for the query evaluation problem or even — the simpler problem — checking the entailment between RDF(S) graphs involving variables.

To overcome the limitation of both approaches, we provide a new approach for answering queries over RDF(S) graphs. Our approach consists of rewriting the query using a set of rules, and then evaluating the transformed query over the graph to be queried. The query rewriting approach that we will present is similar in spirit to the query rewriting methods using a set of views [Papakonstantinou and Vassalos, 1999; Calvanese *et al.*, 2000b; Grahne and Thomo, 2003]. In contrast to these methods, our approach uses the data contained in the graph (*i.e.,* the rules are inferred from RDF(S) entailment rules).

Before proceeding, let us first introduce the RDF(S) language (its vocabulary and semantics), recall the closure method for checking RDF(S) consequence and its drawbacks. Then, we present our approach for querying RDF(S) graphs.

Section 8.1 of this chapter is dedicated to the presentation of RDF(S) languages, Section 8.2 and Section 8.3 present the closure approach and the rewriting approach for querying RDF(S) knowledge bases, respectively.

rdfs:domain[`dom`]	rdfs:Container[`cont`]	rdfs:Resource[`res`]
rdfs:range[`range`]	rdfs:isDefinedBy[`isDefined`]	rdf:subject[`subj`]
rdfs:Class[`class`]	rdfs:Literal[`literal`]	rdf:first[`first`]
rdf:value[`value`]	rdfs:subClassOf[`sc`]	rdf:Property[`prop`]
rdfs:label[`label`]	rdfs:subPropertyOf[`sp`]	rdf:rest[`rest`]
rdf:nil[`nil`]	rdfs:comment[`comment`]	rdf: 1[`1`]
rdf:type[`type`]	rdf:predicate[`pred`]	rdf: 2[`2`]
rdf:object[`obj`]	rdf:Statement[`stat`]	...
rdf:List[`list`]	rdfs:member[`member`]	rdf: i[`i`]
rdf:Alt[`alt`]	rdfs:Datatype[`datatype`]	...
rdf:Bag[`bag`]	rdf:XMLLiteral[`xmlLit`]	
rdf:Seq[`seq`]	rdfs:seeAlso[`seeAlso`]	
	rdfs:ContainerMembership Property[`contMP`]	

Table 8.1: The RDF(S) Vocabulary.

8.1 RDF(S)

8.1.1 RDF(S) vocabulary

In RDF, there exists a set of reserved words, the RDF(S) vocabulary (RDF Schema [Brickley and Guha, 2004]), designed to describe relationships between resources like classes (*e.g.* `classA subClassOf classB`) and relationships between properties (*e.g.* `propA subPropertyOf propB`). The RDF(S) vocabulary is given in Table 8.1 as it appears in [Hayes, 2004]. The shortcuts that we will use for each of them are given in brackets.

From now on, we use $rdfsV$ to denote the RDF(S) vocabulary.

8.1.2 RDF(S) semantics

In addition to the usual interpretation mapping, special mapping is used in RDFS interpretations to allow interpreting the set of classes which is a subset of I_R.

Definition 8.1.1 *An RDFS interpretation of a vocabulary V is a tuple $I = \langle I_R, I_P, Class, I_{EXT}, I_{CEXT}, Lit, \iota\rangle$ such that:*

- *$Class \subseteq I_R$ is a distinguished subset of I_R identifying if a resource denotes a class of resources;*
- *$I_{CEXT} : Class \rightarrow 2^{I_R}$ is a mapping that assigns a set of resources to every resource denoting a class;*

– *$Lit \subseteq I_R$ is the set of literal values, Lit contains all plain literals in $\mathcal{L} \cap V$.*

The remainder are defined as in the simple interpretations.

Additional conditions are added to the resources associated to terms of RDF(S) vocabularies in an RDF(S) interpretation to be an RDF(S) model of an RDF(S) graph. These conditions include the satisfaction of the RDF(S) axiomatic triples as appeared in the normative semantics of RDF [Hayes, 2004].

Definition 8.1.2 (RDF(S) Model) *Let G be an RDF(S) graph, and $I = \langle I_R, I_P, Class, I_{EXT}, I_{CEXT}, Lit, \iota\rangle$ be an RDFS interpretation of a vocabulary $V \subseteq rdfsV \cup \mathcal{V}$ such that $\mathcal{V}(G) \subseteq V$. Then I is an RDF(S) model of G if and only if I satisfies the following conditions:*

1. *Simple semantics:*
   a) *there exists an extension ι' of ι to $\mathcal{B}(G)$ such that for each triple $\langle s, p, o\rangle$ of G, $\iota'(p) \in I_P$ and $\langle \iota'(s), \iota'(o)\rangle \in I_{EXT}(\iota'(p))$.*
2. *RDF semantics:*
   a) $x \in I_P \Leftrightarrow \langle x, \iota'(\texttt{prop})\rangle \in I_{EXT}(\iota'(\texttt{type}))$.
   b) *If $\ell \in term(G)$ is a typed XML literal with lexical form w, then $\iota'(\ell)$ is the XML literal value of w, $\iota'(\ell) \in Lit$, and $\langle \iota'(\ell), \iota'(\texttt{xmlLit})\rangle \in I_{EXT}(\iota'(\texttt{type}))$.*
3. *RDFS Classes:*
   a) $x \in I_R$, $x \in I_{CEXT}(\iota'(\texttt{res}))$.
   b) $x \in Class$, $x \in I_{CEXT}(\iota'(\texttt{class}))$.
   c) $x \in Lit$, $x \in I_{CEXT}(\iota'(\texttt{literal}))$.
4. *RDFS Subproperty:*
   a) *$I_{EXT}(\iota'(\texttt{sp}))$ is transitive and reflexive over I_P.*
   b) *if $\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{sp}))$ then $x, y \in I_P$ and $I_{EXT}(x) \subseteq I_{EXT}(y)$.*
5. *RDFS Subclass:*
   a) *$I_{EXT}(\iota'(\texttt{sc}))$ is transitive and reflexive over $Class$.*
   b) *$\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{sc}))$, then $x, y \in Class$ and $I_{CEXT}(x) \subseteq I_{CEXT}(y)$.*
6. *RDFS Typing:*
   a) $x \in I_{CEXT}(y)$, $(x, y) \in I_{EXT}(\iota'(\texttt{type}))$.

b) if $\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{dom}))$ *and* $\langle u, v\rangle \in I_{EXT}(x)$ *then* $u \in I_{CEXT}(y)$.

c) if $\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{range}))$ *and* $\langle u, v\rangle \in I_{EXT}(x)$ *then* $v \in I_{CEXT}(y)$.

[a)]

7. *RDFS Additionals:*

   a) if $x \in Class$ *then* $\langle x, \iota'(\texttt{res})\rangle \in I_{EXT}(\iota'(\texttt{sc}))$.

   b) if $x \in I_{CEXT}(\iota'(\texttt{datatype}))$ *then* $\langle x, \iota'(\texttt{literal})\rangle \in I_{EXT}(\iota'(\texttt{sc}))$.

   c) if $x \in I_{CEXT}(\iota'(\texttt{contMP}))$ *then* $\langle x, \iota'(\texttt{member})\rangle \in I_{EXT}(\iota'(\texttt{sp}))$.

Definition 8.1.3 (RDFS consequence) *Let G and H be two RDFS graphs, then G RDFS entails H, denoted by $G \models_{RDFS} H$, iff every RDFS model of G is also an RDFS model of H.*

8.2 RDF(S) Closure and Query Answering

One possible approach for querying an RDF(S) graph G in a sound and complete way is by computing the closure graph of G, *i.e.,* the graph obtained by saturating G with all informations that can be deduced using a set of predefined rules called RDF(S) rules, then evaluating the query over the closure graph.

Let G be an RDF(S) graph of an RDF(S) vocabulary V. The RDF(S) closure of G, where $\hat{G}$ denotes the closure of the RDF(S) graph G, is obtained in the following way:

[RDF1]	add all RDF axiomatic triples to $\hat{G}$;
[RDF2]	if $\langle s, p, o\rangle$ in $\hat{G}$, then $\langle p, \texttt{type}, \texttt{prop}\rangle$ is a triple of $\hat{G}$;
[RDF3]	if $\langle s, p, \ell\rangle$ is a triple of $\hat{G}$, where ℓ is an `xmlLit` typed literal and the lexical representation s is a well-formed XML literal, then $\langle s, p, xml(s)\rangle$ and $\langle xml(s), \texttt{type}, \texttt{xmlLit}\rangle$ are two triples of $\hat{G}$;
[RDFS 1]	add all RDFS axiomatic triples to $\hat{G}$;
[RDFS 6]	if $\langle a, \texttt{dom}, x\rangle$ and $\langle u, a, y\rangle$ are two triples of $\hat{G}$, then $\langle u, \texttt{type}, x\rangle$ is a triple of $\hat{G}$;
[RDFS 7]	if $\langle a, \texttt{range}, x\rangle$ and $\langle u, a, v\rangle$ are triples of $\hat{G}$, then $\langle v, \texttt{type}, x\rangle$ is a triple of $\hat{G}$;
[RDFS 8A]	if $\langle x, \texttt{type}, \texttt{prop}\rangle$ in $\hat{G}$, then $\langle x, \texttt{sp}, x\rangle$ is a triple of $\hat{G}$;
[RDFS 8B]	if $\langle x, \texttt{sp}, y\rangle$ and $\langle y, \texttt{sp}, z\rangle$ are two triples of $\hat{G}$, then $\langle x, \texttt{sp}, z\rangle$ is a triple of $\hat{G}$;

[RDFS 9] if $\langle a, \texttt{sp}, b\rangle$ and $\langle x, a, y\rangle$ are two triples of $\hat{G}$, then $\langle x, b, y\rangle$ is a triple of $\hat{G}$;

[RDFS 10] if $\langle x, \texttt{type}, \texttt{class}\rangle$ in $\hat{G}$, then $\langle x, \texttt{sc}, \texttt{res}\rangle$ is a triple of $\hat{G}$;

[RDFS 11] if $\langle u, \texttt{sc}, x\rangle$ and $\langle y, \texttt{type}, u\rangle$ are triples of $\hat{G}$, then $\langle y, \texttt{type}, x\rangle$ is a triple of $\hat{G}$;

[RDFS 12A] if $\langle x, \texttt{type}, \texttt{class}\rangle$ is a triple of $\hat{G}$, then $\langle x, \texttt{sc}, x\rangle$ is a triple of $\hat{G}$;

[RDFS 12B] if $\langle x, \texttt{sc}, y\rangle$ and $\langle y, \texttt{sc}, z\rangle$ are two triples of $\hat{G}$, then $\langle x, \texttt{sc}, z\rangle$ is a triple of $\hat{G}$;

[RDFS 13] if $\langle x, \texttt{type}, \texttt{contMP}\rangle$ is a triple of $\hat{G}$, then $\langle x, \texttt{prop}, \texttt{member}\rangle$ is a triple of $\hat{G}$;

[RDFS 14] $\langle x, \texttt{type}, \texttt{datatype}\rangle$ is a triple of $\hat{G}$, then $\langle x, \texttt{sc}, \texttt{literal}\rangle$ is a triple of $\hat{G}$.

A closure operation that can be applied to an RDF(S) graph permits to reduce the RDF(S) entailment to simple RDF entailment. A finite and polynomial closure, called *partial closure*, is proposed independently in [Baget, 2003; Horst, 2005]. Let G and H be two RDFS graphs on an RDFS vocabulary V. The partial closure of G given H, denoted $\hat{G}\backslash H$, is obtained in the following way:

1. let k be the maximum of i's such that `rdf:_i` is a term of G or of H;
2. replace the rule [RDF 1] by the rule [RDF 1P] add all RDF axiomatic triples except those that use `rdf:_i` with $i > k$. In the same way, replace the rule [RDFS 1] by the rule [RDFS 1P] add all RDFS axiomatic triples except those that use `rdf:_i` with $i > k$;
3. apply the modified rules.

Theorem 8.2.1 ([Hayes, 2004]) *Let G and H be satisfiable RDFS graphs, then $G \models_{RDFS} H$ if and only if $(\hat{G}\backslash H) \models H$.*

From this results and the equivalence between the entailment and homomorphisms (Theorem 2.3.4 and Theorem 4.3.5), it is thus possible to use homomorphisms for checking the RDF(S) consequences.

Corollary 8.2.2 (Homomorphisms and RDF(S) entailment) *Let G be a satisfiable RDFS graph and H be a graph, then $G \models_{RDFS} H$ iff there exists an homomorphism from H into $(\hat{G}\backslash H)$.*

In this result, the homomorphism used corresponds to the kind of the graph H which can be an RDFS, a PRDF or a CPRDF graph. For the query evaluation problem, it is sufficient to enumerate the set of homomorphisms from the query graph pattern(s) into the closure graph.

As we mentioned before, this approach has several drawbacks which limited its use. It takes time proportional to $|H| \times |G|^2$ in the worst case [Muñoz *et al.*, 2007]. Moreover, it is not applicable, for example, in the case when we do not have access to the graph to be queried. In this case, we cannot calculate the closure graph. If it is not the case, then we need to download the RDF(S) graph to calculate locally its closure. Finally, the finite closure needs to be recalculated at each time we ask a query.

8.3 RDF(S) Entailment and Query Rewriting

In this section, we present a rewriting method for evaluating SPARQL or (C)PSPARQL queries over RDF(S) graphs. This method captures RDF(S) semantics, in particular, the core fragment introduced in [Muñoz *et al.*, 2007].

8.3.1 FROM SPARQL/RDFS to PSPARQL/RDF

We give in this subsection a rewriting system for evaluating SPARQL queries over RDF(S) graphs. In particular, we show that every SPARQL query Q that will be evaluated over an RDF(S) graph G can be transformed to a PSPARQL query Q' such that evaluating Q over $\hat{G}$, the closure graph of G, is equivalent to evaluating Q' over G. The system consists of a set of rewriting rules of the form $\tau : g \rightarrow g'$, where g is a basic graph and g' is a PSPARQL graph pattern. g' is obtained from g by applying the possible rule(s) to each triple in g, *i.e.,* $g' = \tau(g) = \{\tau(t) \mid t$ is a triple in $g\}$. In every rule, s and o are elements from the RDF terminology, *i.e.,* literals, urirefs, or variables.

Note that the input of the system is a basic graph and not a SPARQL graph pattern. This is because the evaluation of a SPARQL graph pattern is composed from the evaluation of the basic graphs that make the query (see Definition 3.3.5). To illustrate the approach, let us consider ρdf [Muñoz *et al.*, 2007], the subset of RDF(S) that contains the following vocabulary:

$$\rho\text{df} = \{\texttt{sp}, \texttt{sc}, \texttt{type}, \texttt{dom}, \texttt{range}\}$$

This subset forms the core fragment of RDF(S) for the RDF language developers use as indicated in [Muñoz *et al.*, 2007].

SubClass rule:

$$\tau(\langle s, \texttt{sc}, o\rangle) = \langle s, \texttt{sc}^+, o\rangle$$

This rule handles the transitive semantics of the subclass relation. Finding the subclasses of a given class can be achieved by navigating all its direct subclasses.

Subproperty rule:

$$\tau(\langle s, \texttt{sp}, o\rangle) = \langle s, \texttt{sp}^+, o\rangle$$

This rule handles the transitive semantics of the subproperty relation. Finding the subproperties of a given property can be achieved by navigating all its direct subproperties.

$$\tau(\langle s, p, o\rangle) = \{\langle s, ?x, o\rangle, \langle ?x, \texttt{sp}^*, p\rangle\}$$

This rule shows that the subject-object pairs occurred in the subporperties of a given property are inherited to it, where p is an urirefs.

Typing rule:

$$\begin{array}{lll} \tau(s, \texttt{type}, o) & = & \{\langle s, \texttt{type} \cdot \texttt{sc}^*, o\rangle\} \\ & \texttt{UNION} & \{\langle s, ?p_1, ?y\rangle, \langle ?p_1, \texttt{sp}^*, ?p_2\rangle, \langle ?p_2, \texttt{dom} \cdot \texttt{sc}^*, o\rangle\} \\ & \texttt{UNION} & \{\langle ?y, ?p_1, s\rangle, \langle ?p_1, \texttt{sp}^*, ?p_2\rangle, \langle ?p_2, \texttt{range} \cdot \texttt{sc}^*, o\rangle\} \end{array}$$

This rule shows that the instance mapped to s is of type the class mapped to o (we use the word "mapped" since s and/or o can be variables) if one of the following conditions holds:

1. the instance mapped to s is a type of one of the subclasses of the class mapped to o by following the subclass relationship zero or several times. The zero time is used since s can be directly of type o;

2. if there exists a property such that the instances appearing as a subject are all of type of one of the subclasses mapped to o;

3. if there exists a property such that the instances appearing as an object are all of type of one of the subclasses mapped to o.

