Foundations of RDF Databases

Marcelo Arenas[1], Claudio Gutierrez[2], and Jorge Pérez[1]

[1] Department of Computer Science, Pontificia Universidad Católica de Chile
[2] Department of Computer Science, Universidad de Chile

Abstract. The goal of this paper is to give an overview of the basics of the theory of RDF databases. We provide a formal definition of RDF that includes the features that distinguish this model from other graph data models. We then move into the fundamental issue of querying RDF data. We start by considering the RDF query language SPARQL, which is a W3C Recommendation since January 2008. We provide an algebraic syntax and a compositional semantics for this language, study the complexity of the evaluation problem for different fragments of SPARQL, and consider the problem of optimizing the evaluation of SPARQL queries, showing that a natural fragment of this language has some good properties in this respect. We furthermore study the expressive power of SPARQL, by comparing it with some well-known query languages such as relational algebra. We conclude by considering the issue of querying RDF data in the presence of RDFS vocabulary. In particular, we present a recently proposed extension of SPARQL with navigational capabilities.

1 Introduction

The Resource Description Framework (RDF) [34] is a data model for representing information about World Wide Web resources. Jointly with its release in 1998 as Recommendation of the W3C, the natural problem of querying RDF data was raised. Since then, several designs and implementations of RDF query languages have been proposed. In 2004, the RDF Data Access Working Group, part of the W3C Semantic Web Activity, released a first public working draft of a query language for RDF, called SPARQL [45]. Since then, SPARQL has been rapidly adopted as the standard for querying Semantic Web data. In January 2008, SPARQL became a W3C Recommendation.

RDF and SPARQL are two of the core technologies in the data and query layers of the *Semantic Web stack*. In this paper, we give an overview of the current state of the theory of RDF and SPARQL from a database perspective. We first provide a formal definition of RDF that includes the features that distinguish this model from other database models. We then move into the fundamental issue of querying RDF data with SPARQL. We provide an algebraic syntax and a compositional semantics for this language, study the complexity of the evaluation problem for different fragments of SPARQL, and consider the problem of optimizing the evaluation of SPARQL queries, showing that a natural fragment of this language has some good properties in this respect. We furthermore

study the expressive power of SPARQL, by comparing it with some well-known query languages such as relational algebra. We conclude by considering the issue of querying RDF data in the presence of RDFS vocabulary. In particular, we present a recently proposed extension of SPARQL with navigational capabilities, and show that this language is expressive enough to deal with the semantics of the RDFS vocabulary.

The paper is organized as follows. In Section 2, we introduce RDF as a data model. In Section 3, we provide a formalization of the syntax and semantics of SPARQL. In Section 4, we study the complexity of the evaluation problem for SPARQL and some optimization results for this language. In Section 5, we study the expressiveness of SPARQL. Finally, we present in Section 6 an extension of SPARQL that gives navigational capabilities to the language and allows to deal with the RDFS vocabulary.

2   The RDF Data Model

The Semantic Web is a proposal to build an infrastructure of machine-readable semantics for the data on the Web. In 1998, the W3C issued a recommendation of a metadata model and language to serve as the basis for such infrastructure, the *Resource Description Framework (RDF)* [32]. As RDF evolves, it is increasingly gaining attraction from both researchers and practitioners, and is being implemented in world-wide initiatives such as the Open Directory Project [39], Dublin Core [48], FOAF [49], and RSS [46].

RDF follows the W3C design principles of interoperability, extensibility, evolution and decentralization. Particularly, the RDF model was designed to have a simple data model, with a formal semantics and provable inference, with an extensible URI-based vocabulary, and which allows anyone to make statements about any resource. In the RDF model, the universe to be modeled is a set of *resources*, essentially anything that can have a *universal resource identifier*, URI [50]. The language to describe them is a set of *properties*, technically binary predicates. Descriptions are *statements* very much in the subject-predicate-object structure, where predicate and object are resources or strings. Both subject and object can be anonymous objects, known as *blank nodes.* In addition, the RDF specification includes a built-in vocabulary with a normative semantics (RDFS). This vocabulary deals with inheritance of classes and properties, as well as typing, among other features [11].

The RDF model is specified in a series of W3C documents [11,27,32,34]. In this section, we introduce an abstract version of the RDF data model, which is both a fragment following faithfully the original specification, and also an abstract version suitable to do formal analysis. What is left out are features of RDF dealing with some implementation issues, such as detailed typing issues, some distinguish vocabulary which has no particular semantics, and all topics involved with the XML-based syntax and serialization. The original formulation of this fragment was introduced in [23], and enriched and corrected in [37]. The main goal of isolating such a fragment is to have a simple and stable core over

which to discuss theoretical issues, dealing with RDF from a database point of view.

2.1   RDF Graphs

Assume there are pairwise disjoint infinite sets U (RDF URI references) and B (Blank nodes)[1]. Through the paper we assume U and B fixed, and for simplicity we denote unions of these sets simply concatenating their names. A tuple $(s, p, o) \in UB \times U \times UB$ is called an *RDF triple*. In this tuple, s is the *subject*, p the *predicate*, and o the *object*.

Definition 1. *An* RDF graph *(or simply a graph) is a set of RDF triples. A graph is* ground *if it has no blank nodes.*

Graphically, we represent RDF graphs as follows: each triple (s, p, o) is represented by a labeled edge $s \xrightarrow{p} o$. Notice that the set of arc labels can have a non-empty intersection with the set of node labels. Thus, technically speaking, and "RDF graph" is not a graph in the classical sense (for further discussion on this issue see [26]).

In what follows, we need the fundamental notion of homomorphism. Given two RDF graphs G_1 and G_2, a *homomorphism* $h : G_1 \to G_2$ is a mapping from UB to UB such that $h(u) = u$ for every element $u \in U$, and for every triple (s, p, o) in G_1, it holds that $(h(s), h(p), h(o)) \in G_2$. We denote by $h(G_1)$ the RDF graph $\{(h(s), h(p), h(o)) \mid (s, p, o) \in G_1\}$. Thus, a homomorphism h from G_1 to G_2 is such that $h(G_1) \subseteq G_2$.

2.2   RDFS

The RDF specification includes a set of reserved words, the RDFS vocabulary (RDF Schema [11]), which is designed to describe relationships between resources and properties like attributes of resources (traditional attribute-value pairs). Roughly speaking, this vocabulary can be conceptually divided into the following groups:

(a) A set of *properties*, which are binary relations between subject resources and object resources: rdfs:subPropertyOf (denoted by `sp` in this paper), rdfs:subClassOf (`sc`), rdfs:domain (`dom`), rdfs:range (`range`) and rdf:type (`type`).
(b) A set of classes, that denote set of resources. Elements of a class are known as *instances* of that class. To state that a resource is an instance of a class, the reserved word `type` may be used.

---

[1] For the sake of simplicity, here we do not make a special distinction between URIs and Literals, and we assume that RDF graphs are constructed by using only URIs and Blank nodes. The inclusion of literals does not change any of the results of this paper.