From the reflexivity semantics of sp and sc, we can deduce that any class (respectively, property) is a subclass (respectively, subproperty) of itself. We can deduce, for example, from $\langle p_1 \ \texttt{sp} \ p_2 \rangle$ that $\langle p_1 \ \texttt{sp} \ p_1 \rangle$ and $\langle p_2 \ \texttt{sp} \ p_2 \rangle$. We assume that sp and sc are reflexive relaxed relations as done in [Muñoz *et al.*, 2007]. With this assumption, we have the following property.

Proposition 8.3.1 *Let G and P be two ρdf graphs, then $Eval(P, \hat{G}, \Omega)$ is equivalent to $Eval(\tau(P), G, \Omega)$.*

8.3.2 FROM PSPARQL/RDFS to CPSPARQL/RDF

The same ideas could be used when evaluating (C)PSPARQL queries. More precisely, to evaluate a (C)PSPARQL query Q over an RDF(S) graph G, either we calculate first the closure graph of G (*i.e.,* $\hat{G}$) and then we evaluate Q over $\hat{G}$ or we rewrite Q into a semantically equivalent one Q' and then evaluate Q' over G.

For example, given the following PSPARQL query Q:

```
SELECT ?City
WHERE { ex:Grenoble ex:transport+ ?City .}
```

and the following RDF(S) graph G:

```
{
   ⟨ex:train        sp        ex:transport ⟩,
   ⟨ex:plane        sp        ex:transport ⟩,
   ⟨ex:Grenoble   ex:train    ex:Lyon ⟩,
   ⟨ex:Lyon       ex:train    ex:Paris ⟩,
   ⟨ex:Lyon       ex:plane    ex:Amman ⟩,
}
```

To evaluate Q over G, we can calculate the closure of G to have:

```
{
   ⟨ex:train          sp          ex:transport⟩,
   ⟨ex:plane          sp          ex:transport⟩,
   ⟨ex:Grenoble     ex:train      ex:Lyon⟩,
   ⟨ex:Lyon         ex:train      ex:Paris⟩,
   ⟨ex:Lyon         ex:plane      ex:Amman⟩,
   ⟨ex:Grenoble   ex:transport    ex:Lyon⟩,
   ⟨ex:Lyon       ex:transport    ex:Paris⟩,
   ⟨ex:Lyon       ex:transport    ex:Amman⟩
}
```

and then evaluate Q over the closure graph. In this case, we have the following answers:

?City
ex:Paris
ex:Amman

Another method consists of rewriting Q using the rules introduced in the previous subsection. In this respect, the transformed query Q' of Q is:

```
SELECT ?City
WHERE {
        ?p sp* ex:transport .
        ex:Grenoble (?p)+ ?City .
      }
```

This way, we can match only paths of the same repeated edge labels. In the graph G, we can find `ex:Paris` but not `ex:Amman`.

A third approach, which we have adopted in the (C)PSPARQL query evaluator prototype, is to rewrite (C)PSPARQL queries into a semantically equivalent CPSPARQL query by introducing constraints in the traversed edges.

For example, the query Q can be transformed to:

```
SELECT ?City
WHERE {
        CONSTRAINT const1 [EDGE ?P]: { ?P sp* ex:transport . }
        ex:Grenoble (# % const1 %)+ ?City .
      }
```

where # is the symbol that can be used in regular expressions to match any term (anonymous or blank variable). It is followed by a constraint, which means that the matched symbol (predicate label) must be a subPropertyOf transport.

In the same way, the transformation can be applied to every property $p \notin \{$`sp`, `sc`, `type`, `dom`, `range`$\}$ occurring inside a given (constrained) regular expression in a (C)PSPARQL query.

For negated properties, the transformation depends on the semantics of the negation operator over RDFS semantics. Indeed, there are two possible directions that can be exhibited. Let us illustrate them using the following RDFS graph:

```
{
   ⟨ex:trainTGV       sp        ex:train⟩,
   ⟨ex:plane          sp        ex:transport⟩,
   ⟨ex:train          sp        ex:transport⟩,
   ⟨ex:Paris       ex:plane     ex:Lyon⟩,
   ⟨ex:Lyon       ex:trainTGV   ex:Grenoble⟩,
   ⟨ex:Lyon        ex:plane     ex:Amman⟩
}
```

and the following query:

```
SELECT ?City
WHERE { ex:Paris (!ex:train)+ ?City .}
```

If we interpret `!ex:train` by the existence of a property other than `ex:train` or even any of its sub-properties, then the query can be transformed to:

```
SELECT ?City
WHERE {
        CONSTRAINT const1 [EDGE ?P]: { ?P (!sp)* ex:train . }
        ex:Paris (# % const1 %)+ ?City .
      }
```

In this case, `ex:Grenoble` is not an answer to the query. However, if we interpret (`!ex:train`) by the existence of a property only other than `ex:train` (this time a sub-property like `ex:trainTGV` can satisfy the expression), then the query can be simply transformed to:

```
SELECT ?City
WHERE {
        ex:Paris (?Mean)+ ?City .
        FILTER (?Mean != ex:train) .
      }
```

`ex:Grenoble` this time is an answer to the query.

This way, the following examples of expressive patterns given in [Arenas *et al.*, 2008]:

```
(next::[(next::sp)*./self::transport])+
```

and

```
(next::[(next::sp)*./self::transport])+/
    self::[(next::[(next::sp)*./self::bus ])*./self::London] /
         (next::[(next::sp)*./self::transport])+
```

can be expressed in CPSPARQL by:

```
SELECT ?City1 ?City2
WHERE {
        CONSTRAINT const1 [EDGE ?P]: { ?P sp* transport . }
        ?City1 (# % const1 %)+ ?City2 .
      }
```

and

```
SELECT ?City1 ?City2
WHERE {
        CONSTRAINT const1 [EDGE ?P]: { ?P sp* transport . }
        CONSTRAINT const2 [EDGE ?P2]: { ?P2 sp* bus . }
        ?City1 (# % const1 %)+ ?StopCity .
        ?StopCity (# % const2 %)* Lonon .
        ?StopCity (# % const1 %)+ ?City2 .
      }
```

8.3.3 Example: putting all together

Consider an RDF(S) graph that contains information about researchers like the following one:

```
{
   ⟨ex:Person1    foaf:name    "Faisal Alkhateeb" ⟩,
   ⟨ex:Person1     ex:topic    "Query Languages" ⟩,
   ⟨ex:Person1     rdf:type    ex:PhdResearcher ⟩,
   ⟨ex:Person1   ex:worksWith  ex:Person2 ⟩,
   ⟨ex:Person2     rdf:type    ex:Researcher ⟩,
   ⟨ex:Person2    foaf:name    "Jérôme Euzenat" ⟩,
   ⟨ex:works          sp       foaf:knows ⟩,
   ...
}
```

Then the following CPSPARQL query:

```
SELECT ?Person1
WHERE {
        CONSTRAINT const1 [EDGE ?P:] { ?P sp+ ex:knows . }

        { ?Person1 rdf:type.sc* ex:Researcher . }
           UNION
        { ?Person1 ?P1 ?Y .
          ?P1 sp* ?P2 .
          ?P2 dom.sc* ex:Researcher .
        }
           UNION
        { ?Y ?P1 ?Person1 .
          ?P1 sp* ?P2 .
          ?P2 range.sc* ex:Researcher .
        }
        ?Person1 (# % const1 %)+ ?Person2 .
        ?Person2 foaf:name "Jérôme Euzenat" .
      }
```

could be used to find researchers knowing "Jérôme Euzenat".

8.4 Conclusion

We have presented in this chapter several approaches that can be used for querying RDF(S) graphs with SPARQL queries. Each of them has its advantages and disadvantages which may limit their usage. We have also presented our own approach which essentially relies on rewriting a given SPARQL query into a PSPARQL query.

Implementation and Experiments

9

Contents

9.1 Implementation 131
9.1.1 Graph representation of RDF data model 132
9.1.2 Input and output data 133
9.1.3 Query evaluation algorithm 133
9.2 Experiments 134
9.2.1 Conformance test 134
9.2.2 Run time test 134
9.2.3 RDFS test 139
9.2.4 Hardness test 142
9.2.5 Preliminary SwetoDBLP test 147
9.3 Conclusion 148

Introduction

This chapter consists of two sections: Section 9.1 describes a concrete implementation of the (C)PSPARQL query language, and Section 9.2 provides the experimental results of this implementation.

9.1 Implementation

A CPSPARQL prototype has been implemented in Java[1]. We choose Java since it is an expressive programming language suitable for expressing data as object-oriented and platform-independent programming paradigms. The prototype is a

[1] `http://psparql.inrialpes.fr/`

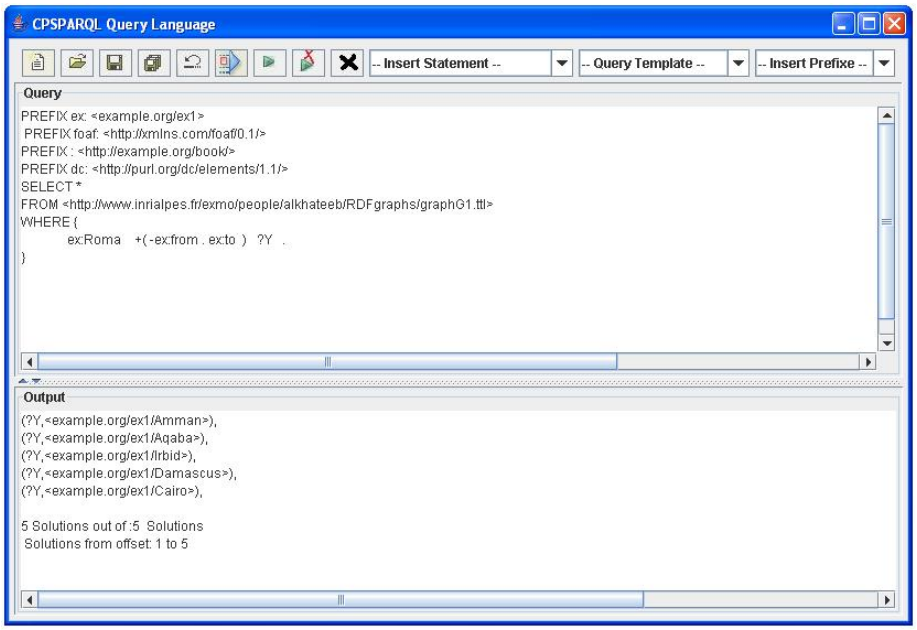

Figure 9.1: The CPSPARQL query evaluator interface.

query engine that mainly parses a text query; loads the RDF graph(s) to be queried, which are identified in the query using urirefs; and then evaluates the query providing the answers of the query. We answer the following questions in the subsequent subsections: how do we represent the RDF data model? What are the input and the output data and their types? And what are the evaluation algorithm(s)?

9.1.1 Graph representation of RDF data model

The prototype represents the RDF data model as a directed labeled graph by providing the following abstract interfaces and their implementations:

1. Nodes of the RDF data model represented by a class. Each node is an instance of that class, indexed by an identifier (a number), has a string name used for representing the URI (Uniform Resource Identifier) or the literal associated to it in the graph, an out-vector pointing to the set of output edges from this node in the graph, and possibly an in-vector pointing to the set of input edges to this node in the graph.

2. Edges are represented by a class. Each edge is an instance of that class, has an identifier (a number), has a label for storing the predicate label of an

RDF triple, and a vector containing tow elements pointing to the subject and object nodes of an RDF triple, respectively.

3. An RDF graph is represented as a directed labeled graph that contains a vector of nodes and a vector of edges. The vector of nodes contains all elements appearing as subjects and objects in the RDF data model such that each element is represented by a node whose name is the label of that element. Each triple is represented by an edge whose label is the predicate of the triple and the end-points of the edge are the two nodes associated to the subject and the object of the triple, and the out-vector of the subject node (respectively, the in-vector of the object node) points to the edge associated to that triple.

9.1.2 Input and output data

The prototype contains a class that can be used for evaluating CPSPARQL queries. The input to this class is a text query. This query will be then passed to the query parser that extracts the locations of the RDF data model, local files or web sources using URIs; passes these locations to the RDF data model parser that loads RDF documents which must be written in the Turtle language [Beckett, 2006], and then parses these documents to extracts from them the RDF triples. These RDF triples will be sent to another class to construct the RDF dataset that contains a default graph and a set of named graphs. The RDF dataset will be then returned to the query parser that continue parsing the text query to evaluate its graph patterns over the designated graph (*i.e.,* the active graph as it is called in [Prud'hommeaux and Seaborne, 2008]). The output of the parser is a text string which forms the result of the query evaluation.

9.1.3 Query evaluation algorithm

The query evaluation algorithm of the prototype is based upon the semantics of the language described in Section 5.2 following the evaluation semantic of SPARQL [Prud'hommeaux and Seaborne, 2008]. It has two main algorithms: one concerns evaluating (C)PRDF graphs (*i.e.,* computing (C)PRDF-GRDF entailment), which follows the backtrack algorithm presented in Section 5.4.2, and the other one is a rewriting algorithm that implements the rules described in Section 8.3.

9.2 Experiments

9.2.1 Conformance test

The prototype passes all test cases designed by DAWG (Data Access Working Group) for the SPARQL query language[2] except the ones that concern the DESCRIBE query format. The prototype is currently under experiment for the recently proposed test suite[3].

9.2.2 Run time test

We have tested the performance of the CPSPARQL prototype on a Dell machine with Bi-processor Xeon 5050 3GHz and 4GB of RAM. Java 1.5.0_07 has been used, and assigned 976 MB of RAM. We have run the test using several queries against different RDF graph sizes from {5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000} triples. We have repeated the tests 50 times for each graph size, and the average time is taken.

RDF graphs. The RDF graphs are constructed randomly with different sizes using a random graph generator. To have a connected graph and to test queries containing path expressions, nodes of the graphs are selected from 800 distinct nodes representing cities around the world and edges are selected from 4 distinct edge labels namely `{train, plane, bus, taxi}`. The average in and out degrees ($in\text{–}d$ and $out\text{–}d$) are calculated in function of the graph size, $in\text{–}d = out\text{–}d = \sqrt[2]{n}$, where n is the required number of edges. These settings increase the opportunity of having paths between cities with the same label, and also cycles.

Test 1. The first test is executed on a query without path expressions, and the time is taken between the beginning and return of the query answers. We observed that the time after a particular graph size has a stable state as shown in Figure 9.2. This observation may be justified by the time required to initial settings.

Test 2. In this test, the time is taken between the beginning and return of the first query answer as given in Figure 9.3. If we compare the time required for answering the given query and that required for providing the first solution, we can see that there exists a large difference between them.

Test 3. We have executed in this test a query containing a path expression with the positive closure `SELECT * WHERE {s p+ ?o }`, where `s` is a node selected randomly and `p` is selected from the edge labels. The positive closure is chosen

[2] `http://www.w3.org/2001/sw/DataAccess/tests/`
[3] `http://www.w3.org/2001/sw/DataAccess/tests/r2`

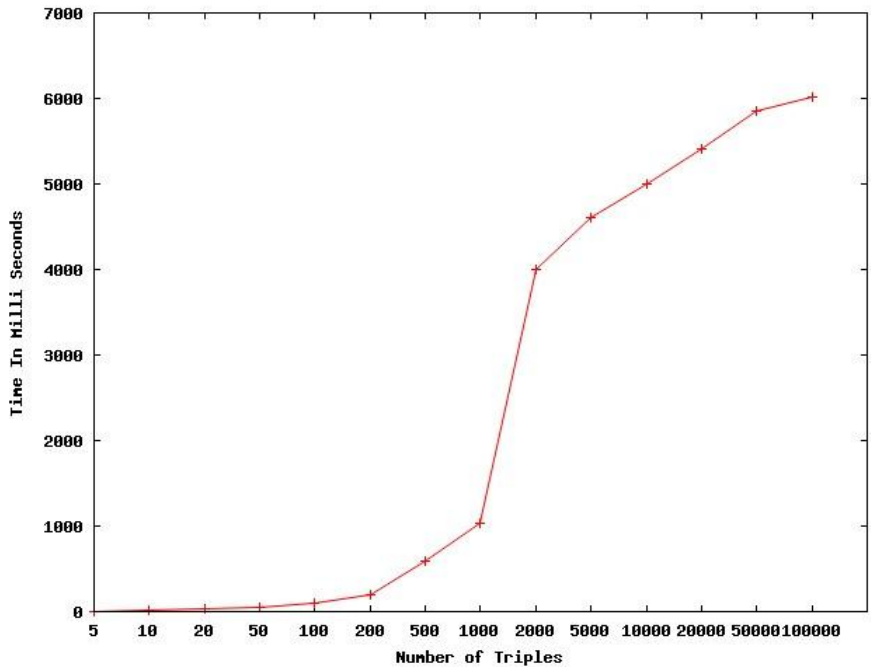

Figure 9.2: Time for answering query: `SELECT * WHERE {?s ?p ?o }`.

since it takes the longest time. Though the execution time as shown in Figure 9.4 followed an exponential growth, it does not exceed 7 seconds for the largest graph size.

Test 4. The goal of this test is to observe the effects of constraints on performances. For that, we have constructed randomly an RDF graph similar to one in Figure 6.1, *i.e.,* a graph containing only the following three kinds of triples ⟨`?newVar`, `ex:from`, `C1`⟩, ⟨`?newVar`, `rdf:type`, `transport`⟩, and ⟨`?newVar`, `ex:to`, `C2`⟩, where `transport` is one of the following transportation means {train, plane, bus, taxi}. We have executed in this test the following two CPSPARQL queries, Q_1 containing a constrained regular expression and Q_2 with a regular expression (without a constraint):

```
SELECT *
WHERE {
       CONSTRAINT const1 ]ALL ?Trip]:
                                { ?Trip rdf:type ex:Plane . }
       ?s (ex:from-%const1%.ex:to)+ ?o
      }
```

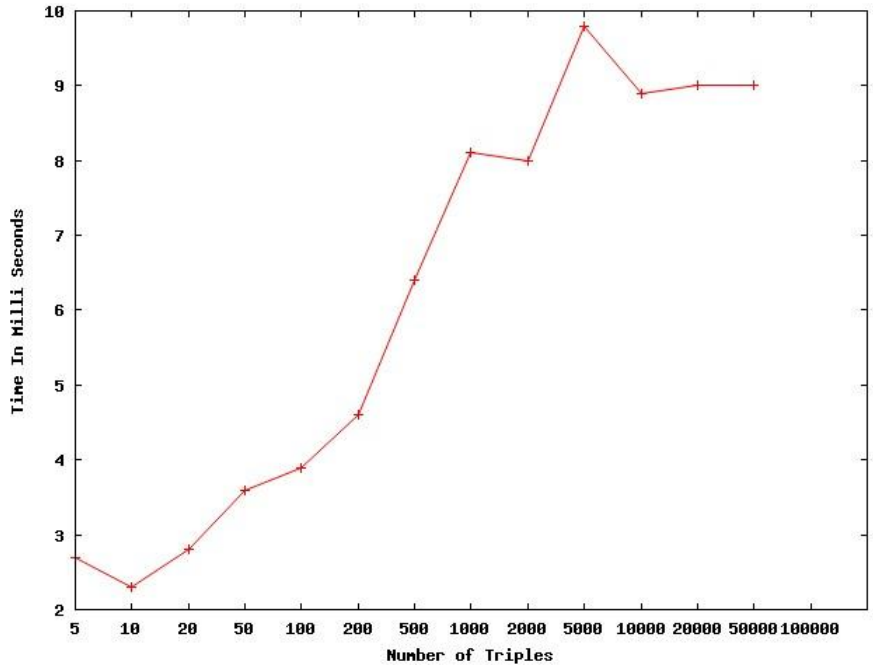

Figure 9.3: Time for finding the first solution for query: `SELECT * WHERE {?s ?p ?o }`.

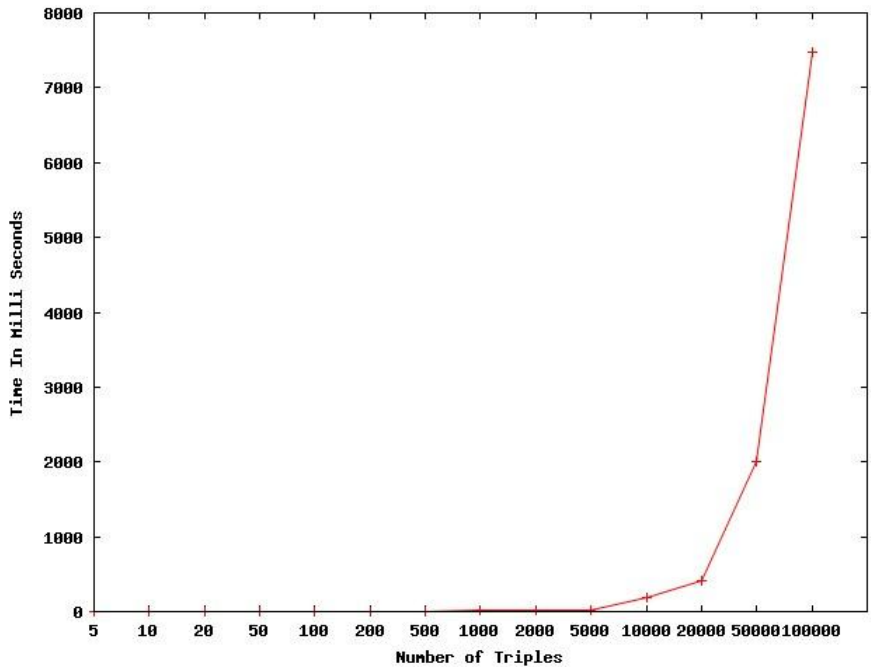

Figure 9.4: Time for answering query: `SELECT * WHERE {s p+ ?o }`.

n =	**5**	**10**	**20**	**50**	**100**	**200**	**500**	**1000**	**2000**
Q_1	0	1	1	6	9	13	44	134	402
Q_2	2	4	9	15	36	90	293	2120	5018

Table 9.1: Average number of answers for Q_1 and Q_2.

and

```
SELECT *
WHERE {
        ?s (ex:from- . ex:to)+ ?o .
       }
```

As shown in Figure 9.5, the time for the query with constrained regular expression is better than that of the query without it. This shows that our query evaluator takes advantage of the constraints for cutting the search space during evaluation as it does not explore paths that cannot lead to a solution. Table 9.1 shows some of the average number of answers of each query (*i.e.,* Q_1 and Q_2) for selected graph sizes. There exists a large difference between the two expected answers (or paths).

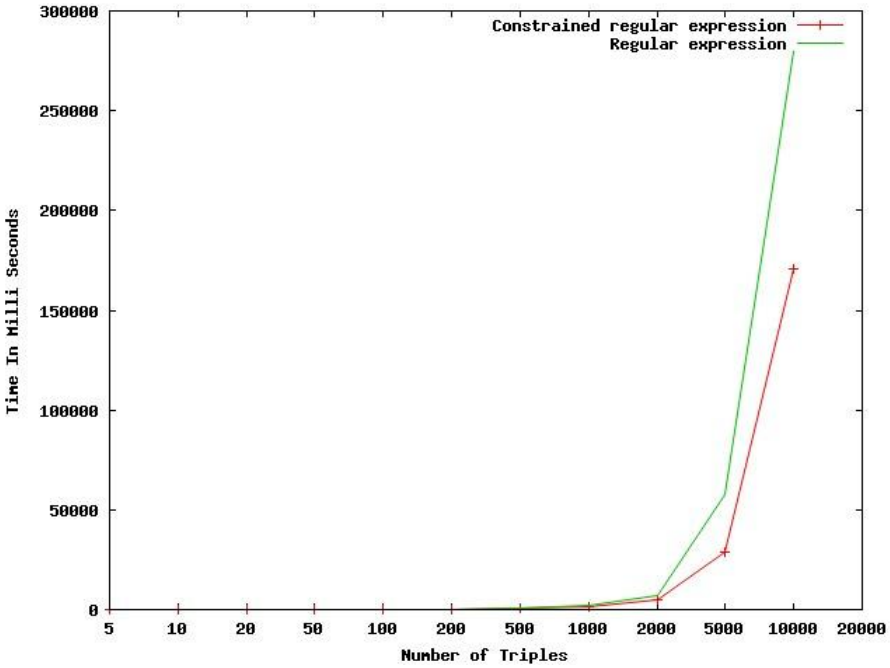

Figure 9.5: Time for answering a CPSPARQL query with and without constrained regular expressions.

9.2.3 RDFS test

This test, like all the remaining ones, is performed on a Sony machine with Intel Core TM 2 Duo Processor T7100 1.80GHz and 2GB of RAM. Java 1.5.0_07 has been used, and assigned 900 MB of RAM. Its goal is to observe the behavior of the prototype for querying RDFS graphs. More precisely, we want to compare, given a query Q, the time required to evaluate Q and its transformed query Q' (see Section 8.3). However, this test uses a specified RDF schema (the Univ-Bench ontology), instead of having a model of what are the difficult RDF schemas in this context or which RDF schema are realistic [Theoharis *et al.*, 2008].

RDFS graphs. We have used the Lehigh University Benchmark[4] [Guo *et al.*, 2005] for generating the RDF graphs. This tool generates RDF, OWL or DAML+OIL data over the Univ-Bench ontology. We have divided this schema into different sizes[5] {100, 200, 300} triples. We have then used these schemas to generate different RDFS graph sizes[6] {200, 500, 1000, 2000, 5000, 10000} triples. We have run the tests using several queries (involving RDFS vocabularies) against the generated graphs together with the RDF schemas.

Test 1. This test is executed on a query Q_1 that searches all superclasses to which a randomly selected class `s` belongs. Figures 9.6 and 9.7 shows the times required for evaluating the query and its transformed one, respectively.

Test 2. In this test, we have executed a query Q_2 that searches all object resources connected by a repeated property (*i.e.,* a randomly selected property not in the RDFS vocabulary) to a subject resource (randomly selected). Note that according to the transformation system presented in Chapter 8, the transformed query of:

```
SELECT *
WHERE {s p+ ?o }
```

is the following one:

```
SELECT *
WHERE {
   CONSTRAINT const1 [EDGE ?prop ]:  { ?prop sp+ p . }
   s ( # % const1 % )+ ?o .
}
```

[4] http://swat.cse.lehigh.edu/projects/lubm/

[5] http://www.inrialpes.fr/exmo/people/alkhateeb/RDFgraphs/rdfsSchemas/

[6] http://www.inrialpes.fr/exmo/people/alkhateeb/RDFgraphs/rdfsTestbeds/

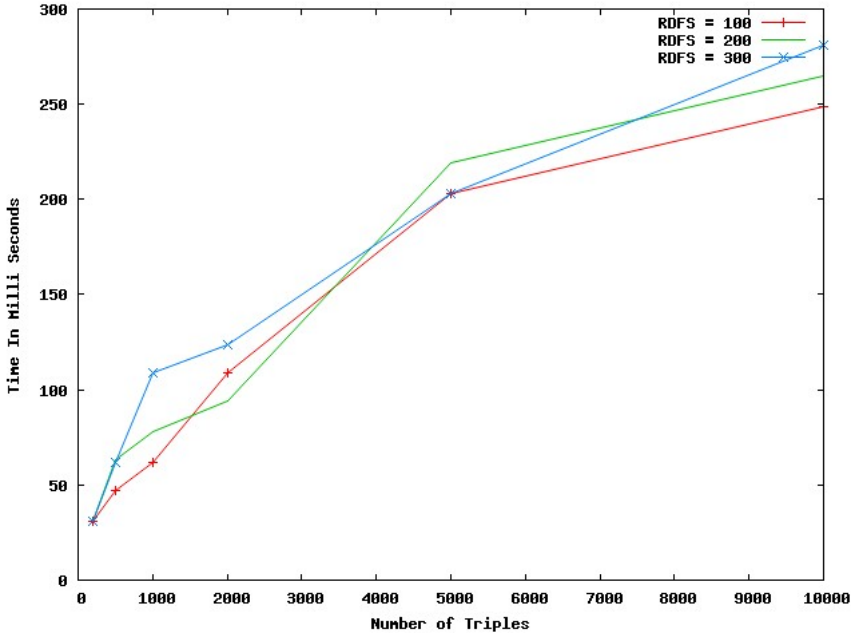

Figure 9.6: Time for answering query: `SELECT * WHERE {s sc ?o }`

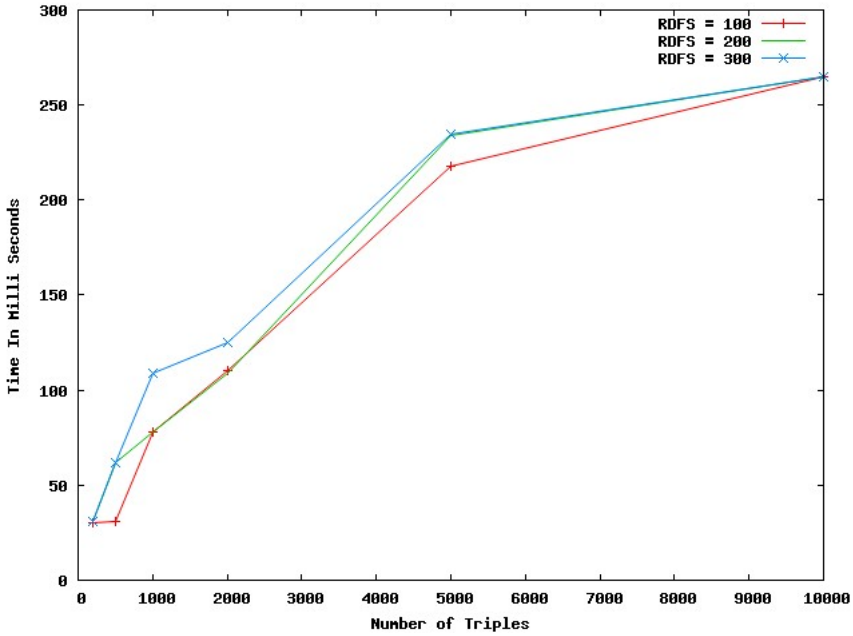

Figure 9.7: Time for answering query: `SELECT * WHERE {s sc+ ?o }`

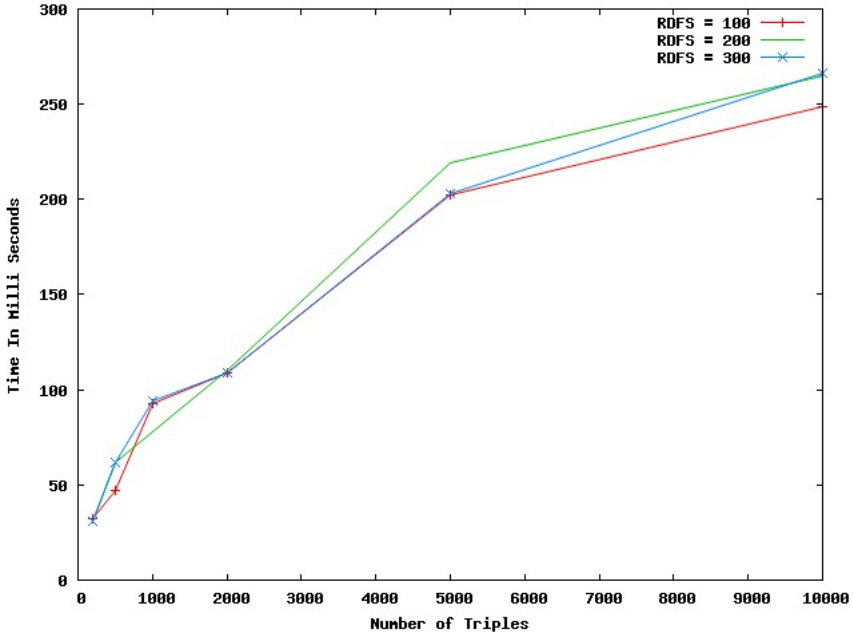

Figure 9.8: Time for answering query: `SELECT * WHERE {s p+ ?o }`

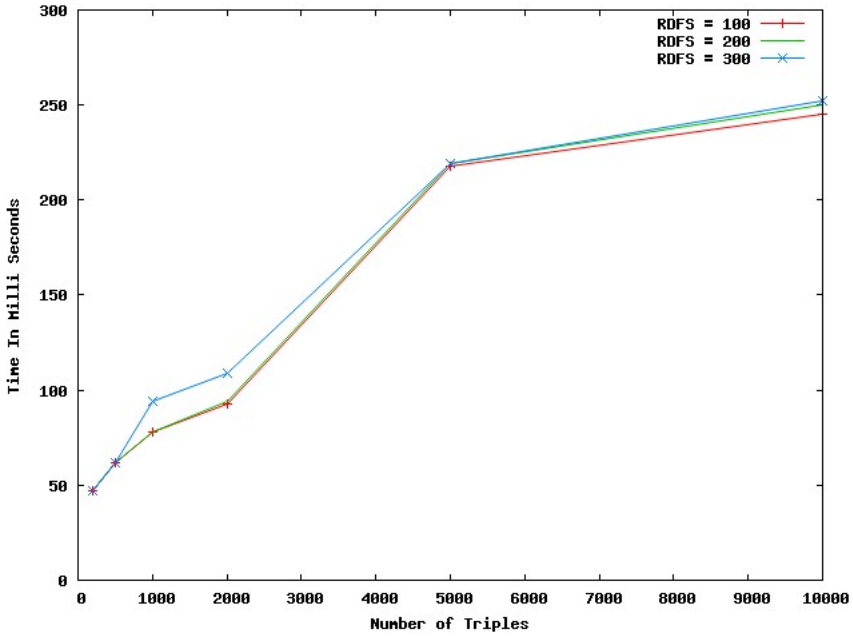

Figure 9.9: Time for answering query: `SELECT * WHERE {s` τ`(p+) ?o }`

Figures 9.8 and 9.9 shows the times required for answering the above two queries, respectively.