(c) Other functionalities, like a system of classes and properties to describe lists, and a system for doing reification.
(d) Utility vocabulary used to document, comment, etc. (the complete vocabulary can be found in [11]).

The groups in (b), (c) and (d) have a light semantics, essentially describing their internal relationships in the ontological design of the system of classes of RDFS. Their semantics is defined by a set of "axiomatic triples" [27], which express the relationships among these reserved words. All axiomatic triples are "structural", in the sense that do not refer to external data. Much of this semantics corresponds to what in standard languages is captured via typing.

On the contrary, the group (a) is formed by predicates whose intended meaning is non-trivial, and is designed to relate individual pieces of data external to the vocabulary of the language. Their semantics is defined by rules which involve variables (to be instantiated by actual data). For example, rdfs:subClassOf (`sc`) is a reflexive and transitive binary property; and when combined with rdf:type (`type`) specify that the type of an individual (a class) can be lifted to that of a superclass.

The group (a) forms the core of the RDF language and, from a theoretical point of view, it has been shown to be a very stable core to work with (the detailed arguments supporting this claim are given in [37]). Thus, throughout the paper we focused on the fragment of RDFS given by the set of keywords $\{sp, sc, type, dom, range\}$.

2.3 Semantics of RDF Graphs

In this section, we present the formalization of the semantics of RDF given in [27,37]. The normative semantics for RDF graphs given in [27] follows a standard logical treatment, including classical notions such as model, interpretation, entailment, and so on. We present the simplification of the normative semantics proposed in [37]. It is important to notice that these two approaches were shown to be equivalent for the fragment of the RDFS vocabulary considered in this paper [37].

An RDF interpretation is a tuple $\mathcal{I} = (Res, Prop, Class, PExt, CExt, Int)$, where (1) Res is a nonempty set of *resources*, called the *domain* or *universe* of $\mathcal{I}$; (2) $Prop$ is a set of property names (not necessarily disjoint from Res); (3) $Class \subseteq Res$ is a distinguished subset of Res identifying if a resource denotes a class of resources; (4) $PExt : Prop \rightarrow 2^{Res \times Res}$, a mapping that assigns an *extension* to each property name; (5) $CExt : Class \rightarrow 2^{Res}$ a mapping that assigns a set of resources to every resource denoting a class; (6) $Int : U \rightarrow Res \cup Prop$, the *interpretation mapping*, is a mapping that assigns a resource or a property name to each element of U.

Intuitively, a ground triple (s, p, o) in a graph G is true under the interpretation $\mathcal{I}$, if p is *interpreted* as a property name, s and o are *interpreted* as resources, and the interpretation of the pair (s, o) belongs to the extension of the property assigned to p. Formally, we say that $\mathcal{I}$ satisfies the ground triple (s, p, o) if

$Int(p) \in Prop$ and $(Int(s), Int(o)) \in PExt(Int(p))$. An interpretation must also satisfy additional conditions induced by the usage of the RDFS vocabulary. For example, an interpretation satisfying the triple $(c_1, \texttt{sc}, c_2)$ must interpret c_1 and c_2 as classes of resources, and must assign to c_1 a subset of the set assigned to c_2. More formally, we say that $\mathcal{I}$ satisfies $(c_1, \texttt{sc}, c_2)$ if $Int(c_1), Int(c_2) \in Class$ and $CExt(c_1) \subseteq CExt(c_2)$.

Blank nodes work as existential variables. Intuitively, a triple (x, p, o) would be true under $\mathcal{I}$, where x is a blank node, if there exists a resource s such that (s, p, o) is true under $\mathcal{I}$. An arbitrary element can be chosen when interpreting a blank node, with the restriction that all the occurrences of the same blank node in an RDF graph must be replaced by the same value. To formally deal with blank nodes, an extension of the interpretation mapping Int is used. Let $A : B \rightarrow Res$ be a function between blank nodes and resources. Then $Int_A : UB \rightarrow Res$ is defined as the extension of function Int: $Int_A(x) = A(x)$ for $x \in B$, and $Int_A(x) = Int(x)$ for $x \in U$.

We next formalize the notion of *model* for an RDF graph [27,37]. We say that the RDF interpretation $\mathcal{I} = (Res, Prop, Class, PExt, CExt, Int)$ is a model of (is an interpretation for) an RDF graph G, denoted by $\mathcal{I} \models G$, if the following conditions hold:

Simple Interpretation:
  - there exists a function $A : B \rightarrow Res$ such that for each $(s, p, o) \in G$, it holds that $Int(p) \in Prop$ and $(Int_A(s), Int_A(o)) \in PExt(Int(p))$.

Properties and Classes:
  - $Int(\texttt{sp})$, $Int(\texttt{sc})$, $Int(\texttt{type})$, $Int(\texttt{dom})$, $Int(\texttt{range}) \in Prop$,
  - if $(x, y) \in PExt(Int(\texttt{dom})) \cup PExt(Int(\texttt{range}))$, then $x \in Prop$ and $y \in Class$.

Sub-property:
  - $PExt(Int(\texttt{sp}))$ is transitive and reflexive over $Prop$,
  - if $(x, y) \in PExt(Int(\texttt{sp}))$, then $x, y \in Prop$ and $PExt(x) \subseteq PExt(y)$.

Sub-class:
  - $PExt(Int(\texttt{sc}))$ is transitive and reflexive over $Class$,
  - if $(x, y) \in PExt(Int(\texttt{sc}))$, then $x, y \in Class$ and $CExt(x) \subseteq CExt(y)$.

Typing:
  - $(x, y) \in PExt(Int(\texttt{type}))$ if and only if $y \in Class$ and $x \in CExt(y)$,
  - if $(x, y) \in PExt(Int(\texttt{dom}))$ and $(u, v) \in PExt(x)$, then $u \in CExt(y)$,
  - if $(x, y) \in PExt(Int(\texttt{range}))$ and $(u, v) \in PExt(x)$, then $v \in CExt(y)$.