Test 3. We have executed in this test a query Q_3 containing the `type` relation. As shown in Figures 9.10 and 9.11, there rather exist a gap between the time reserved for answering this query and the previous ones. This is expected since the patterns in the transformed query is much larger in size and number than the original one. In addition, we have used a naive transformation, *i.e.,* the transformed query is:

```
SELECT *
WHERE {
        { s type.sc* ?o . }
        UNION
        { s ?p1 ?y . ?p1 sp* ?p2 . ?p2 domain.sc* ?o . }
        UNION
        { ?y ?p1 s . ?p1 sp* ?p2 . ?p2 range.sc* ?o . }
      }
```

However, if we look at the transformed query, we see that there exists repeated patterns (*e.g.* `?p1 sp* ?p2` and `.sc* ?o`). We think that the evaluation of this query can be optimized in two ways: either by calculating the answers to these patterns only once or using the typing system used in RQL [Karvounarakis *et al.*, 2002].

Figure 9.12 shows the time required for evaluating only the first sub-pattern of the above query.

The following table presents the average number of answers for the above queries and their transformed queries.

query	average number of answers
Q_1	1
Q'_1	3
Q_2	4
Q'_2	6
Q_3	2
Q'_3	4
Q''_3	5

9.2.4 Hardness test

In this test, we have varied the average in- and out-degrees of the graph to be tested, where in-out degree of 25 means that the average number of in-coming and out-coming edges to nodes in the graph is 25. RDF graphs have been constructed using

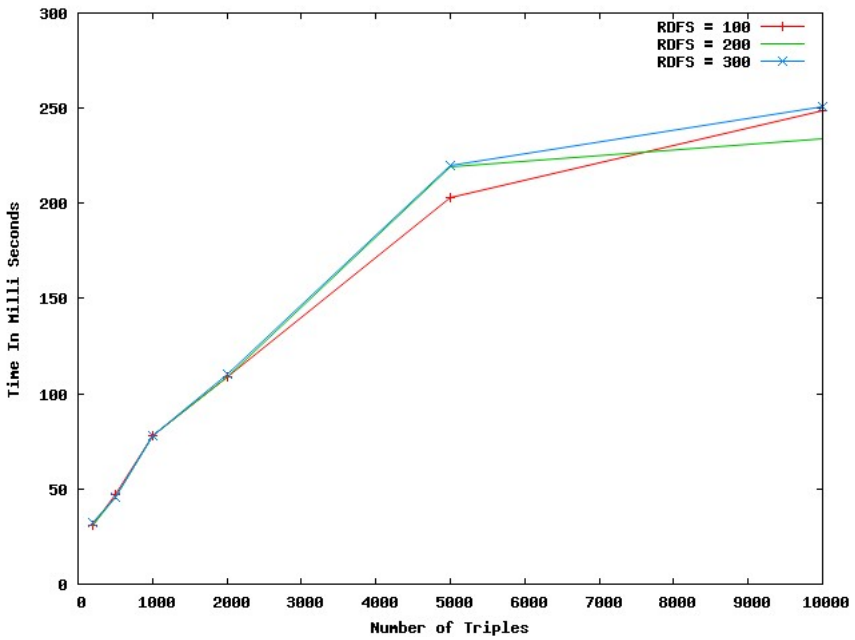

Figure 9.10: Time for answering query: `SELECT * WHERE {s type ?o }`

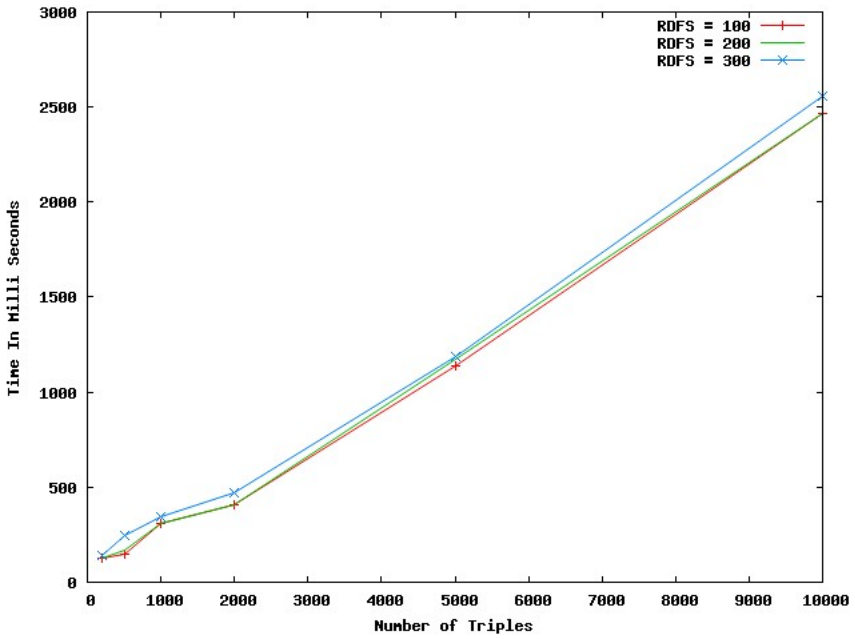

Figure 9.11: Time for answering query: `SELECT * WHERE { s` τ`(type) ?o }`

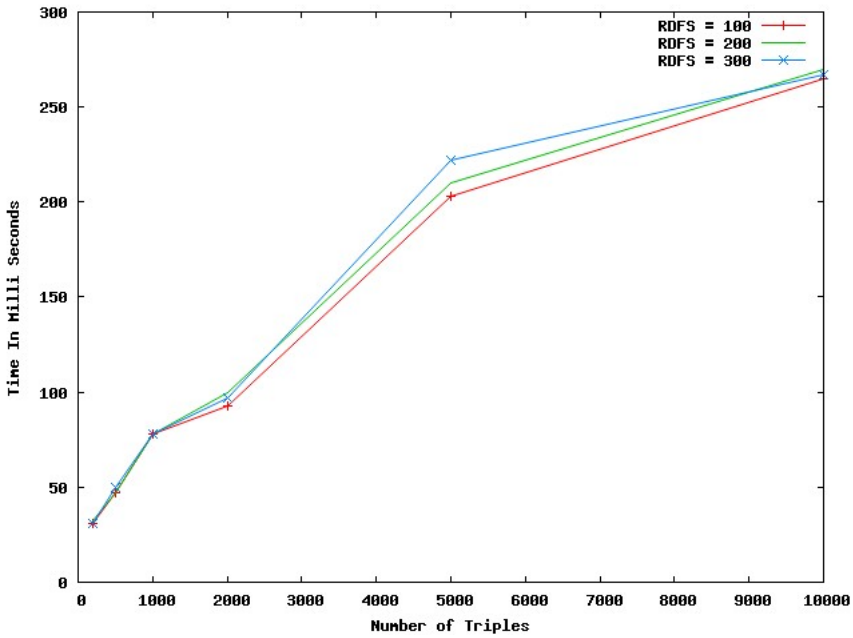

Figure 9.12: Time for answering query: `SELECT * WHERE {s type.sc* ?o }`

a random graph generator[7] with in-degrees equal to out-degrees. We have executed the following two queries (Q_1 and Q_2, respectively) 50 times and measured the average time and the average number of answers for each one. Each time, `s` and `p` are selected randomly from the node and the edge labels, respectively.

```
SELECT *
WHERE {
        s p ?o .
      }
```

Q_1 that searches all nodes connected to a selected resource node.

```
SELECT *
WHERE {
        s p+ ?o .
      }
```

Q_2 that searches all nodes connected by a path of a repeated property to a resource node.

[7] `http://www.inrialpes.fr/exmo/people/alkhateeb/RDFgraphs/randomGraphGenerator/`

Test 1. In this test, we have constructed RDF graphs with fixed number of nodes. In this case, the total number of edges is determined by the in- and out-degrees, *i.e.*, $\#edges = \#nodes * in{-}d$ (where $in{-}d = out{-}d$). Moreover, edge labels are selected randomly from `{train, plane, bus, taxi}`.

Figure 9.13 shows the average time required for answering Q_1 and Q_2 over an RDF graph with 1000 nodes and different in-out degrees.

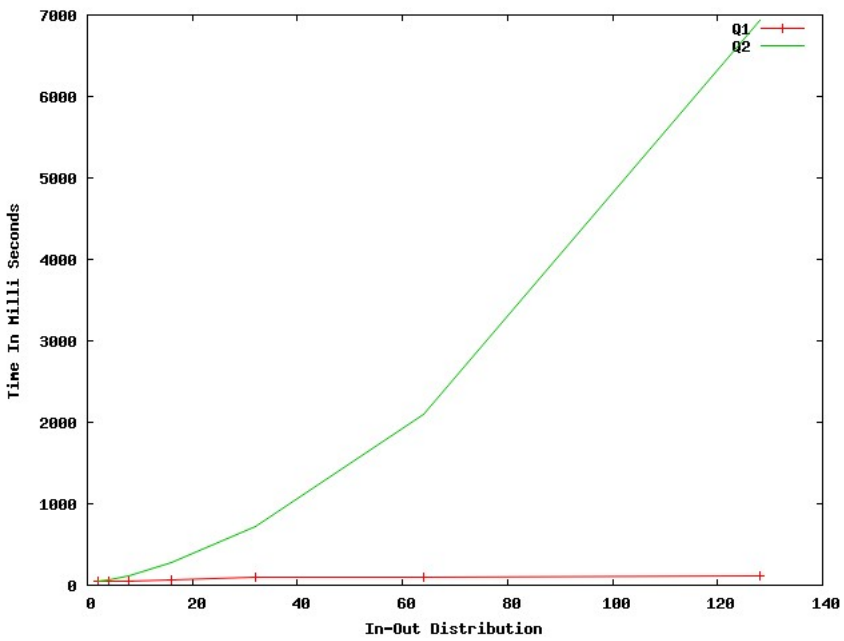

Figure 9.13: Time for answering Q_1 and Q_2 with different in-out degrees and fixed number of nodes.

Table 9.2 provides the average number of answers for Q_1 and Q_2 over an RDF graph with number of nodes equal to 1000, where columns represent the in-degree values. As expected, the higher the in-out degree, the more number of answers the two queries have.

in-out degree	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Q_1	0	1	2	5	8	16	32
Q_2	1	30	645	980	998	1000	1000

Table 9.2: Average number of answers for Q_1 and Q_2.

Test 2. In this test, we have constructed RDF graphs with fixed sizes, *i.e.,* the number of edges is fixed. The number of nodes depends on the in- and out-degrees, *i.e.,* $\#nodes = graphSize/in{-}d$ (where $in{-}d = out{-}d$, and $graphSize$ denotes the number of required edges). Moreover, edge labels are selected randomly from `{train, plane, bus, taxi}`.

Figure 9.13 shows the average time required for answering Q_1 and Q_2 over an RDF graph with 10000 edges and different in-out degrees.

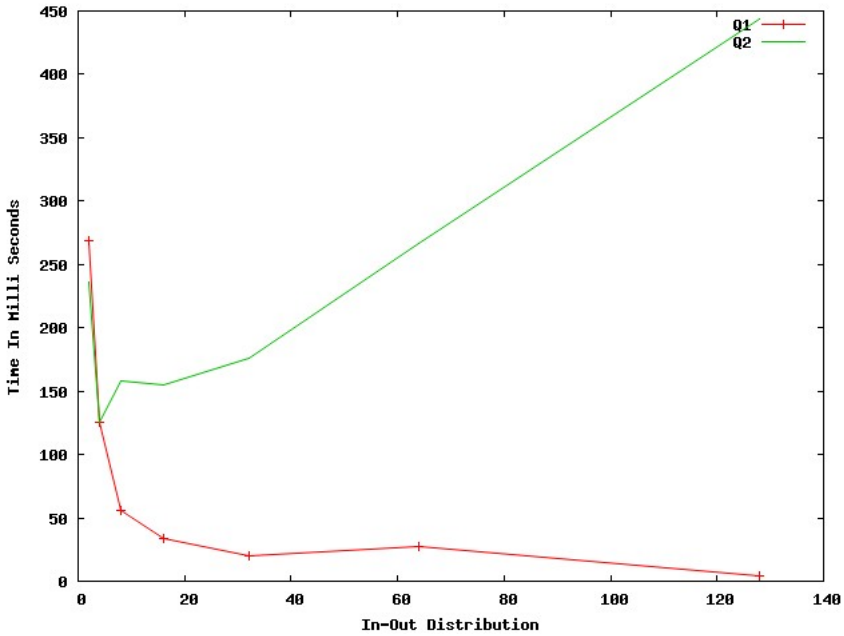

Figure 9.14: Time for answering Q_1 and Q_2 over an RDF graph with different in-out degrees and fixed number of edges.

Table 9.3 provides the average number of answers for Q_1 and Q_2 over an RDF graph with size 10000.

in-out degree	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Q_1	0	1	1	2	4	6	12
Q_2	1	37	945	612	312	156	78

Table 9.3: Average number of answers for Q_1 and Q_2.

Although the number of answers of Q_2 for the in-out degree 128 is less than the other degrees, the required time is greater than the others. This is because the

search space became larger (respectively, the number of nodes smaller) than for the previous degrees.

9.2.5 Preliminary SwetoDBLP test

In this test, we have used a randomly selected part (swetodblp_april_2008_part_1) of the SwetoDBLP[8]. We have further divided this part into several sub-parts [9].

	Number of edges	Number of nodes
sub-parts 1 and 2	167831	107919
sub-parts 3 and 4	130821	85251

We have executed two kinds of queries that mainly search paths from a randomly selected resource node. The first kind Q_1 contains only a path variable connecting a resource and a variable destination nodes, which is similar to the following one:

```
SELECT *
WHERE {
        <http://dblp.uni-trier.de/.../MatskinH95> ??pathVar ?o .
      }
```

The second kind of queries Q_2 consists of searching paths from a selected node with a predefined regular expression such as:

```
PREFIX opus: <http://lsdis.cs.uga.edu/projects/semdis/opus#>
SELECT *
WHERE {
        DEFINED BY ??pathVar <http://dblp.uni-trier.de/.../MatskinH95>
                                (opus:cites $|$ opus: isIncludedIn)+ ?o
        <http://dblp.uni-trier.de/.../MatskinH95> ??pathVar ?o .
      }
```

We have repeated the tests 50 times for each kind of queries and measured the average running time excluding the load time. Resource nodes are selected randomly at each time for the two query kinds.

Table 9.4 summarizes the average time for the queries over sub-parts 1 and 2, and sub-parts 3 and 4. As shown in the table, the time does not exceed 12 seconds in the worst case. Moreover, for the queries with predefined regular expressions, it is always less than that for those with only path variable. As shown Table 9.5, the average number of answers for the first query is greater than that for the second kind.

	Q_1	Q_2
sub-parts 1 and 2	11 seconds	8 seconds
sub-parts 3 and 4	12 seconds	9 seconds

Table 9.4: Average time for queries over different sub-parts of part-1 of swetoDBLP.

	Q_1	Q_2
sub-parts 1 and 2	96	66
sub-parts 1 and 2	120	70

Table 9.5: The average number of answers for the two kinds of queries queries.

It should be noted that the number of answers (or matched paths) does not denote the number of investigated paths. Moreover, we do not need to investigate each individual path to find answers, but instead each possible map (see for example the definition of homomorphisms in Chapter 4).

We have identified the following resource node with a large number of answers: `<http://dblp.uni-trier.de/rec/bibtex/books/cs/UllmanH89>`.

	Q_1	Q_2
Number of answers	576	538

9.3 Conclusion

We have presented a first implementation of a (C)PSPARQL query evaluator which does not use any optimization technique.This prototype has demonstrated its ability to evaluate (C)PSPARQL queries as well as regular SPARQL queries through the compliance tests provided by W3C. This naive implementation has experimentally exhibited a reasonable behavior at this preliminary stage.We showed some advantages of using CPSPARQL, in particular in involving constraints during the path search (and not a posteriori), since this clearly avoid exploring a too large part of the search space. Finally, we have tested various extensions to (C)PSPARQL, in particular for dealing with restricted RDFS ontologies. Our prototype is freely available in source code, as well as all the test generators so that others can use them to test it further.