Example 1. Figure 1 shows an RDF graph storing information about painters. All the triples in the graph are composed by elements in U, except for the triples containing the blank node X. Consider now the interpretation $\mathcal{I} = (Res, Prop, Class, PExt, CExt, Int)$ defined as follows:

  - $Res = \{$Painter, Guayasamin, Cubist, creates, paints, Guernica, Bilbao$\}$.
  - $Prop = \{$paints, creates, exhibited_in, $\texttt{type}, \texttt{sp}, \texttt{sc}, \texttt{dom}, \texttt{range}\}$.
  - $Class = \{$Cubist, Painter$\}$.
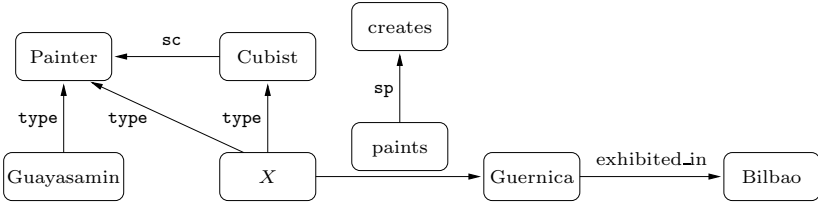 - $PExt$ is such that:

Fig. 1. Example of an RDF graph

- • $PExt(\text{paints}) = PExt(\text{creates}) = \{(\text{Guayasamin, Guernica})\}$,
- • $PExt(\text{exhibited\_in}) = \{(\text{Guernica, Bilbao})\}$,
- • $PExt(\texttt{type}) = \{(\text{Guayasamin, Cubist}), (\text{Guayasamin, Painter})\}$,
- • $PExt(\texttt{sp}) = \{(\text{paints, create})\} \cup \{(x,x) \mid x \in Prop\}$,
- • $PExt(\texttt{sc}) = \{(\text{Cubist, Painter}), (\text{Cubist, Cubist}), (\text{Painter, Painter})\}$,
- • $PExt(\texttt{dom}) = PExt(\texttt{range}) = \emptyset$.
- – $CExt$ is such that $CExt(\text{Cubist}) = CExt(\text{Painter}) = \{\text{Guayasamin}\}$.
- – Int is the identity mapping over $Res \cup Prop$.

Notice that in our interpretation the sets Res and $Prop$ are subsets of U, but in general, Res and $Prop$ can be arbitrary sets. Let G be the RDF graph of Fig. 1. By considering the function $A : B \rightarrow Res$ such that $A(X) = \text{Guayasamin}$, it can be shown that $\mathcal{I} \models G$, that is, $\mathcal{I}$ satisfies all the conditions to be a model of G.

In the interpretation $\mathcal{I}$, we use Guayasamin as a *witness* for the blank node X. Another model of G can use a different witness. For example consider the interpretation $\mathcal{I}' = (Res', Prop, Class, PExt', CExt', Int')$ where:

- – $Res' = Res \cup \{\text{Picasso}\}$.
- – $PExt'$ is such that:
    - • $PExt'(\text{paints}) = PExt'(\text{creates}) = \{(\text{Picasso, Guernica})\}$,
    - • $PExt'(\texttt{type}) = \{$ (Picasso, Cubist), (Picasso, Painter), (Guayasamin, Painter) $\}$,
    - • $PExt'$ is equal to $PExt$ in every other case.
- – $CExt'$ is such that $CExt'(\text{Cubist}) = \{\text{Picasso}\}$ and $CExt'(\text{Painter}) = \{\text{Picasso, Guayasamin}\}$
- – Int' is the identity mapping over $Res' \cup Prop$.

It can be shown that interpretation $\mathcal{I}'$ is also a model for G, but this time using Picasso as witness for the blank node X in G.   □

2.4   A Deductive System for RDFS

The notion of entailment has shown to be of fundamental importance for many tasks in the database context, and as such it also plays a fundamental role in the context of RDF. Indeed, this notion has been present since the beginning of the Semantic Web initiative. In this section, we study this concept in detail.

Given RDF graphs G_1 and G_2, we say that G_1 *entails* G_2, denoted by $G_1 \models G_2$, if for every interpretation $\mathcal{I}$ such that $\mathcal{I} \models G_1$, it holds that $\mathcal{I} \models G_2$. In [37], the authors showed that this entailment notion between RDF graphs is equivalent to the W3C normative notion of entailment [27], for the fragment of the RDFS vocabulary considered in this paper. In Table 1, we present a deductive system for this notion. This system was given in [37], and is based on a set of rules for $\models$ introduced in [27].

The first rule in Tab. 1 captures the semantics of blank nodes. In every rule (2)-(7), letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{Y}$, stand for *variables* to be replaced by actual terms. More formally, an *instantiation* of a rule (2)-(7) is a replacement of the variables occurring in the triples of the rule by elements of UB, such that all the

Table 1. RDFS inference rules

1. Existential:

$$\frac{G}{G'} \quad \text{for a homomorphism } h : G' \to G$$

2. Subproperty:

   (a) $\dfrac{(\mathcal{A},\text{sp},\mathcal{B})\ (\mathcal{B},\text{sp},\mathcal{C})}{(\mathcal{A},\text{sp},\mathcal{C})}$

   (b) $\dfrac{(\mathcal{A},\text{sp},\mathcal{B})\ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\mathcal{B},\mathcal{Y})}$

3. Subclass:

   (a) $\dfrac{(\mathcal{A},\text{sc},\mathcal{B})\ (\mathcal{B},\text{sc},\mathcal{C})}{(\mathcal{A},\text{sc},\mathcal{C})}$

   (b) $\dfrac{(\mathcal{A},\text{sc},\mathcal{B})\ (\mathcal{X},\text{type},\mathcal{A})}{(\mathcal{X},\text{type},\mathcal{B})}$

4. Typing:

   (a) $\dfrac{(\mathcal{A},\text{dom},\mathcal{B})\ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\text{type},\mathcal{B})}$

   (b) $\dfrac{(\mathcal{A},\text{range},\mathcal{B})\ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{Y},\text{type},\mathcal{B})}$

5. Implicit Typing:

   (a) $\dfrac{(\mathcal{A},\text{dom},\mathcal{B})\ (\mathcal{C},\text{sp},\mathcal{A})\ (\mathcal{X},\mathcal{C},\mathcal{Y})}{(\mathcal{X},\text{type},\mathcal{B})}$

   (b) $\dfrac{(\mathcal{A},\text{range},\mathcal{B})\ (\mathcal{C},\text{sp},\mathcal{A})\ (\mathcal{X},\mathcal{C},\mathcal{Y})}{(\mathcal{Y},\text{type},\mathcal{B})}$

6. Subproperty Reflexivity:

   (a) $\dfrac{(\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{A},\text{sp},\mathcal{A})}$

   (b) $\dfrac{(\mathcal{A},\text{sp},\mathcal{B})}{(\mathcal{A},\text{sp},\mathcal{A})\ (\mathcal{B},\text{sp},\mathcal{B})}$

   (c) $\dfrac{}{(p,\text{sp},p)}$ for $p \in \{\text{sp}, \text{sc}, \text{dom}, \text{range}, \text{type}\}$

   (d) $\dfrac{(\mathcal{A},p,\mathcal{X})}{(\mathcal{A},\text{sp},\mathcal{A})}$ for $p \in \{\text{dom}, \text{range}\}$

7. Subclass Reflexivity:

   (a) $\dfrac{(\mathcal{A},\text{sc},\mathcal{B})}{(\mathcal{A},\text{sc},\mathcal{A})\ (\mathcal{B},\text{sc},\mathcal{B})}$

   (b) $\dfrac{(\mathcal{X},p,\mathcal{A})}{(\mathcal{A},\text{sc},\mathcal{A})}$ for $p \in \{\text{dom}, \text{range}, \text{type}\}$
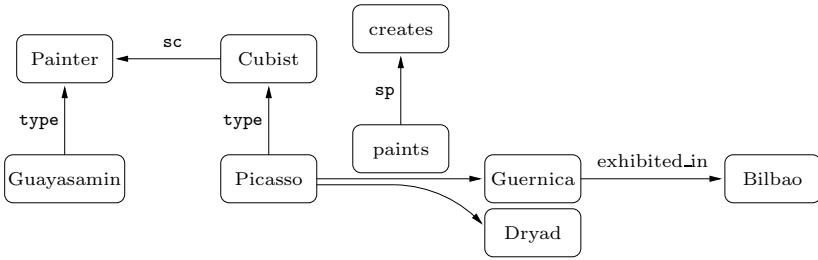
Fig. 2. RDF graph from which we can deduce the graph in Fig. 1

triples obtained after the replacement are well formed RDF triples, that is, not assigning blank nodes to variables in predicate positions.