[8] `http://lsdis.cs.uga.edu/projects/semdis/swetodblp/`

[9] `http://www.inrialpes.fr/exmo/people/alkhateeb/RDFgraphs/swetoPart1-subparts/`

Conclusion 10

Contents

10.1 Summary **149**

10.2 Future Directions **150**

10.2.1 Using query languages for XML document generation 151

10.2.2 Processing alignment with query languages 151

10.2.3 Query Answering in distributed environments 152

10.2.4 Optimization, indexing and storage mechanisms . . . 152

10.1 Summary

This thesis addresses the problem of supporting path expressions and path extractions in semantic web knowledge bases. As we mentioned in Chapter 1, the current query languages for the semantic web either rely on the relational algebra which lack the possibility of expressing recursive queries or are purely path-based languages which support limited forms of path traversals mechanisms and have no support for conjunctive queries and SQL-like functionalities.

Our study is therefore motivated by the need of developing a compromised language that supports both querying paradigms. Though the study can be made to other formalisms, it is applied in the context of the RDF(S) and its data model as a directed labeled graphs presented in Chapter 2. Chapter 3 discussed the current querying paradigms and highlighted the differences between them and our proposal.

Our contributions consist of three main parts:

- We have presented in Chapter 4 a general graph model, called PRDF, supporting path expressions in RDF knowledge bases. The originality of this model is its generality which is argued by the fact that the demonstration framework (including semantics, algorithms as well as the completeness results) still works with any mean used to generate regular languages to be instantiated to this model. However, as it is outlined in the thesis, the complexity will depend on the path expressions used to instantiate the model.

- Since SPARQL is expected to gain popularity as the official query language for RDF, we have made our choice to avoid reinventing a query language and benefit from the existing standards. So, we have instantiated the PRDF graph model to regular expressions providing a novel extension to SPARQL, called PSPARQL in Chapter 5. We have provided its syntax, its semantics as well as algorithms for evaluating PSPARQL queries over simple RDF graphs. The originality of our algorithms is their soundness and completeness with respect to RDF semantics, and the hidden reasoning algorithm (*i.e.,* based on a rewriting method) for querying RDF(S) graphs (including this time RDF and RDFS vocabularies) which is a missing piece of SPARQL.

- PSPARQL was the basis for developing a new extension, called CPSPARQL, that further allows other constructs in SPARQL such as constraints on internal nodes and edges on traversed paths. As discussed in Chapter 6, this extension provided several advantages, among them, it adds expressivity to (P)SPARQL and enhances the efficiency using predefined constraints that prune on-the-fly irrelevant paths.

The implementation of our extensions together with the empirical study including several tests given in Chapter 9 (such as the compatibility tests using SPARQL test cases provided by the Data Access Working Group of SPARQL, practical tests and others) shows the expressive power and the efficiency of our prototype with respect to other languages.

10.2 Future Directions

Our future work will regard several directions discussed in the following subsequent sections.

10.2.1 Using query languages for XML document generation

In this direction, we aim to bridge the gap between XML and Semantic Web technologies in the context of document generation. In particular, we use SPARQL query language for generating XML documents from queried RDF data [Alkhateeb and Laborie, 2008]. Additionally, SPARQL queries can be embedded in a form of XML templates to allow constructing missed information from the query answers. A future work in this direction consists in controlling the number of generated documents by permitting, for example, the user to interact with the system. This way she/he can select desired answers of a query to be composed with other query answers. Another issue concerns semantic preservation of imported templates (*e.g.* preserving the urirefs) as well as studying the possibility to add some control on the importation of a template.

10.2.2 Processing alignment with query languages

Problems raised by heterogeneous ontologies can be solved by establishing correspondences between entities of these ontologies and processing the resulting alignment for data transformation. The use of query languages as suggested in [Euzenat *et al.*, 2008] for data transformation would be a natural choice since they allow data extraction and transformation. SPARQL is a good candidate for that purpose, in particular, when ontologies are described in RDF(S) and OWL. However, there are missing pieces of SPARQL like aggregate functions, value-generating and paths. The integration of the two proposed languages, namely SPARQL++ and CPSPARQL, provides queries which are sufficient for covering expressive alignment languages (*e.g.* [Euzenat *et al.*, 2007]). For example, the following CPSPARQL query:

```
CONSTRUCT { ?x o2:potentialCollaborator ?y . }
WHERE { ?x foaf:knows+ ?y.
        ?x o1:topic ?t.
        ?y o1:topic ?t.
        ?x rdf:type o1:researcher .
        ?y rdf:type o1:researcher .
      }
```

could be used to create an ontology that contains the `potentialCollaborator` relation between two researchers expressed by the fact that one researcher is potentially collaborator to another one if they work on the same topic and know each other.

10.2.3 Query Answering in distributed environments

In this direction, we would like to benefit from the strong relation between conjunctive queries and SPARQL, and from our initial work on answering conjunctive queries in distributed environments [Alkhateeb and Zimmermann, 2007] for designing a distributed query evaluation infrastructure for supporting path queries in distributed environments. In this article, we have considered query answering over a distributed knowledge bases system and defined the distributed answers of a given query expressed in terms of one knowledge base or ontology (called the target ontology) in the system. Since answers to a SPARQL query are defined by constructing maps from GRDF graphs of the query into the knowledge base and consider a GRDF graph as a particular case of a conjunctive query, we can use the distributed answer definition to define answers to SPARQL queries.

10.2.4 Optimization, indexing and storage mechanisms

Firstly, we think that the task of evaluating queries involving path expressions is heavyweight and, despite the good timing results, our prototype needs to be optimized for practical use to be scaled over large RDF knowledge bases. For this direction, we will investigate several optimization techniques that can be applied to query and/or RDF knowledge bases including but not limited to the approach of [Diwan *et al.*, 1996] for clustering graphs to minimize external path length.

Secondly, our current implementation to the evaluation algorithms is based on the main memory, and we will investigate the possibility of developing an indexing mechanism for queries involving path expressions that can be used for efficient disk-based query evaluation.

Finally, we would also benefit from the current DBMSs to provide, for example, an underlying storage infrastructure for our implementation.

Bibliography

[Abiteboul *et al.*, 1997] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L.Wiener. The lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1):68–88, 1997.

[Abiteboul, 1997] Serge Abiteboul. Querying semi-structured data. In *Proceeding of the 6th International Conference on Database Theory (ICDT). Volume 1186 of LNCS., Springer-Verlag*, pages 1–18, 1997.

[Agrawal, 1988] Rakesh Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, 1988.

[Aho and Ullman, 1979] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL 1979)*, pages 110–119, New York, NY, USA, 1979. ACM.

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading (MA US), 1974.

[Aho, 1980] Alfred V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, New York (NY US), 1980.

[Alechina *et al.*, 2003] Natasha Alechina, Stéphane Demri, and Maarten de Rijke. A modal perspective on path constraints. *Journal of Logic and Computation*, 13:1–18, 2003.

[Alkhateeb and Laborie, 2008] Faisal Alkhateeb and Sébastien Laborie. Towards Extending and Using SPARQL for Modular Document Generation. In *Proceed-*

ings of The Eight ACM Symposium on Document Engineering (DocEng2008), 16-19 Sepember, São Polo (Brésil), 2008.

[Alkhateeb and Zimmermann, 2007] Faisal Alkhateeb and Antoine Zimmermann. Query Answering in Distributed Description Logics. In Houda Labiod and Mohamad Badra, editors, *New Technologies, Mobility and Security - Proceedings of NTMS'2007 Conference*, pages 523–534. Springer, may 2007.

[Alkhateeb *et al.*, 2005] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Complex path queries for RDF graphs. In *Poster proceedings of the 4th International Semantic Web Conference (ISWC'05), Galway (IE)*, 2005.

[Alkhateeb *et al.*, 2007] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. RDF with regular expressions. Research report 6191, INRIA, Montbonnot (FR), 2007.

[Alkhateeb *et al.*, 2008a] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Constrained regular expressions in SPARQL. In *Proceedings of the 2008 International Conference on Semantic Web and Web Services (SWWS'08), to appear*, 2008.

[Alkhateeb *et al.*, 2008b] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with Regular Expression Patterns (for Querying RDF). *to appear in Journal of Web Semantics*, 2008. Submitted in November 23, 2006.

[Alkhateeb, 2007] Faisal Alkhateeb. Une extension de RDF avec des expressions régulières. In *actes de 8e Rencontres Nationales des Jeunes Chercheurs en Inteligence Artificielle (RJCIA)*, pages 1–14, July 2007.

[Amann and Scholl, 1992] Bernd Amann and Michel Scholl. Gram: A graph data model and query language. In *Proceedings of European Conference on Hypertext (ECHT)*, pages 201–211, 1992.

[Angles and Gutierrez, 2008] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008.

[Angles and Gutiérrez, 1995] Renzo Angles and Claudio Gutiérrez. Querying RDF data from a graph database perspective. In *Proceedings of the European Semantic Web Conference (ESWC)*, pages 346–360, 1995.

[Anyanwu *et al.*, 2007] Kemafor Anyanwu, Angela Maduko, and Amit P. Sheth. SPARQ2L: towards support for subgraph extraction queries in RDF databases. In *Proceedings of the 16th international conference on World Wide Web (WWW'07)*, pages 797–806, 2007.

[Arenas *et al.*, 2008] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. An extension of sparql for rdfs. In *Post Proceedings of Joint SWDB and ODBIS workshop on Semantic Web, Ontologies, Databases SWDB-ODBIS'07. LNCS, vol. 5005*, pages 1–2, Vienna, Austria, September 2008.

[Baget, 2003] Jean-François Baget. Homomorphismes d'hypergraphes pour la subsomption en RDF. In *Proceedings of the 3e journées nationales sur les modèles de raisonnement (JNMR), Paris (France)*, pages 1–24, 2003.

[Baget, 2005] Jean-François Baget. RDF entailment as a graph homomorphism. In *Proceedings of the 4th International Semantic Web Conference (ISWC'05), Galway (IE)*, pages 82–96, 2005.

[Balcazar *et al.*, 1988] Jose Luis Balcazar, Josep Diaz, and Joaquim Gabarro. *Structural complexity 1*. Springer-Verlag, New York (NY, USA), 1988.

[Beckett, 2004] Dave Beckett. RDF/XML syntax specification (revised). W3C Recommendation, February 2004.

[Beckett, 2006] Dave Beckett. Turtle - terse RDF triple language. Technical report, Hewlett-Packard, Bristol (UK), 2006.

[Berners-Lee *et al.*, 2001] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web, 2001. `http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21`.

[Bray *et al.*, 2006] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0. Recommendation, W3C, August 2006. `http://www.w3.org/TR/REC-xml/`.

[Brickley and Guha, 2004] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, 2004. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[Broekstra, 2003] Jeen Broekstra. SeRQL: Sesame RDF query language. In *SWAP Delivrable 3.2 Method Design*, 2003.

[Buneman *et al.*, 1995] Peter Buneman, Susan B. Davidson, and Dan Suciu. Programming constructs for unstructured data. In *Proceedings of the 1995 International Workshop on Database Programming Languages*, page 12, 1995.

[Buneman *et al.*, 1996] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 505–516, 1996.

[Buneman, 1997] Peter Buneman. Semistructured data. In *Tutorial in Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.

[Calvanese *et al.*, 2000a] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, pages 176–185, 2000.

[Calvanese *et al.*, 2000b] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query processing for regular path queries with inverse. In *Proceedings of the Nineteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2000)*, pages 58–66, ACM Press, New York, 2000.

[Carroll and Klyne, 2004] Jeremy J. Carroll and Graham Klyne. RDF concepts and abstract syntax. W3C recommendation, W3C, February 2004.

[Clark and DeRose, 1999] James Clark and Steve DeRose. XML Path Language (XPath). W3C Recommendation, 1999. `http://www.w3.org/TR/xpath`.

[Clark, 1978] Keith L. Clark. Negation as failure. In *Logic and Data Bases (actes symposium on logic and databases, Toulouse (FR), 1977, Hervé Galaire, Jack Minker (éds.))*, pages 293–322, Plenum Press, New York, 1978.

[Codd, 1970] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Codd, 1983] Edgar F. Codd. A relational model of data for large shared data banks. 26(1):64–69, 1983.

[Consens and Mendelzon, 1990] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416, 1990.

[Corby and Faron-Zucker, 2007a] Olivier Corby and Catherine Faron-Zucker. Implementation of SPARQL Query Language Based on Graph Homomorphism. In *Proceedings of the 15th International Conference on Conceptual Structure (ICCS'07)*, pages 472–475, Sheffield, UK, 2007.

[Corby and Faron-Zucker, 2007b] Olivier Corby and Catherine Faron-Zucker. RDF/SPARQL design pattern for contextual metadata. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)*, pages 470–473, Washington, DC, USA, 2007.

[Corby *et al.*, 2000] Olivier Corby, Rose Dieng, and Cédric Hébert. A conceptual graph model for W3C resource description framework. In *Proceedings of the International Conference on Conceptual Structures*, pages 468–482, 2000.

[Corby *et al.*, 2004] Olivier Corby, Rose Dieng-Kuntz, and Catherine Faron-Zucker. Querying the Semantic web with Corese Search Engine. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'2004), sub-conference (PAIS'2004), Valencia (Spain)*, pages 705–709, 2004.

[Cruz *et al.*, 1987] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 323–330, New York, NY, USA, 1987.

[Cruz *et al.*, 1988] Isabel. F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. G+: Recursive queries without recursion. In *Proceedings of Second International Conference on Expert Database Systems*, pages 355–368, 1988.

[Cyganiak, 2005] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Labs, 2005. `http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html`.

[de Bruijn *et al.*, 2005] Jos de Bruijn, Enrico Franconi, and Sergio Tessaris. Logical reconstruction of normative RDF. In *International Workshop on OWL: Experiences and Directions (OWLED 2005), Galway, Ireland*, 2005.

[de Moor and David, 2003] O. de Moor and E. David. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1-2):15–35, 2003.

[Diwan *et al.*, 1996] Ajit A. Diwan, Sanjeeva Rane, Sridhar Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings of the 22th International Conference on Very Large Databases VLDB*, pages 342–353, 1996.

[Euzenat and Shvaiko, 2007] Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[Euzenat *et al.*, 2007] Jérôme Euzenat, François Scharffe, and Antoine Zimmermann. Expressive alignment language and implementation. Knowledge Web Network of Excellence. project Deliverable d2.2.10, 2007.

[Euzenat *et al.*, 2008] Jérôme Euzenat, Axel Polleres, and François Scharffe. Processing ontology alignments with SPARQL. In *Proceedings of the International Workshop on Ontology Alignment and Visualization OnAV'08*, 2008. To appear.

[Fernandez *et al.*, 1997] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A Query Language for a Web-Site Management System. *ACM SIGMOD Record*, 26(3):4–11, 1997.

[Florescu *et al.*, 1998] Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS '98)*, pages 139–148, New York, NY, USA, 1998.

[Franconi and Tessaris, 2005] Enrico Franconi and Sergio Tessaris. The semantics of SPARQL. W3C working draft, November 2005. `http://www.inf.unibz.it/krdb/w3c/sparql/`.

[Genevès *et al.*, 2007] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *Proceedings of PLDI'07*, pages 342–351, 2007.

[Golomb and Baumert, 1965] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the ACM*, 12(5):516–524, 1965.

[Gottlob *et al.*, 1999] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. In *Proceedings of the*

Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99), pages 394–399, 1999.

[Grahne and Thomo, 2003] Gösta Grahne and Alex Thomo. Query containment and rewriting using views for regular path queries under constraints. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS 2003)*, pages 111–122, New York, NY, USA, 2003. ACM.

[Gting, 1994] Ralf Hartmut Gting. GraphDB: Modeling and querying graphs in databases. In *Proceedings of 20th International Conference on Very Large Databases*, pages 297–308, 1994.

[Guo *et al.*, 2005] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2):158–182, 2005.

[Gutierrez *et al.*, 2004] Claudio Gutierrez, Carlos Hurtado, and Alberto O. Mendelzon. Foundations of semantic web databases. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 95–106, 2004.

[Gyssens *et al.*, 1990] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A graph-oriented object model for database end-user interfaces. *SIGMOD Record*, 19(2):24–33, 1990.

[Haase *et al.*, 2004] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of RDF query languages. In *Proceedings 3rd International Semantic Web Conference*, pages 502–517, Hiroshima (JP), 2004.

[Harris and Shadbolt, 2005] Stephen Harris and Nigel Shadbolt. SPARQL query processing with conventional relational database systems. In *Web Information Systems Engineering (WISE'05 Workshops)*, pages 235–244, 2005.