An *application* of a rule to a graph G is defined as follows. For rule (1), if h is a homomorphism from G' to G, then G' is the result of an application of rule (1) to G. If r is any of the rules (2)-(7), and there is an instantiation $\frac{R}{R'}$ of r such that $R \subseteq G$, then the graph $G' = G \cup R'$ is the result of an application of r to G. We say that a graph G' is *deduced from* G, if G' is obtained from G by successively applying the rules in Tab. 1.

In [37], the authors proved that the set of rules in Tab. 1 is sound and complete for the inference problem for the fragment of RDFS consisting of the reserved words `sc`, `sp`, `range`, `dom` and `type`. That is, it captures the semantics of the normative RDF specification when one focuses on the fragment of the RDFS vocabulary considered in this paper.

Theorem 1 (Soundness and completeness [37]). *Let G and H be RDF graphs, then $G \models H$ iff H is deduced from G by applying rules in Tab. 1.*

It is worth mentioning that the set of rules presented in [27] is not complete for $\models$ (this was pointed out by Marin in [35]). The problem with the system proposed in [27] is that a blank node X can be implicitly used as a property in triples like $(a, \mathtt{sp}, X)$, $(X, \mathtt{dom}, b)$, and $(X, \mathtt{range}, c)$. This problem was solved in [37] by following the approach proposed by Marin [35]. In fact, the rules (5a)-(5b) were added to the system given in [27] to deal with this problem.

Example 2. Let G be the graph in Fig. 1 and G' the graph in Fig. 2. Notice that the triples (Picasso, `type`, Cubist) and (Cubist, `sc`, Painter) belong to G'. Thus, by using rule (3b) we obtain that $G'' = G' \cup \{(\text{Picasso}, \mathtt{type}, \text{Painter})\}$ is deduced from G'. Moreover, if we consider a homomorphism h such that $h(X) = \text{Picasso}$, then we have that $h(G) \subseteq h(G'')$, and, thus, applying rule (1) we conclude that G can be deduced from G''. Therefore, the graph G can be deduced from G' by successively applying rules (3b) and (1). Hence, from Theorem 1 we have that every model of G' is also a model of G, i.e. $G' \models G$.   □

In [37], the authors showed that the deductive system of Tab. 1 can be simplified by imposing some syntactic restrictions on RDF graphs. The most simple case

is obtained when G and H are graphs that do not have blank nodes, and do not mention RDFS vocabulary. In that case, the entailment relation $G \models H$ is reduced to just testing whether $H \subseteq G$. On the other hand, if G and H are RDF graphs that do not mention RDFS vocabulary (but possibly blank nodes), then $G \models H$ if and only if H can be obtained from G by using rule (1), that is, if and only if there exists a homomorphism $h : H \to G^2$. Another important simplification is obtained if one forbids the presence of *reflexive triples*. A triple t is reflexive if t is of the form $(x, \mathtt{sp}, x)$ or $(x, \mathtt{sc}, x)$ for $x \in UB$. We formalize two of these special cases in the following proposition.

Proposition 1 ([37])

1. *If G and H are RDF graphs that do not mention RDFS vocabulary, then $G \models H$ iff there exists a homomorphism $h : H \to G$.*
2. *If G and H are RDF graphs that have neither blank nodes nor reflexive triples, then $G \models H$ iff H can be deduced from G by using rules (2)-(4).*

In the following sections, we study the fundamental problem of querying RDF data. There is no yet consensus in the Semantic Web community on how to define a query language for RDF that includes all the features of the RDF data model, in particular blank nodes and the RDFS vocabulary. The specification of SPARQL, the standard language for RDF, currently considers RDF data without RDFS vocabulary and with no special semantics for blank nodes. Thus, we study SPARQL in the next sections focusing on ground RDF graphs with no RDFS vocabulary. In Section 6.3, we explore the possibility of having an RDF query language capable of dealing with the special semantics of the RDFS vocabulary.

3   The RDF Query Language SPARQL

In 2004, the RDF Data Access Working Group, part of the W3C Semantic Web Activity, released a first public working draft of a query language for RDF, called SPARQL [45][3]. Since then, SPARQL has been rapidly adopted as the standard for querying Semantic Web data. In January 2008, SPARQL became a W3C Recommendation.

RDF is a directed labeled graph data format and, thus, SPARQL is essentially a graph-matching query language. SPARQL queries are composed by three parts. The *pattern matching part*, which includes several interesting features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering values of possible matchings, and the possibility of choosing the data source to be matched by a pattern. The *solution modifiers*, which once the output of the pattern has been computed (in the form of a table of values of variables), allow to modify these values applying classical operators like projection, distinct, order and limit. Finally, the *output* of a SPARQL query can be of different types:

---

[2] Notice that this result is also a corollary of [16].

[3] The name SPARQL is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language*.

yes/no queries, selections of values of the variables which match the patterns, construction of new RDF data from these values, and descriptions of resources.

The definition of a formal semantics for SPARQL has played a key role in the standardization process of this query language. Although taken one by one the features of SPARQL are intuitive and simple to describe and understand, it turns out that the combination of them makes SPARQL into a complex language. Reaching a consensus in the W3C standardization process about a formal semantics for SPARQL was not an easy task. The initial efforts to define SPARQL were driven by use cases, mostly by specifying the expected output for particular example queries. In fact, the interpretations of examples and the exact outcomes of cases not covered in the initial drafts of the SPARQL specification, were a matter of long discussions in the W3C mailing lists. In [40], the authors presented one of the first formalizations of a semantics for a fragment of the language. Currently, the official specification of SPARQL [45], endorsed by the W3C, formalizes a semantics based on [40].

A formalization of a semantics for SPARQL is beneficial for several reasons, including to serve as a tool to identify and derive relations among the constructors that stay hidden in the use cases, identify redundant and contradicting notions, to drive and help the implementation of query engines, and to study the complexity, expressiveness, and further natural database questions like rewriting and optimization. In this section, we present a streamlined version of the core fragment of SPARQL with precise algebraic syntax and a formal compositional semantics based on [40].