[Hayes and Gutierrez, 1996] Jonathan Hayes and Claudio Gutierrez. Bipartite graphs as intermediate model for RDF. In *Proceedings of the 3th International Semantic Web Conference (ISWC)*, pages 47Ű–61, 1996.

[Hayes, 2004] Patrick Hayes. RDF semantics. W3C Recommendation, February 2004.

[Horst, 2004] Herman J. ter Horst. Extending the RDFS entailment lemma. In *Proceedings of the Third International Semantic web Conference (ISWC2004)*, pages 77–91, 2004.

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

[Jagadish, 1989] H. V. Jagadish. Incorporating hierarchy in a relational model of data. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 78–87. ACM Press, 1989.

[Karvounarakis *et al.*, 2002] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A declarative query language for RDF. In *Proceedings of the 11th International Conference on the World Wide Web (WWW2002)*, 2002.

[Kerschberg *et al.*, 1976] Larry Kerschberg, Anthony C. Klug, and Dennis Tsichritzis. A taxonomy of data models. In *Proceedings of Systems for Large Data Bases*, pages 43–64. North Holland & IFIP, 1976.

[Kiefer *et al.*, 2007] Christoph Kiefer, Abraham Bernstein, Hong Joo Lee, Mark Klein, and Markus Stocker. Semantic process retrieval with iSPARQL. In *Proceedings of the 4th European Semantic Web Conference (ESWC '07)*. Springer, 2007.

[Kiesel *et al.*, 1996] Norbert Kiesel, Andy Schurr, and Bernhard Westfechtel. GRAS: A graph-oriented software engineering database system. In *IPSEN Book*, pages 397–425, 1996.

[Kim, 1990] Won Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on Knowledge and Data Engineering*, 02(3):327–341, 1990.

[Kochut and Janik, 2007] Krys Kochut and Maciej Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In *Proceedings of 4th European Semantic Web Conferenc (ESWC'07)*, pages 145–159, 2007.

[Kunii, 1987] H. S. Kunii. DBMS with graph data model for knowledge handling. In *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 138–142, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[Kuper and Vardi, 1993] Gabriel M. Kuper and Moshe Y. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.

[Lassila, 2002] Ora Lassila. Taking the RDF model theory out for a spin. In *Proceedings of the First International Semantic Web Conference on The Semantic Web (ISWC 2002)*, pages 307–317, London, UK, 2002. Springer-Verlag.

[Lécluse *et al.*, 1988] Christophe Lécluse, Philippe Richard, and Fernando Vélez. O2, an object-oriented data model. *ACM SIGMOD Recor*, 17(3):424–433, 1988.

[Liu *et al.*, 2004] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 219–230, 2004.

[Manola and Miller, 2004] Frank Manola and Eric Miller. RDF primer. W3C recommendation, 2004. `http://www.w3.org/TR/REC-rdf-syntax/`.

[Matono *et al.*, 2005] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A path-based relational rdf database. In *Proceedings of the 16th Australasian database conference (ADC'05)*, pages 95–103, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[McGuinness and van Harmelen, 2004] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C recommendation, 2004. `http://www.w3.org/TR/owl-features/`.

[Mendelzon and Wood, 1995] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[Mendelzon *et al.*, 1997] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. Querying the world wide web. *Int. J. on Digital Libraries*, 1(1):54–67, 1997. `citeseer.ist.psu.edu/mendelzon97querying.html`.

[Miller *et al.*, 2004] Eric Miller, Ralph Swick, and Dan Brickley. Resource description framework RDF. Recommendation, W3C, 2004.

[Mugnier and Chein, 1992] Marie-Laure Mugnier and Michel Chein. Conceptual graphs: Fundamental notions. *Revue d'intelligence artificielle*, 6(4):365–406, 1992.

[Muñoz *et al.*, 2007] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Minimal deductive systems for rdf. In *Proceedings of the 4th European Semantic Web Conference(ESWC 2007)*, pages 53–67, 2007. `http://www.springerlink.com/content/g8w64n1264874118/`.

[Olson and Ogbuji, 2002] Mike Olson and Uche Ogbuji. Versa: Path-based RDF query language, 2002. `http://copia.ogbuji.net/files/Versa.html`.

[Papakonstantinou and Vassalos, 1999] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD 1999)*, pages 455–466, ACM Press, New York, NY, USA, 1999. `citeseer.ist.psu.edu/article/papakonstantinou99query.html`.

[Papakonstantinou *et al.*, 1995] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of 11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[Perez *et al.*, 2006] Jorge Perez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference*, pages 30–43, Athens (GA US), 2006.

[Polleres *et al.*, 2007] Axel Polleres, François Scharffe, and Roman Schindlauer. SPARQL++ for mapping between RDF vocabularies. In *Proceedings of OTM Conferences*, pages 878–896, 2007.

[Polleres, 2007] Axel Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th World Wide Web Conference (WWW)*, pages 787–796, 2007.

[Prud'hommeaux and Seaborne, 2008] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Recommendation, January 2008.

[Resnik, 1995] Philipp Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 448–453, Montréal (CA), 1995.

[Seaborne, 2004] Andy Seaborne. RDQL - a query language for RDF. Member submission, W3C, 2004.

[Serfiotis *et al.*, 2005] Giorgos Serfiotis, Ioanna Koffina1, Vassilis Christophides, and Val Tannen. Containment and Minimization of RDF/S Query Patterns. In *Proceedings of the 4th International Semantic Web Conference. Lecture Notes in Computer Science, Springer (2005)*, pages 607–623, 2005.

[Shipman, 1981] David W. Shipman. The functional data model and the data languages DAPLEX. *ACM Transactions on Database Systems (TODS)*, 6(1):140–173, 1981.

[Sirin and Parsia, 2007] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL query for OWL-DL. In *Proceedings of the 3rd OWL Experiences and Directions Workshop (OWLED 2007)*, 2007.

[Souzis, 2004] Adam Souzis. RxPath specification proposal, 2004. `http://rx4rdf.liminalzone.org/RxPathSpec`.

[Tarjan, 1981] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of (ACM)*, 28(3):594–614, 1981.

[Theoharis *et al.*, 2008] Yannis Theoharis, Yannis Tzitzikas, Dimitris Kotzinos, and Vassilis Christophides. On graph features of semantic web schemas. *IEEE Transactions on Knowledge and Data Engineering*, 20(5):692–702, 2008.

[Vardi, 1982] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing STOC'82*, pages 137–146, 1982.

[Wood, 1988] Peter T. Wood. *Queries on Graphs*. PhD thesis, Department of Computer Science, University of Toronto, 1988.

[Yannakakis, 1990] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230–242, 1990.

[Zhang and Yoshikawa, 2008] Xinpeng Zhang and Masatoshi Yoshikawa. DCB-DQueryŰQuery Language for RDF using Dynamic Concise Bounded Description. In *Proceedings of the Data Engineering Workshop (DEWS2008), to appear*, 2008.

Part III

Appendices

CPSPARQL Grammar

A

The grammar of CPSPARQL, which is an extension to SPARQL grammar, is given in EBNF specification in the following table.

```
[1]    Query                      ::=  Prolog ( SelectQuery
                                          | ConstructQuery
                                          | DescribeQuery
                                          | AskQuery )
[2]    Prolog                     ::=  BaseDecl? PrefixDecl*
[3]    BaseDecl                   ::=  'BASE' Q_IRI_REF
[4]    PrefixDecl                 ::=  'PREFIX' QNAME_NS Q_IRI_REF
[5]    SelectQuery                ::=  'SELECT' 'DISTINCT'? ( Var+
                                          | '*' ) DatasetClause*
                                          WhereClause SolutionModifier
[6]    ConstructQuery             ::=  'CONSTRUCT' ConstructTemplate
                                          DatasetClause* WhereClause
                                          SolutionModifier
[7]    DescribeQuery              ::=  'DESCRIBE' ( VarOrIRIref+
                                          | '*' ) DatasetClause*
                                          WhereClause? SolutionModifier
[8]    AskQuery                   ::=  'ASK' DatasetClause* WhereClause
[9]    DatasetClause              ::=  'FROM' ( DefaultGraphClause
                                          | NamedGraphClause )
[10]   DefaultGraphClause         ::=  SourceSelector
[11]   NamedGraphClause           ::=  'NAMED' SourceSelector
[12]   SourceSelector             ::=  IRIref
[13]   WhereClause                ::=  'WHERE'? GroupGraphPattern
[14]   SolutionModifier           ::=  OrderClause? LimitClause?
                                          OffsetClause?
[15]   OrderClause                ::=  'ORDER' 'BY' OrderCondition+
[16]   OrderCondition             ::=  ( ( 'ASC' | 'DESC' )
                                          BrackettedExpression )
                                          | ( FunctionCall | Var
```

```
                                          | BrackettedExpression )
[17]    LimitClause                  ::=  'LIMIT' INTEGER
[18]    OffsetClause                 ::=  'OFFSET' INTEGER
[19]    GroupGraphPattern            ::=  '{' GraphPattern '}'
[20]    GraphPattern                 ::=  FilteredBasicGraphPattern
                                           ( GraphPatternNotTriples '.'?
                                           GraphPattern )?
[21']   FilteredBasicGraphPattern    ::=  PathConstraint? BlockOfTriples?
                                           ( Constraint '.'?
                                           FilteredBasicGraphPattern )?
[22]    BlockOfTriples               ::=  PathTriplesSameSubject ( '.'
                                           PathTriplesSameSubject? )*
[23']   GraphPatternNotTriples       ::=  OptionalGraphPattern
                                           | GroupOrUnionGraphPattern
                                           | GraphGraphPattern
                                           | DefinedPathVar
[24]    OptionalGraphPattern         ::=  'OPTIONAL' GroupGraphPattern
[25]    GraphGraphPattern            ::=  'GRAPH' VarOrBlankNodeOrIRIref
                                           GroupGraphPattern
[26]    GroupOrUnionGraphPattern     ::=  GroupGraphPattern ( 'UNION'
                                           GroupGraphPattern )*
[26.1]  DefinedPathVar               ::=  'DEFINED' 'BY' PathVar
                                           VarOrTerm PathVerb VarOrTerm
[26.2]  PathVar                      ::=  VAR3
[27]    Constraint                   ::=  'FILTER' ( BrackettedExpression
                                           | BuiltInCall | FunctionCall )
[27.1]  PathConstraint               ::=  'CONSTRAINT' ConstraintName
                                           RegularExpressionConstraint
[27.2]  ConstraintName               ::=  NCNAME | NCNAME_PREFIX
[28]    FunctionCall                 ::=  IRIref ArgList
[29]    ArgList                      ::=  ( NIL | '(' Expression ( ','
                                           Expression )* ')' )
[30]    ConstructTemplate            ::=  '{' ConstructTriples '}'
[31]    ConstructTriples             ::=  ( TriplesSameSubject ( '.'
                                           ConstructTriples )? )?
[32]    TriplesSameSubject           ::=  VarOrTerm PropertyListNotEmpty
                                           | TriplesNode PropertyList
[32.1]  PathTriplesSameSubject       ::=  VarOrTerm
                                           PathPropertyListNotEmpty
                                           | PathTriplesNode
                                           PathPropertyList
[33]    PropertyList                 ::=  PropertyListNotEmpty?
[33.1]  PathPropertyList             ::=  PathPropertyListNotEmpty?
[34]    PropertyListNotEmpty         ::=  Verb ObjectList ( ';'
                                           PropertyList )?
[34.1]  PathPropertyListNotEmpty     ::=  PathVerb PathObjectList ( ';'
                                          PathPropertyList )?
[35]    ObjectList                   ::=  GraphNode ( ',' ObjectList )?

[35.1]  PathObjectList               ::=  PathGraphNode ( ','
                                           PathObjectList )?
```

```
[36]    Verb                          ::=  VarOrIRIref | 'a'
[36.1]  PathVerb                      ::=  RegularExpression |
                                            ConstrainedRegularExpression
                                            | PathVar
[37]    TriplesNode                   ::=  Collection |
                                            BlankNodePropertyList
[37.1]  PathTriplesNode               ::=  PathCollection |
                                            PathBlankNodePropertyList
[38]    BlankNodePropertyList         ::=  '[' PropertyListNotEmpty ']'
[38.1]  PathBlankNodePropertyList     ::=  '[' PathPropertyListNotEmpty ']'
[39]    Collection                    ::=  '(' GraphNode+ ')'
[39.1]  PathCollection                ::=  '(' PathGraphNode+ ')'
[40]    GraphNode                     ::=  VarOrTerm | TriplesNode
[40.1]  PathGraphNode                 ::=  VarOrTerm | PathTriplesNode
[41]    VarOrTerm                     ::=  Var | GraphTerm
[42]    VarOrIRIref                   ::=  Var | IRIref
[41.1]  RegularExpression             ::=  Rexp (( '|' | '.' ) Rexp )*
[41.2]  ConstrainedRegularExpression  ::=  '{' Rexp (( '|' | '.' ) Rexp )*
                                            '}' '%' ( ( 'SUM' | 'AVG' |
                                            'COUNT' ) '(' Var ( ','
                                            NumericLiteral )? ')' )?
                                            ConstraintName |
                                            RegularExpressionConstraint |
                                            'DISTINCT' | 'LENGTH'
                                            NumericLiteral '%' Rexp
[41.3]  RegularExpressionConstraint   ::=  ( '[' | ']' ) ( 'ALL' | 'EXISTS'
                                            'EDGE' | 'INTEGER'| 'ODD' |
                                            'EVEN' ) Var ( '[' | ']' ) ':'
                                            GraphPattern
[41.4]  Rexp                          ::=  ( '+' | '*' )? Atom
[41.5]  Atom                          ::=  '!'IRIref | '#' | 'a' |
                                           '#' EdgeConstraint | VarOrIRIref
                                            | '(' RegularExpression |
                                            ConstrainedRegularExpression
                                            ')'
[41.6]  EdgeConstraint               ::=  '[' 'EDGE' Var ']' ':'
                                            GraphPattern

[43]    VarOrBlankNodeOrIRIref        ::=  Var | BlankNode | IRIref
[44]    Var                           ::=  VAR1 | VAR2
[45]    GraphTerm                     ::=  IRIref | RDFLiteral | ( '-' |
                                            '+' )? NumericLiteral |
                                            BooleanLiteral | BlankNode |
                                            NIL
[46]    Expression                    ::=  ConditionalOrExpression
[47]    ConditionalOrExpression       ::=  ConditionalAndExpression ( '||'
                                            ConditionalAndExpression )*
[48]    ConditionalAndExpression      ::=  ValueLogical ( '&&'
                                            ValueLogical )*
[49]    ValueLogical                  ::=  RelationalExpression
[50]    RelationalExpression          ::=  NumericExpression
```

```
                                        ( '=' NumericExpression
                                        | '!=' NumericExpression
                                        | '<'  NumericExpression
                                        | '>' NumericExpression
                                        | '<=' NumericExpression
                                        | '>=' NumericExpression )?

[51]    NumericExpression          ::=  AdditiveExpression
[52]    AdditiveExpression         ::=  MultiplicativeExpression ( '+'
                                         MultiplicativeExpression | '-'
                                         MultiplicativeExpression )*
[53]    MultiplicativeExpression   ::=  UnaryExpression
                                        ( '*' UnaryExpression
                                        | '/' UnaryExpression )*
[54]    UnaryExpression            ::=   '!' PrimaryExpression
                                        | '+' PrimaryExpression
                                        | '-' PrimaryExpression
                                        | PrimaryExpression
[55]    PrimaryExpression          ::=  BrackettedExpression
                                        | BuiltInCall
                                        | IRIrefOrFunction | RDFLiteral
                                        | NumericLiteral
                                        | BooleanLiteral | BlankNode
                                        | Var
[56]    BrackettedExpression       ::=  '(' Expression ')'
[57']   BuiltInCall                ::=   'STR' '(' Expression ')'
                                        | 'LANG' '(' Expression ')'
                                        | 'LANGMATCHES' '(' Expression
                                        ',' Expression ')'
                                        | 'DATATYPE' '(' Expression ')'
                                        | 'BOUND' '(' Var ')'
                                        | 'isIRI' '(' Expression ')'
                                        | 'isURI' '(' Expression ')'
                                        | 'isBLANK' '(' Expression ')'
                                        | 'isLITERAL' '(' Expression
                                          ')'
                                        | 'SUM' '(' Var ')' | 'AVG' '('
                                         Var ')' | 'COUNT' '(' Var ')'
                                        | RegexExpression
[58]    RegexExpression            ::=  'REGEX' '(' Expression ','
                                         Expression (',' Expression)?
                                         ')'
[59]    IRIrefOrFunction           ::=  IRIref ArgList?
[60]    RDFLiteral                 ::=  String ( LANGTAG | ( '^^'
                                         IRIref ) )?
[61]    NumericLiteral             ::=  INTEGER | DECIMAL | DOUBLE
[62]    BooleanLiteral             ::=  'true' | 'false'
[63]    String                     ::=  STRING_LITERAL1
                                        | STRING_LITERAL2
                                        | STRING_LITERAL_LONG1
                                        | STRING_LITERAL_LONG2
```

```
[64]    IRIref                      ::=  Q_IRI_REF | QName
[65]    QName                       ::=  QNAME | QNAME_NS
[66]    BlankNode                   ::=  BLANK_NODE_LABEL | ANON

[67]    Q_IRI_REF                   ::=  '<' ([^<>'{}|^`]-[#x00-#x20])*
                                          '>'
[68]    QNAME_NS                    ::=  NCNAME_PREFIX? ':'
[69]    QNAME                       ::=  NCNAME_PREFIX? ':' NCNAME?
[70]    BLANK_NODE_LABEL            ::=  '_:' NCNAME
[71]    VAR1                        ::=  '?' VARNAME
[71.1]  VAR3                        ::=  '??' VARNAME
[72]    VAR2                        ::=  '$' VARNAME
[73]    LANGTAG                     ::=  '@' [a-zA-Z]+ ( '-' [a-zA-Z0-9]+
                                           )*
[74]    INTEGER                     ::=  [0-9]+
[75]    DECIMAL                     ::=  [0-9]+ '.' [0-9]* | '.' [0-9]+
[76]    DOUBLE                      ::=  [0-9]+ '.' [0-9]* EXPONENT | '.'
                                          ([0-9])+ EXPONENT | ([0-9])+
                                          EXPONENT
[77]    EXPONENT                    ::=  [eE] [+-]? [0-9]+
[78]    STRING_LITERAL1             ::=  "'" ( ([^#x27#x5C#xA#xD])
                                          | ECHAR | UCHAR )* "'"
[79]    STRING_LITERAL2             ::=  '"' ( ([^#x22#x5C#xA#xD])
                                          | ECHAR | UCHAR )* '"'
[80]    STRING_LITERAL_LONG1        ::=  "'''" ( ( "'" | "''" )? ( [^'\]
                                          | ECHAR | UCHAR ) )* "'''"
[81]    STRING_LITERAL_LONG2        ::=  '"""' ( ( '"' | '""' )? ( [^"\]
                                          | ECHAR | UCHAR ) )* '"""'
[82]    ECHAR                       ::=  '\' [tbnrf\"']
[83]    UCHAR                       ::=  '\' ( 'u' HEX HEX HEX HEX |
                                          'U' HEX HEX HEX HEX HEX HEX HEX
                                          HEX )
[84]    HEX                         ::=  [0-9] | [A-F] | [a-f]
[85]    NIL                         ::=  '(' WS* ')'
[86]    WS                          ::=  #x20 | #x9 | #xD | #xA
[87]    ANON                        ::=  '[' WS* ']'
[88]    NCCHAR1p                    ::=  [A-Z]
                                          | [a-z]
                                          | [#x00C0-#x00D6]
                                          | [#x00D8-#x00F6]
                                          | [#x00F8-#x02FF]
                                          | [#x0370-#x037D]
                                          | [#x037F-#x1FFF]
                                          | [#x200C-#x200D]
                                          | [#x2070-#x218F]
                                          | [#x2C00-#x2FEF]
                                          | [#x3001-#xD7FF]
                                          | [#xF900-#xFDCF]
                                          | [#xFDF0-#xFFFD]
                                          | [#x10000-#xEFFFF]
                                          | UCHAR
```

```
[89]    NCCHAR1          ::=  NCCHAR1p | '_'
[90]    VARNAME          ::=  ( NCCHAR1 | [0-9] ) ( NCCHAR1
                               | [0-9] | #x00B7
                               | [#x0300-#x036F]
                               | [#x203F-#x2040] )*
[91]    NCCHAR           ::=  NCCHAR1 | '-' | [0-9]
                               | #x00B7 | [#x0300-#x036F]
                               | [#x203F-#x2040]
[92]    NCNAME_PREFIX    ::=  NCCHAR1p ((NCCHAR |'.')*
                               NCCHAR)?
[93]    NCNAME           ::=  NCCHAR1 ((NCCHAR |'.')*
                               NCCHAR)?
```