One of the delicate issues in the definition of a semantics for SPARQL is the treatment of *optional matching* and incomplete answers. The idea behind optional matching is to allow information to be added if the information is available in the data source, instead of just failing to give an answer whenever some part of the pattern does not match. This feature of optional matching is crucial in Semantic Web applications, and more specifically in RDF data management, where it is assumed that every application have only partial knowledge about the resources being managed. The semantics of SPARQL is formalized by using *partial mappings* between variables in the patterns and actual values in the RDF graph being queried. This formalization allows one to deal with partial answers in a clean way, and is based on the extension of some classical relational algebra operators to work over sets of partial mappings.

A SPARQL query is of the form *head* ← *body*, where the *body* of the query is a complex RDF graph pattern expression that may include RDF triples with variables, conjunctions, disjunctions, optional parts and constraints over the values of the variables, and the *head* of the query is an expression that indicates how to construct the answer to the query. The evaluation of a query Q against an RDF graph G is done in two steps: the body of Q is matched against G to obtain a set of bindings for the variables in the body, and then using the information on the head of Q, these bindings are processed applying classical relational operators (projection, distinct, etc.) to produce the answer to the query.

It should be noticed that the normative specification of SPARQL [45] is defined over RDF graphs without RDFS vocabulary, and not considering the special semantics of blank nodes. In this section, we work over the same setting.

3.1 Syntax and Semantics of SPARQL Graph Patterns

We first concentrate on the body of SPARQL queries, i.e. in the graph pattern matching facility.

The official syntax of SPARQL [45] considers operators OPTIONAL, UNION, FILTER, and *concatenation* via a point symbol (.), to construct graph pattern expressions. The syntax also considers { } to group patterns, and some implicit rules of precedence and association. For example, the point symbol (.) has precedence over OPTIONAL, and OPTIONAL is left associative. In order to avoid ambiguities in the parsing of expressions, we present the syntax of SPARQL graph patterns in a more traditional algebraic formalism, using binary operators AND (.), UNION (UNION), OPT (OPTIONAL), and FILTER (FILTER). We fully parenthesize expressions making explicit the precedence and association of operators.

Assume the existence of a set of variables V disjoint from U. A SPARQL graph pattern expression is defined recursively as follows:

1. A tuple from $(U \cup V) \times (U \cup V) \times (U \cup V)$ is a graph pattern (a *triple pattern*).
2. If P_1 and P_2 are graph patterns, then expressions $(P_1$ AND $P_2)$, $(P_1$ OPT $P_2)$, and $(P_1$ UNION $P_2)$ are graph patterns (*conjunction graph pattern*, *optional graph pattern*, and *union graph pattern*, respectively).
3. If P is a graph pattern and R is a SPARQL *built-in* condition, then the expression $(P$ FILTER $R)$ is a graph pattern (a *filter graph pattern*).

A SPARQL *built-in* condition is constructed using elements of the set $U \cup V$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), inequality symbols ($<$, $\leq$, $\geq$, $>$), the equality symbol ($=$), unary predicates like bound, isBlank, and isIRI, plus other features (see [45] for a complete list). In this paper, we restrict to the fragment where the built-in condition is a Boolean combination of terms constructed by using $=$ and bound, that is:

1. If $?X, ?Y \in V$ and $c \in U$, then bound($?X$), $?X = c$ and $?X =?Y$ are built-in conditions.
2. If R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions.

Let P be a SPARQL graph pattern. In the rest of the paper, we use var(P) to denote the set of variables occurring in P. In particular, if t is a triple pattern, then var(t) denotes the set of variables occurring in the components of t. Similarly, for a built-in condition R, we use var(R) to denote the set of variables occurring in R.

To define the semantics of SPARQL graph pattern expressions, we need to introduce some terminology. A *mapping* μ from V to U is a partial function

$\mu : V \rightarrow U$. Abusing notation, for a triple pattern t we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, denoted by $\mathrm{dom}(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* when for all $?X \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$, it is the case that $\mu_1(?X) = \mu_2(?X)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. Intuitively, μ_1 and μ_2 are compatibles if μ_1 *can be extended* with μ_2 to obtain a new mapping, and vice versa. Note that two mappings with disjoint domains are always compatible, and that the empty mapping $\mu_\emptyset$ (i.e. the mapping with empty domain) is compatible with any other mapping.

Let Ω_1 and Ω_2 be sets of mappings. We define the join of, the union of and the difference between Ω_1 and Ω_2 as [40]:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1, \mu_2 \text{ are compatible mappings}\},$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},$$
$$\Omega_1 \smallsetminus \Omega_2 = \{\mu \in \Omega_1 \mid \text{ for all } \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\}.$$

Based on the previous operators, we define the left outer-join as:

$$\Omega_1 \,\bowtie\!\!\!\!\!\!\!\!\!\,\!\!\!\!\!\!\!\!\!\,\, \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).$$

Intuitively, $\Omega_1 \,\bowtie\!\!\!\!\!\!\!\,\, \Omega_2$ is the set of mappings that result from extending mappings in Ω_1 with their compatible mappings in Ω_2, and $\Omega_1 \smallsetminus \Omega_2$ is the set of mappings in Ω_1 that cannot be extended with any mapping in Ω_2. The operation $\Omega_1 \cup \Omega_2$ is the usual set theoretical union. A mapping μ is in $\Omega_1 \,\bowtie\!\!\!\!\!\!\!\,\, \Omega_2$ if it is the extension of a mapping of Ω_1 with a compatible mapping of Ω_2, or if it belongs to Ω_1 and cannot be extended with any mapping of Ω_2. These operations resemble relational algebra operations over sets of mappings (partial functions) [52].

We are ready to define the semantics of graph pattern expressions as a function $[\![\cdot]\!]_G$ which takes a pattern expression and returns a set of mappings. We follow the approach in [23] defining the semantics as the set of mappings that matches the graph G. For the sake of readability, the semantics of filter expressions is presented in a separate definition.

Definition 2. *The* evaluation *of a graph pattern P over an RDF graph G, denoted by $[\![P]\!]_G$, is defined recursively as follows:*

1. *if P is a triple pattern t, then $[\![P]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{var}(t) \text{ and } \mu(t) \in G\}$.*
2. *if P is $(P_1 \text{ AND } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.*
3. *if P is $(P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \,\bowtie\!\!\!\!\!\!\!\,\, [\![P_2]\!]_G$.*
4. *if P is $(P_1 \text{ UNION } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$.*

The idea behind the OPT operator is to allow for *optional matching* of patterns. Consider pattern expression $(P_1 \text{ OPT } P_2)$ and let μ_1 be a mapping in $[\![P_1]\!]_G$. If there exists a mapping $\mu_2 \in [\![P_2]\!]_G$ such that μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ belongs to $[\![(P_1 \text{ OPT } P_2)]\!]_G$. But if no such a mapping μ_2 exists, then μ_1 belongs to $[\![(P_1 \text{ OPT } P_2)]\!]_G$. Thus, operator OPT allows information to be added to a mapping μ if the information is available, instead of just rejecting μ whenever some part of the pattern does not match.

The semantics of filter expressions goes as follows. Given a mapping μ and a built-in condition R, we say that μ satisfies R, denoted by $\mu \models R$, if:

1. R is bound($?X$) and $?X \in \text{dom}(\mu)$;
2. R is $?X = c$, $?X \in \text{dom}(\mu)$ and $\mu(?X) = c$;
3. R is $?X = ?Y$, $?X \in \text{dom}(\mu)$, $?Y \in \text{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$;
4. R is $(\neg R_1)$, R_1 is a built-in condition, and it is not the case that $\mu \models R_1$;
5. R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in conditions, and $\mu \models R_1$ or $\mu \models R_2$;
6. R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in conditions, $\mu \models R_1$ and $\mu \models R_2$.

Definition 3. *Given an RDF graph G and a filter expression $(P$ FILTER $R)$,*

$$[\![(P \text{ FILTER } R)]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu \models R\}.$$

In the normative semantics of SPARQL [45], there is an additional feature of graph patterns that allows to query several different RDF graphs with a single pattern. This is accomplished with the GRAPH operator that allows to dynamically change the graph being used in the evaluation of a pattern. For the sake of readability, we do not include here the GRAPH operator. We refer the reader to [42] for a formalization of SPARQL graph patterns including GRAPH, and to [9] for some tutorial material.

In the rest of the paper, we usually represent sets of mappings as tables where each row represents a mapping in the set. We label every row with the name of a mapping, and every column with the name of a variable. If a mapping is not defined for some variable, then we simply leave empty the corresponding position. For instance, the table

	$?X$	$?Y$	$?Z$	$?V$	$?W$
$\mu_1 :$	a	b			
$\mu_2 :$		c			d
$\mu_3 :$			e		

represents the set $\Omega = \{\mu_1, \mu_2, \mu_3\}$ where

- dom(μ_1) = $\{?X, ?Y\}$, $\mu_1(?X) = a$, and $\mu_1(?Y) = b$,
- dom(μ_2) = $\{?Y, ?W\}$, $\mu_2(?Y) = c$, and $\mu_2(?W) = d$,
- dom(μ_3) = $\{?Z\}$, and $\mu_3(?Z) = e$.

Sometimes we use notation $\{\{?X \to a, ?Y \to b\}, \{?Y \to c, ?W \to d\}, \{?Z \to e\}\}$ for a set of mappings as the one above.

Example 3. Consider an RDF graph G storing information about professors in a university:

$G = \{$ (B_1, name,   paul),   (B_1, phone,   777-3426),
         (B_2, name,   john),   (B_2, email,   john@acd.edu),
         (B_3, name,   george),   (B_3, webPage, www.george.edu),
         (B_4, name,   ringo),   (B_4, email,   ringo@acd.edu),
         (B_4, webPage, www.starr.edu),   (B_4, phone,   888-4537) $\}$

The following are graph pattern expressions and their evaluations over G:

- $P_1 = ((?A, \text{email}, ?E) \text{ AND } (?A, \text{webPage}, ?W))$. Then

$[\![P_1]\!]_G =$

	?A	?E	?W
μ_1 :	B_4	ringo@acd.edu	www.starr.edu

- $P_2 = ((?A, \text{email}, ?E) \text{ OPT } (?A, \text{webPage}, ?W))$. Then

$[\![P_2]\!]_G =$

	?A	?E	?W
μ_1 :	B_2	john@acd.edu	
μ_2 :	B_4	ringo@acd.edu	www.starr.edu

- $P_3 = (((?A, \text{name}, ?N) \text{ OPT } (?A, \text{email}, ?E)) \text{ OPT } (?A, \text{webPage}, ?W))$. Then

$[\![P_3]\!]_G =$

	?A	?N	?E	?W
μ_1 :	B_1	paul		
μ_2 :	B_2	john	john@acd.edu	
μ_3 :	B_3	george		www.george.edu
μ_4 :	B_4	ringo	ringo@acd.edu	www.starr.edu

- $P_4 = ((?A, \text{name}, ?N) \text{ OPT } ((?A, \text{email}, ?E) \text{ OPT } (?A, \text{webPage}, ?W)))$. Then

$[\![P_4]\!]_G =$

	?A	?N	?E	?W
μ_1 :	B_1	paul		
μ_2 :	B_2	john	john@acd.edu	
μ_3 :	B_3	george		
μ_4 :	B_4	ringo	ringo@acd.edu	www.starr.edu

Notice the difference between $[\![P_2]\!]_G$ and $[\![P_3]\!]_G$. These two examples show that $[\![((A \text{ OPT } B) \text{ OPT } C)]\!]_G \neq [\![(A \text{ OPT } (B \text{ OPT } C))]\!]_G$ in general.

- $P_5 = ((?A, \text{name}, ?N) \text{ AND } ((?A, \text{email}, ?E) \text{ UNION } (?A, \text{webPage}, ?W)))$. Then

$[\![P_5]\!]_G =$

	?A	?N	?E	?W
μ_1 :	B_2	john	john@acd.edu	
μ_2 :	B_3	george		www.george.edu
μ_3 :	B_4	ringo	ringo@acd.edu	
μ_4 :	B_4	ringo		www.starr.edu

- $P_6 = (((?A, \text{name}, ?N) \text{ OPT } (?A, \text{phone}, ?P)) \text{ FILTER } ?N = \text{paul})$. Then

$[\![P_6]\!]_G =$

	?A	?N	?P
μ_1 :	B_1	paul	777-3426

$\square$

Simple algebraic properties. We say that two graph patterns P_1 and P_2 are *equivalent*, denoted by $P_1 \equiv P_2$, if $[\![P_1]\!]_G = [\![P_2]\!]_G$ for every RDF graph G. The following simple lemma states some simple algebraic properties of AND and UNION operators. These properties are direct consequence of the semantics of AND and UNION, both based on set-theoretical union.

Lemma 1 ([40]). *The operators* AND *and* UNION *are associative and commutative and the operator* AND *distribute over* UNION*. That is, if P_1, P_2 and P_3 are graph patterns, then it holds that:*

 - $(P_1 \text{ AND } P_2) \equiv (P_2 \text{ AND } P_1)$
 - $(P_1 \text{ UNION } P_2) \equiv (P_2 \text{ UNION } P_1)$
 - $(P_1 \text{ AND } (P_2 \text{ AND } P_3)) \equiv ((P_1 \text{ AND } P_2) \text{ AND } P_3)$
 - $(P_1 \text{ UNION } (P_2 \text{ UNION } P_3)) \equiv ((P_1 \text{ UNION } P_2) \text{ UNION } P_3)$
 - $(P_1 \text{ AND } (P_2 \text{ UNION } P_3)) \equiv ((P_1 \text{ AND } P_2) \text{ UNION } (P_1 \text{ AND } P_3))$

The above lemma permits us to avoid parenthesis when writing sequences of either AND operators or UNION operators. This is consistent with the definitions of *Group Graph Pattern* and *Union Graph Pattern* in [45]. We use Lemma 1 to simplify the notation in the following sections.