Résumé étendu
B

Contents

B.1 Motivations et objectifs **174**

B.2 Résumé des contributions **176**

B.3 Organisation de la thèse **178**

B.4 Conclusions **180**

B.4.1 Contributions 180

B.4.2 Comparaison avec les autres langages 181

B.4.3 Perspectives 183

Le world wide web (ou tout simplement le web) est devenu la première source de connaissances pour tous les domaines de la vie. On peut le considérer comme un vaste système d'information qui permet d'échanger des ressources tels que des documents. Le web sémantique est une extension de l'évolution du web visant à donner une forme bien définie et une sémantique aux ressources du web (par exemple, le contenu d'une page web HTML) [Berners-Lee *et al.*, 2001]. Répondre aux requêtes est une fonctionnalité essentielle d'un système d'information, et ainsi du Web Sémantique. Cette thèse étudie les mécanismes actuels de requêtes pour le Web sémantique et le problème de support des chemins dans les bases de connaissances. La motivation de ce travail provient de limitations des langages de requêtes actuels pour supporter et extraire les chemins dans les requêtes.

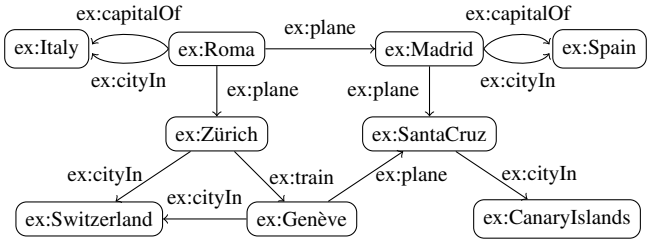

Figure B.1: Un graphe RDF.

B.1 Motivations et objectifs

RDF (Resource Description Framework) est un langage de représentation de connaissances dédié à l'annotation de documents et plus généralement de ressources dans le cadre du Web Sémantique [Miller *et al.*, 2004]. Syntaxiquement, un document RDF peut être représenté indifféremment par un ensemble de triplets (sujet, prédicat, objet), par un document XML, ou par un graphe étiqueté (d'où son nom de graphe RDF). Un graphe RDF est doté d'une sémantique en théorie des modèles [Hayes, 2004], ce qui permet de définir formellement la notion de conséquence sémantique entre graphes RDF, c'est-à-dire, qu'un graphe RDF est une conséquence sémantique d'un autre.

Exemple B.1.1 *Le graphe RDF de la figure B.1, par exemple, se compose d'un ensemble d'arcs reliant des villes avec des moyens de transport tels que chaque arc ou triplet de la forme* (C_1, t, C_2) *indique qu'il existe un moyen de transport de la ville* C_1 *à la ville* C_2 *(ou* C_2 *est directement accessible à partir de* C_1 *par* t*).*

Aujourd'hui, beaucoup de ressources sont annotées par RDF dû à la simplicité de son modèle de données, la sémantique formelle, et l'existence d'un mécanisme d'inférence correct et complet. Bien que RDF ait été initialement conçu comme un langage de représentation des connaissances, il peut être utilisé pour les requêtes RDF. Ainsi, la syntaxe de RDF sert uniformément à représenter des connaissances et à exprimer des requêtes: "Q est une conséquence sémantique de G" peut s'exprimer par "G contient une réponse à la requête Q". Un homomorphisme de graphe permet de calculer cette conséquence de façon correcte et complète [Gutierrez *et al.*, 2004; Baget, 2005]. Plus précisément, la réponse à une requête Q est basée sur le calcul de l'ensemble des homomorphismes possibles de Q dans le graphe RDF représentant la base de connaissances.

Figure B.2: Un patron de graphe de SPARQL.

La nécessité d'ajouter plus l'expressivité dans les requêtes a conduit à définir SPARQL [Prud'hommeaux and Seaborne, 2008], une recommandation du W3C développée pour interroger une base de connaissances RDF (*cf.* [Haase *et al.*, 2004] pour une comparaison des langages de requête pour RDF). Les requêtes SPARQL sont définies à partir des patron des graphes (graph patterns) qui sont fondamentalement des graphes RDF (ou plus précisément, des graphes RDF avec des variables tels que définis dans [Horst, 2004]). Les affectations (maps) qui sont utilisées pour calculer les réponses à une requête dans une base de connaissances RDF sont exploitées par [Perez *et al.*, 2006] pour définir des réponses aux requêtes SPARQL plus complexes et plus expressives en utilisant, par exemple, les disjonctions ou des contraintes fonctionnelles entre les littéraux de la réponse.

Exemple B.1.2 *Un patron de graphe de SPARQL permet de faire une correspondre entre une requête et un graphe RDF. La figure B.2 présente un tel patron. Il peut être utilisé pour trouver les noms des villes et des pays connectés à Roma. Si ce patron est utilisé dans une requête SPARQL contre le graphe G de la figure B.1, il retournera "Madrid" avec son pays "Espagne" et le moyen de transport "plane", et "Zürich" avec son pays "Suisse" et le moyen de transport "plane".*

Néanmoins, la plupart des langages de requêtes qui sont basées sur la sémantique de RDF, comme SPARQL, n'ont pas la capacité d'exprimer et d'extraire des chemins, ce qui est nécessaire pour de nombreuses applications. Par exemple, si l'on veut vérifier s'il existe un itinéraire d'une ville à l'autre (voir Exemple B.1.3).

Une autre approche, employée avec succès dans les bases de données [Cruz *et al.*, 1987; Cruz *et al.*, 1988; de Moor and David, 2003; Liu *et al.*, 2004; Abiteboul *et al.*, 1997; Buneman *et al.*, 1996], mais peu dans le domaine du Web Sémantique, utilise également la structure du graphe RDF, mais ne repose pas sur la sémantique du langage. Dans cette approche, les requêtes sont des expressions régulières, et une réponse est une paire de sommets reliés par au moins un chemin du graphe dont la concaténation des étiquettes des arcs forme un mot qui appartient au langage engendré par l'expression régulière.

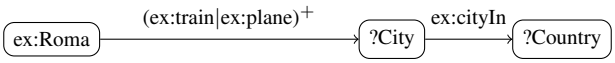

Figure B.3: Un patron de graphe avec des expressions régulières.

Exemple B.1.3 *Considérons un graphe RDF représentant un réseaux de moyens de transport, comme par exemple le graphe G de la figure B.1. L'expression régulière (*`ex:train` | `ex:plane`*)*$^+$*, quand elle sera utilisée comme une requête, cherche les paires de sommets connectés par une séquence de trains et d'avions. Si elle est appliquée au sommet* `ex:Roma` *de G, elle engendre les chemins menant aux sommets* `ex:Madrid`*,* `ex:SantaCruz`*,* `ex:Zürich` *et* `ex:Genève`*. Cette requête, car elle représente des chemins de longueur inconnue, ne peut être exprimée en SPARQL. D'autre part, le graphe de la figure B.2, qui représente un patron de graphe d'une requête SPARQL, ne peut être exprimé par une simple expression régulière.*

Ces deux approches sont orthogonales, certaines requêtes qui peuvent être exprimées dans une approche ne peut pas être exprimées dans l'autre. La figure B.2 montre une requête dont l'image homomorphique dans la base de données n'est pas un chemin et ne peut donc pas être exprimée par une expression régulière, alors que la sémantique RDF ne permet pas d'exprimer des chemins de longueur indéterminée.

Afin de surmonter cette limitation, nous avons développé une approche qui combine les avantages des deux approches. Cette approche combinée, dans laquelle les arcs du graphe peuvent être étiquetés par d'expressions régulières, peut être utilisée pour supporter les chemins (voir la figure B.3).

B.2 Résumé des contributions

Afin de définir formellement ce langage, nous avons d'abord introduit PRDF (pour Path RDF) comme une extension de RDF dans laquelle les arcs peuvent être étiquetés par des expressions régulières [Alkhateeb *et al.*, 2005; Alkhateeb *et al.*, 2007]. Parce que nous voulons fonder la définition de notre langage sur la sémantique de RDF en laissant la porte ouverte à d'autres extensions, nous définissons la sémantique de PRDF au-dessus de la sémantique RDF, et nous fournissons un algorithme pour vérifier si un graphe PRDF est une conséquence sémantique d'un graphe RDF.

Les graphes PRDF servent ensuite à définir une extension de SPARQL, le langage de base PSPARQL, remplaçant les patrons de graphes RDF utilisés dans SPARQL par des graphes PRDF, c'est-à-dire des patrons de graphes avec des expressions régulières.

Exemple B.2.1 *La requête PSPARQL suivante:*

```
SELECT ?City
WHERE {
        ex:Paris (ex:train|ex:plane)+ ?City .
        ?City ex:capitalOf ?Country .
      }
ORDER BY Asc(?City)
```

retourne dans un ordre croissant, l'ensemble des villes accessible de Paris par une séquence de trains et d'avions, qui sont des villes capitales.

Pour ajouter plus d'expressivité à PSPARQL permettant à spécifier des informations sur les sommets qui appartiennent à un chemin défini par une expression régulière "par exemple, tous les arrêts doivent être soit des capitales soit des villes de plus de 200000 habitants.", nous avons étendu PRDF. Plus précisément, nous avons définit CPRDF (Pour Constrained Path RDF) qui étend la syntaxe et la sémantique de PRDF pour gérer les contraintes sur les sommets dans les chemins traversés.

Exemple B.2.2 *Le graphe présenté dans la figure B.4, où const =*]`ALL ?Stop`] : {{`?Stop ex:capitalOf ?Country .`} `UNION` {`?Stop ex:population ?Pop . FILTER (?Pop > 200000)`}}*, est un graphe CPRDF.*

Nous avons aussi caractérisé les réponses à une requête réduite à un graphe CPRDF par des homomorphismes (des affectations particulières) et avons fournit des algorithmes corrects et complets pour calculer ces réponses. Cette propriété est suffisante pour étendre le langage d'interrogation PSPARQL à CPSPARQL en utilisant cette fois les graphes CPRDF dans les patrons de graphes de SPARQL

Exemple B.2.3 *La requête CPSPARQL:*

```
SELECT ?City
WHERE {
        CONSTRAINT const ]ALL ?Stop]: {{ ?Stop ex:capitalOf ?Country. }
                                          UNION
                                       { ?Stop ex:population ?Pop .
                                         FILTER (?Pop > 200000)
```

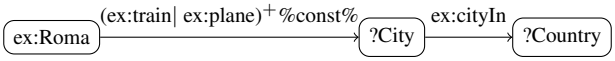

Figure B.4: Un patron de graphe avec des expressions régulières contraintes.

```
                                    }
                                   }
    ex:Paris (ex:train|ex:plane)+%const% ?City .
    ?City ex:capitalOf ?Country .
  }
```

dont le patron de graphe est le graphe CPRDF de l'exemple B.2.2, peut être utilisée pour trouver les noms des villes et des pays tels que chaque ville est connectée à Roma par un chemin (une séquence de trains et d'avions) dont tous les sommets sont soit des villes capitales soit des villes de plus de 200000 *habitants.*

Nous avons implémenté un évaluateur pour répondre aux requêtes PSPARQL ou CPSPARQL. L'entrée à ce évaluateur est une requête sous une forme de texte où chaque RDF graphe identifié par une URL dans la requête doit être rédigée dans le langage Turtle [Beckett, 2006] ou XHTML+RDFa[1] dans un document (X)HTML. L'évaluateur est fourni avec deux parseurs:

- un pour parser des graphes RDF écrites en Turtle, et
- l'autre pour parser des requêtes écrites selon la syntaxe de CPSPARQL, qui est compatible avec la syntaxe de SPARQL (voir `http://psparql.inrialpes.fr`).

La sortie de l'évaluateur est le résultat de la requête sous la forme de texte.

B.3 Organisation de la thèse

Chapitre 2: The RDF Language

Dans ce chapitre, nous présentons le langage RDF simple et sa généralisation GRDF sans introduire les vocabulaires RDF et RDFS: la syntaxe, la sémantique et un mécanisme d'inférence pour interroger les graphes RDF ou GRDF (la requête et la base de connaissance sont deux graphes RDF ou GRDF). Ce langage sera utilisé pour représenter les bases de connaissances sur lesquelles les requêtes seront évaluées.

[1] `http://www.w3.org/TR/xhtml-rdfa-primer/`

Chapitre 3: Querying RDF Graphs

Nous discutons dans ce chapitre les langages de requêtes pour le Web Sémantique en général et en particulier pour RDF, et mettons en évidence les différences principales entre eux et notre proposition. Parmi eux, nous citons les langages de requêtes utilisés dans les bases de données comme G, G+, Graphlog, Lorel, UnQL, RDQL, STRUQL, XPath, et SPARQL et ses extensions comme SPARQ2L et SPARQLeR.