3.2   Query Result Forms

The normative specification of SPARQL [45] considers four *query forms*. These query forms use the mappings obtained after the evaluation of a graph pattern to construct result sets or RDF graphs. The query forms are: (1) SELECT, that performs a *projection* over a set of variables in the evaluation of a graph pattern, (2) CONSTRUCT, that returns an RDF graph constructed by substituting variables in a *template*, (3) ASK, that returns a truth value indicating whether the evaluation of a graph pattern produces at least one mapping, and (4) DESCRIBE, that returns an RDF graph that describes the resources found. In this paper, we only consider the SELECT query form. We refer the reader to [42] for a formalization of the remaining query forms.

Given a mapping $\mu : V \to U$ and a set of variables $W \subseteq V$, the *restriction* of μ to W, denoted by $\mu_{|W}$, is a mapping such that $\text{dom}(\mu_{|W}) = \text{dom}(\mu) \cap W$ and $\mu_{|W}(?X) = \mu(?X)$ for every $?X \in \text{dom}(\mu) \cap W$.

Definition 4. *A SPARQL SELECT query is a tuple (W, P), where P is a graph pattern and W is a set of variables such that $W \subseteq \text{var}(P)$. The answer of (W, P) over an RDF graph G, denoted by $[\![(W, P)]\!]_G$, is the set of mappings:*

$$[\![(W, P)]\!]_G = \{\mu_{|W} \mid \mu \in [\![P]\!]_G\}.$$

Example 4. Consider the RDF graph G and the graph pattern P_3 in Example 3. Then we have that:

$$[\![(\{?N, ?E\}, P_3)]\!]_G =$$

	?N	?E
μ_1 :	paul	
μ_2 :	john	john@acd.edu
μ_3 :	george	
μ_4 :	ringo	ringo@acd.edu

□

In the following sections, we study some fundamental issues regarding the query language SPARQL. The first of that issues is the complexity of the evaluation problem for SPARQL. In Section 4, we focus on studying the complexity of the evaluation problem for SPARQL graph patterns. Then in Section 5, we consider SPARQL SELECT queries to compare the expressive powers of SPARQL and the Relational Algebra.

4   Complexity and Optimization of SPARQL

A fundamental issue in every query language is the complexity of query evaluation and, in particular, what is the influence of each component of the language in this complexity.

In this section, we present a thorough study of the complexity of the evaluation of SPARQL graph patterns based on [40]. In this study, we consider several fragments of SPARQL built incrementally, and present complexity results for each such fragment. Among other results, we show that the complexity of the evaluation problem for general SPARQL graph patterns is PSPACE-complete [40], and that this high complexity is obtained as a consequence of unlimited use of nested optional parts.

Given the high complexity of the evaluation problem for general SPARQL graph patterns, an important question is whether one can find interesting classes of patterns where the query evaluation problem can be solved more efficiently. In [40,41], the authors identified a large class of patterns with the previous characteristic that is defined by a simple and natural syntactic restriction. This class is obtained by forbidding a special form of interaction between variables appearing in optional parts. Patterns satisfying this condition are called *well-designed* [40,41]. Well-designed patterns form a natural fragment of SPARQL that is very common in practice, and has several interesting features. On the one hand, the complexity of the evaluation problem for well-designed patterns is considerably lower, namely coNP-complete. On the other hand, the property of being well designed has important consequences for the optimization of SPARQL queries. We present some rewriting rules for well-designed patterns whose application may have a considerable impact in the cost of evaluating SPARQL queries, and prove the existence of a normal form for well-designed patterns based on the application of these rewriting rules.

4.1   Complexity of Evaluating Graph Pattern Expressions

In this section, we review some the results in the literature regarding the complexity of evaluating SPARQL graph pattern expressions. The first study about

this problem was published in [40], and some refinements of the complexity results of [40] were presented in [47]. This section focuses on the complexity results proved in these two papers.

As is customary when studying the complexity of the evaluation problem for a query language [51], we consider its associated decision problem. We denote this problem by EVALUATION and we define it as follows:

> INPUT   : An RDF graph G, a graph pattern P and a mapping μ.
> QUESTION : Is $\mu \in [\![P]\!]_G$?

It is important to notice that the evaluation problem that we study considers the mapping as part of the input. That is, we study the complexity by measuring how difficult it is to verify whether a given mapping is a solution for a pattern evaluated over an RDF graph. This is the standard *decision* problem considered when studying the complexity of a query language [51], as opposed to the *computation* problem of actually listing the set of solutions (finding all the mappings). To focus on the associated decision problem allows us to obtain a fine grained analysis of the complexity of the evaluation problem, classifying the complexity for different fragments of SPARQL in terms of standard complexity classes. Also notice that the pattern and the graph are both input for EVALUATION. Thus, we study the *combined complexity* of the query language [51].

We start this study by considering the fragment consisting of graph pattern expressions constructed by using only AND and FILTER operators. This simple fragment is interesting as it does not use the two most complicated operators in SPARQL, namely UNION and OPT. Given an RDF graph G, a graph pattern P in this fragment and a mapping μ, it is possible to efficiently check whether $\mu \in [\![P]\!]_G$ by using the following simple algorithm [40]. First, for each triple t in P, verify whether $\mu(t) \in G$. If this is not the case, then return *false*. Otherwise, by using a bottom-up approach, verify whether the expression generated by instantiating the variables in P according to μ satisfies the FILTER conditions in P. If this is the case, then return *true*, else return *false*.

Theorem 2. EVALUATION *can be solved in time $O(|P| \cdot |D|)$ for graph pattern expressions constructed by using only* AND *and* FILTER *operators.*

We continue this study by adding the UNION operator to the AND-FILTER fragment. It is important to notice that the inclusion of UNION in SPARQL is one of the most controversial issues in the definition of this language. The following theorem proved in [40], shows that the inclusion of the UNION operator makes the evaluation problem for SPARQL considerably harder.

Theorem 3 ([40]). EVALUATION *is NP-complete for graph pattern expressions constructed by using only* AND*,* FILTER *and* UNION *operators.*

In [47], the authors strengthen the above result by showing that the complexity of evaluating graph pattern expressions constructed by using only AND and UNION operators is already NP-hard. Thus, we have the following result.

Theorem 4 ([47]). EVALUATION *is NP-complete for graph pattern expressions constructed by using only* AND *and* UNION *operators.*

We now consider the OPT operator, which is the most involved operator in graph pattern expressions and, definitively, the most difficult to define. The following theorem proved in [40] shows that when considering all the operators in SPARQL graph patterns, the evaluation problem becomes considerably harder.