Chapitre 4: A General Graph Framework with Paths

Dans ce chapitre, nous présentons une extension de RDF, appelée PRDF (pour Path RDF). Syntaxiquement, les graphes PRDF sont des graphes RDF étendus dont les arcs sont étiquetés par des générateurs de langages réguliers. Comme exemple, nous utilisons les expressions régulières pour engendrer des langages réguliers possiblement infinis. Ainsi, les langages réguliers (ou expressions régulières) peuvent être utilisés pour engendrer les chemins dans un graphe RDF tel que le mot obtenu par la concaténation des labels des arcs doit appartenir au langage engendré par l'expression régulière. Sémantiquement, nous définissons une sémantique étendant la sémantique formelle de RDF d'une façon à pouvoir interpréter les chemins engendrés par les expressions régulières. Puis, nous fournissons un mécanisme d'inférence correct et complet basé sur des homomorphismes de graphes pour interroger les graphes RDF en utilisant les graphes PRDF comme des requêtes.

Chapitre 5: The PSPARQL Query Language

Les graphes PRDF obtenus dans le chapitre précédent seront utilisés pour définir les patrons de graphes PSPARQL, l'extension de SPARQL que nous proposons. La syntaxe de PSPARQL est définie en remplaçant les patrons de graphes de SPARQL par des graphes PRDF. Puis, nous donnons une sémantique opérationnelle qui étend celle de SPARQL en utilisant les PRDF homomorphismes pour définir les réponses à une requête PSPARQL dans un graphe RDF ainsi des algorithmes pour calculer ces réponses.

Chapitre 6: Constrained Paths in SPARQL

PSPARQL sert aussi à définir dans ce chapitre une nouvelle extension, appelée CPSPARQL. Le langage CPSPARQL étend PSPARQL en permettant d'exprimer

des contraintes sur les sommets dans les chemins traversés. De la même façon que pour PSPARQL, nous définissons la syntaxe et la sémantique de CPSPARQL en étendant les graphes PRDF aux graphes CPRDF (pour Constrained Path RDF) dont les étiquettes des arcs sont des expressions régulières contraintes.

Chapitre 7: Other Possible Extensions

Ce chapitre discute des extensions naturelles de CPSPARQL. Chaque extension est présentée avec un exemple qui montre son utilité. En particulier, utiliser des variables de chemins dans les requêtes de CPSPARQL qui servent à extraire les chemins, exprimer des contraintes sur ces variables, extraire des chemins dont les prédicats sont similaires à un prédicat donné, définir une forme de requêtes imbriquées, et étendre les contraintes utilisées pour définir des expressions régulières contraintes.

Chapitre 8: Querying RDFS Graphs

Dans ce chapitre, nous présentons les langages RDF et RDFS qui sont deux extensions de RDF simple avec des vocabulaires particuliers. Nous présentons aussi plusieurs méthodes pour interroger les graphes RDFS et fournissons une approche possible basée sur la réécriture de requêtes SPARQL en requêtes PSPARQL.

Chapitre 9: Implementation and Experiments

Ce chapitre présente une implémentation concrète de nos extensions basées sur les idées présentées dans les Chapitres 4–6, ainsi que plusieurs tests expérimentaux de ce prototype.

B.4 Conclusions

B.4.1 Contributions

Nous avons traité dans cette thèse le problème du support et de l'extraction des chemins dans les bases de connaissances du web sémantique. Les langages de requêtes actuels pour interroger le web sémantique sont soit basés sur l'algèbre relationnelle qui n'ont pas la possibilité d'exprimer des requêtes récursives ou soit des langages qui supportent une forme limitée de chemin et qui ne supportent pas des requêtes SQL ou des requêtes conjonctives.

Notre travail est donc motivé par la nécessité de mettre au point un compromis qui supporte les deux formes de langages de requêtes. Bien qu'il peut être adapté à d'autres formalismes, il est appliqué dans le contexte de RDF(S) et de son modèle de données.

Nous avons introduit le langage PRDF qui étend la syntaxe et la sémantique du langage RDF de façon à pouvoir étiqueter les arcs d'un graphe par des expressions régulières. Nous avons fournit des algorithmes étendant le calcul d'homomorphismes entre graphes pour calculer les réponses à une requête PRDF dans un graphe RDF et nous avons montré qu'ils sont corrects et complets vis-à-via de la sémantique étendue. Les graphes PRDF obtenus sont utilisés pour définir les patrons de graphes PSPARQL, l'extension de SPARQL que nous proposons.

PSPARQL est la base du développement d'une nouvelle extension, appelée CPSPARQL, qui permet en outre d'exprimer d'autres constructions dans les requêtes SPARQL telles que des contraintes sur les sommets des chemins traversés. Cette extension a plusieurs avantages, parmi eux, elle ajoute plus d'expressivité à PSPARQL et améliore l'efficacité en utilisant des contraintes prédéfinies qui servent à couper les chemins inutiles. Nous avons développé un évaluateur de requêtes PSPARQL et CPSPARQL[2]. Cet évaluateur a passé avec succès tous les tests proposés par le W3C pour le langage de requêtes SPARQL[3].

B.4.2 Comparaison avec les autres langages

Nous avons comparé PSPARQL et CPSPARQL aux autre langages de requêtes en appuyant sur [Haase *et al.*, 2004; Angles and Gutiérrez, 1995]. [Haase *et al.*, 2004] fait une comparaison entre plusieurs langages de requête basée sur 14 traits distincts. Les tests comprennent Path expression, Optional path and Recursion. L'interprétation de ces trois tests est en fait l'utilisation des patrons de graphes, les patrons de graphes optionnels, et des expressions récursives. Pour supprimer l'ambiguïté, nous renommons les trois tests en: Graph pattern, Optional Pattern, et Recursion (ou Regular expression). De [Angles and Gutiérrez, 1995], nous incluons les traits suivants: Adjacent nodes, Adjacent edges, Fixed-length path, Degree of a node, Distance between nodes, and Diameter. Nous ajoutons également les traits suivants: Regular expression variable, Constraints, Path variable, Constrained regular expression, Inverse path, et Non-simple path. Il y avait 8 langages de requêtes dans la comparaison initiale ([Haase *et al.*, 2004]) de laquelle nous

[2] <http://psparql.inrialpes.fr/>
[3] <http://www.w3.org/2001/sw/DataAccess/tests/>

	SPARQL	Corese	SPARQ2L	SPARQLeR	(P/CP)SPARQL	G+	GraphLog	STRUQL	RDQL	SeRQL	Versa	RQL	LOREL
Graph pattern	●	●	●	●	●/●	●	●	●	●	●	●	●	●
Optional pattern	●	●	●	●	●/●	-	-	-	-	●	●	○	-
Union	●	●	●	●	●/●	-	-	-	-	●	●	●	●
Constraints	●	●	●	●	●/●	-	-	-	●	●	●	●	●
Difference	●	●	●	●	●/●	-	-	-	-	●	○	●	-
Quantification	-	-	-	-	-/-	-	-	-	-	●	-	●	●
Aggregation	-	○	-	-	○/○	-	-	-	-	-	●	●	●
Reification	●	●	●	●	●/●	-	-	-	○	●	○	○	-
Collections and Containers	○	○	○	○	○/○	-	-	-	○	○	○	●	-
Namespace	●	●	●	●	●/●	-	-	-	○	●	-	●	-
Language	●	●	●	●	●/●	-	-	-	-	●	-	-	-
Lexical space	●	●	●	●	●/●	-	-	-	●	●	●	●	-
Value space	●	●	●	●	●/●	○	○	○	○	●	-	●	●
Entailment	-	●	-	○	●/●	-	-	-	○	●	-	●	-
Recursion (Regular expression)	-	-	●	●	●/●	●	●	●	-	-	●	○	●
Regular expression variable	-	-	-	-	●/●	●	●	●	-	-	○	○	●
Constrained regular expression	-	-	○	○	-/●	-	-	-	-	-	-	-	-
Fixed-length path	-	●	○	○	●/●	●	●	●	○	○	-	●	●
Path variable	-	●	●	●	●/●	-	-	-	-	-	-	-	●
Inverse Path	-	-	-	●	-/●	●	●	-	-	-	-	-	-
Non-simple path	-	-	●	-	●/●	-	-	-	-	-	-	-	-
Adjacent nodes	●	●	●	●	●/●	●	●	●	○	○	○	○	●
Adjacent edges	●	●	●	●	●/●	●	●	●	○	-	○	○	○
Degree of a node	-	-	-	-	-/-	●	●	●	-	-	-	○	-
Distance between nodes	-	-	-	-	-/-	●	●	●	-	-	-	-	-
Diameter	-	-	-	-	-/-	●	●	●	-	-	-	-	-

Table B.1: Une comparaison entre des langages de requêtes.

choisissons RQL RDQL, SeRQL et Versa, qui semblent représenter les langages les plus expressifs pour supporter les deux types d'interrogation (c'est-à-dire modèles à base de chemin et modèles de la base relationnelle); nous choisissons G+, GraphLog, STRUQL, LOREL de [Angles and Gutiérrez, 1995]; et nous ajoutons SPARQL, Corese, SPARQ2L, SPARQLeR et (C)PSPARQL.

Dans la table B.1, les Colonnes représentent langages de requêtes et les lignes représentent les caractéristiques ou des types requêtes. En outre, nous utilisons - pour indiquer que la fonctionnalité (ou le type de requête) n'a pas un support dans le langage de requêtes, ○ pour indiquer qu'il existe un support partiel (limitée), et enfin • pour un support complet.

La table B.1 résume les différences principales entre les extensions actuelles de SPARQL, (C)PSPARQL et d'autres langages de requêtes. La plupart des éléments autorisés dans ces extensions sont également supportés dans CPSPARQL. Notez que SPARQLeR (respectivement, SPARQ2L) utilise le FILTER de SPARQL (respectivement, utilise ContainANY et ContainALL) pour faire le filtrage des chemins. Par exemple, vérifier si un chemin correspond à un mot dans une expression régulière et vérifier l'existence d'un sommet dans le chemin. Nous conjecturons que nous pouvons exprimer ces contraintes en utilisant les expressions régulières contraintes de CPSPARQL. CPSPARQL et SPARQ2L sont les seules langages qui supportent les chemins avec des cycles. Cependant, les algorithmes en SPARQ2L ne sont pas complètes pour ce genre de chemins, et il n'a pas un support des chemins inverses.

Comme on peut le voir dans la table, il existe un grand nombre de fonctionnalités dans SPARQL et ses extensions qui ne peuvent pas être exprimées dans les langages anticipant SPARQL comme G+, GraphLog, et d'autres.

B.4.3 Perspectives

Traitement de l'alignement avec les langages de requêtes

Les problèmes soulevés par les ontologies hétérogènes peuvent être résolus en établissant les correspondances entre les entités de ces ontologies et en traitant l'alignement pour la transformation de données. L'utilisation des langages de requête comme suggéré dans [Euzenat *et al.*, 2008] pour la transformation des données serait un choix naturel, car ils permettent l'extraction et la transformation de données. SPARQL est donc un bon candidat, en particulier, lorsque les ontologies sont décrites en RDF(S) et OWL. Cependant, il y a des pièces manquantes

de SPARQL comme par exemple le support des chemins, agrégat de fonctions, la génération de valeur. L'intégration de deux langages, comme SPARQL++ et CPSPARQL, fournit des requêtes qui sont suffisantes pour couvrir les langages d'alignement les plus expressifs (comme par exemple [Euzenat *et al.*, 2007]). Par exemple, la requête CPSPARQL suivante:

```
CONSTRUCT { ?x o2:potentialCollaborator ?y . }
WHERE { ?x foaf:knows+ ?y.
        ?x o1:topic ?t.
        ?y o1:topic ?t.
        ?x rdf:type o1:researcher .
        ?y rdf:type o1:researcher .
      }
```

pourrait être utilisée pour créer une ontologie qui contient la relation `potentialCollaborator` entre deux chercheurs exprimé par le fait qu'un chercheur est potentiellement collaborateur à l'autre si ils travaillent sur le même sujet et connaître les uns les autres.

Répondre à une requête dans un système distribué

Dans cette direction, nous voudrions profiter de la relation entre les requêtes conjonctives et les requêtes SPARQL, et de notre initial travail sur répondre aux requêtes conjonctives dans les environnements distribués [Alkhateeb and Zimmermann, 2007] pour la conception d'une infrastructure de l'évaluation de requêtes aux chemins dans les environnements distribués. Dans cet article, nous avons étudié le problème de répondre à une requête sur un système distribué de bases de connaissances et défini les réponses distribuées d'une requête exprimée en termes d'une base de connaissances ou d'un ontologie (appelé l'ontologie cible) dans le système. Comme les réponses à une requête SPARQL sont définies par la construction des affectations (c'est-à-dire, les affectations de graphes GRDF de la requête SPARQL dans la base de connaissances) et un GRDF graphe est un cas particulier d'une requête conjonctive, nous pouvons utiliser la définition de réponse distribuée pour définir des réponses aux requêtes SPARQL.

Résumé: RDF est un langage de représentation des connaissances dédié à l'annotation des ressources dans le Web Sémantique. Bien que RDF peut être lui-même utilisé comme un langage de requêtes pour interroger une base de connaissances RDF (utilisant la conséquence RDF), la nécessité d'ajouter plus d'expressivité dans les requêtes a conduit à définir le langage de requêtes SPARQL. Les requêtes SPARQL sont définies à partir des patrons de graphes qui sont fondamentalement des graphes RDF avec des variables. Les requêtes SPARQL restent limitées car elles ne permettent pas d'exprimer des requêtes avec une séquence non-bornée de relations (par exemple, "Existe-t-il un itinéraire d'une ville A à une ville B qui n'utilise que les trains ou les bus?"). Nous montrons qu'il est possible d'étendre la syntaxe et la sémantique de RDF, définissant le langage PRDF (pour Path RDF) afin que SPARQL puisse surmonter cette limitation en remplaçant simplement les patrons de graphes basiques par des graphes PRDF. Nous étendons aussi PRDF à CPRDF (pour Constrained Path RDF) permettant d'exprimer des contraintes sur les sommets des chemins traversés (par exemple, "En outre, l'une des correspondances doit fournir une connexion sans fil."). Nous avons fourni des algorithmes corrects et complets pour répondre aux requêtes (la requête est un graphe PRDF ou CPRDF, la base de connaissances est un graphe RDF) basés sur un homomorphisme particulier, ainsi qu'une analyse détaillée de la complexité. Enfin, nous utilisons les graphes PRDF ou CPRDF pour généraliser les requêtes SPARQL, définissant les extensions PSPARQL et CPSPARQL, et fournissons des tests expérimentaux en utilisant une implémentation complète de ces deux langages.

Mots-Clés: *Langage de Représentation des Connaissances, RDF(S), Web Sémantique, Langages de Requêtes, SPARQL, Homomorphisme de Graphes, Langages Réguliers, Expressions Régulières, Extensions de SPARQL , PRDF, PSPARQL, CPRDF, CPSPARQL.*

Abstract: RDF is a knowledge representation language dedicated to the annotation of resources within the Semantic Web. Though RDF itself can be used as a query language for an RDF knowledge base (using RDF semantic consequence), the need for added expressivity in queries has led to define the SPARQL query language. SPARQL queries are defined on top of graph patterns that are basically RDF graphs with variables. SPARQL queries remain limited as they do not allow queries with unbounded sequences of relations (e.g. "does there exist a trip from town A to town B using only trains or buses?"). We show that it is possible to extend the RDF syntax and semantics defining the PRDF language (for Path RDF) such that SPARQL can overcome this limitation by simply replacing the basic graph patterns with PRDF graphs, effectively mixing RDF reasoning with database-inspired regular paths. We further extend PRDF to CPRDF (for Constrained Path RDF) to allow expressing constraints on the nodes of traversed paths (e.g. "Moreover, one of the correspondences must provide a wireless connection."). We have provided sound and complete algorithms for answering queries (the query is a PRDF or a CPRDF graph, the knowledge base is an RDF graph) based upon a kind of graph homomorphism, along with a detailed complexity analysis. Finally, we use PRDF or CPRDF graphs to generalize SPARQL graph patterns, defining the PSPARQL and CPSPARQL extensions, and provide experimental tests using a complete implementation of these two query languages.

Keywords: *Knowledge Representation Languages, RDF(S), Querying Semantic Web, SPARQL, Graph Homomorphism, Regular Languages, Regular Expressions, SPARQL Extensions, PRDF, PSPARQL, CPRDF, CPSPARQL.*