Theorem 5 ([40]). EVALUATION *is PSPACE-complete.*

To prove the PSPACE-hardness of EVALUATION, the authors show in [40] how to reduce in polynomial time the quantified boolean formula problem (QBF) to EVALUATION. An instance of QBF is a quantified propositional formula φ of the form:

$$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \cdots \forall x_m \exists y_m \, \psi,$$

where ψ is a quantifier-free formula of the form $C_1 \wedge \cdots \wedge C_n$, with each C_i ($i \in \{1, \ldots, n\}$) being a disjunction of literals, that is, a disjunction of propositional variables x_i and y_j, and negations of propositional variables. Then the problem is to verify whether φ is valid. It is known that QBF is PSPACE-complete [22]. In the encoding presented in [40], the authors use a fixed RDF graph G and a fixed mapping μ. Then they encode formula φ with a pattern P_φ that uses nested OPT operators to encode the *quantifier alternation* of φ, and a graph pattern without OPT to encode the satisfiability of formula ψ. By using a similar idea, it is shown in [47] how to encode formulas φ and ψ by using only the OPT operator, thus strengthening Theorem 5.

Theorem 6 ([47]). EVALUATION *is PSPACE-complete for graph pattern expressions constructed by using only the* OPT *operator.*

When verifying whether $\mu \in [\![P]\!]_G$, it is natural to assume that the size of P is considerably smaller than the size of G. This assumption is very common when studying the complexity of a query language. In fact, it is named *data complexity* in the database literature [51], and it is defined as the complexity of the evaluation problem for a fixed query. More precisely, for the case of SPARQL, given a graph pattern expression P, the evaluation problem for P, denoted by EVALUATION(P), has as input an RDF graph G and a mapping μ, and the problem is to verify whether $\mu \in [\![P]\!]_G$.

Theorem 7 ([40]). EVALUATION(P) *is in LOGSPACE for every graph pattern expression P.*

An important question is whether one can find interesting classes of graph patterns, constructed by imposing simple and natural syntactic restrictions, such that one can obtain lower complexity bounds for the evaluation problem on that classes. In the following section, we introduce a first such restriction.

4.2   A Simple Normal Form for Graph Patterns

We say that a pattern P is UNION-free if P is constructed by using only opera-
tors AND, OPT and FILTER. In [40], the authors proved the following normal-
form result.

Proposition 2 ([40]). *Every graph pattern P is equivalent to a pattern of the
form:*

$$(P_1 \quad \text{UNION} \quad P_2 \quad \text{UNION} \quad P_3 \quad \text{UNION} \quad \cdots \quad \text{UNION} \quad P_n), \qquad (1)$$

where each P_i $(1 \leq i \leq n)$ is UNION-free.

Notice that we omit the parenthesis in the expression (1) given the associativity
of UNION. We say that a graph pattern is in UNION *normal form* if the pattern
is in the form (1).[4]
   The following result shows that for graph patterns in UNION normal form that
do not use the OPT operator, the evaluation problem can be solved efficiently.
It is a direct consequence of Theorem 2.

Corollary 1. EVALUATION *can be solved in time $O(|P| \cdot |G|)$ for graph patterns
in* UNION *normal form constructed by using only* AND, FILTER, *and* UNION
operators.

We have managed to lower the complexity of the AND-FILTER-UNION frag-
ment by imposing a simple normal form. However, Theorem 6 implies that when
the OPT operator is allowed in graph patterns, the complexity of the evaluation
problem is PSPACE-hard even if we restrict to patterns in UNION normal form.
In the following section, we introduce a simple and natural syntactic condition
that patterns usually satisfy in practice. Under this condition, the complexity
of the evaluation of graph patterns in UNION normal form is lower even if the
OPT operator is allowed.

4.3   Well-Designed Graph Patterns

The exact semantics of graph pattern expressions has been extensively discussed
on the mailing list of the W3C. One of the most delicate issues in the definition of
a semantics for graph pattern expressions is the semantics of the OPT operator.
As we have mentioned before, the idea behind the OPT operator is to allow for
optional matching of patterns, that is, to allow information to be added if it
is available, instead of just rejecting whenever some part of a pattern does not

---

[4] In the conference version of [40], the proof of the existence of a
UNION normal form used the equivalence $(P_1 \text{ OPT } (P_2 \text{ UNION } P_3)) \equiv$
$((P_1 \text{ OPT } P_2) \text{ UNION } (P_1 \text{ OPT } P_3))$ (see Proposition 1 in [40]). Unfortunately,
this rule does not hold in general [47]. In the *errata* of [40] (that can be downloaded
from `http://www.ing.puc.cl/~ marenas/publications/errata-iswc06.pdf`), the
authors provide a proof of Proposition 2 without using this rule.

match. However, this intuition fails in some simple, but unnatural, examples. For instance, consider the graph pattern:

$$P = ((?X, \text{name}, \text{john}) \text{ OPT } ((?Y, \text{name}, \text{mick}) \text{ OPT } (?X, \text{email}, ?Z))). \quad (2)$$

What is unnatural about graph pattern P is the fact that $(?X, \text{email}, ?Z)$ is giving optional information for $(?X, \text{name}, \text{john})$, but in P appears as giving optional information for $(?Y, \text{name}, \text{mick})$. For example, $(B_2, \text{name}, \text{john})$ and $(B_2, \text{email}, \text{john@ac.edu})$ are triples in the graph G of Example 3, but the evaluation of P results in the set $\{\{?X \to B_2\}\}$ (since $[\![(?Y, \text{name}, \text{mick})]\!]_G = \emptyset$) without giving information about the email of john.

A careful examination of the examples that produce conflicts reveals a common pattern: A graph pattern P mentions an expression $P' = (P_1 \text{ OPT } P_2)$ and a variable $?X$ occurring both inside P_2 and outside P' but not occurring in P_1. In general, graph pattern expressions satisfying this condition are not natural.

In [40], the authors considered a special class of patterns that they called *well-designed patterns*, obtained by forbidding the form of interaction between variables appearing in optional parts discussed above. To present the formal definition of well-designed patterns, we need to introduce some terminology. We say that a graph pattern Q is *safe* if for every sub-pattern $(P \text{ FILTER } R)$ of Q, it holds that $\text{var}(R) \subseteq \text{var}(P)$. This safety condition is a usual restriction in many database query languages.

Definition 5 ([40]). *A* UNION-*free graph pattern P is well designed if P is safe and, for every sub-pattern $P' = (P_1 \text{ OPT } P_2)$ of P and for every variable $?X$ occurring in P, the following condition holds:*

if $?X$ occurs both inside P_2 and outside P', then it also occurs in P_1.

For instance, pattern (2) above is not well designed. One can extend Definition 5 to patterns in UNION normal form; a pattern $(P_1 \text{ UNION } P_2 \text{ UNION } \cdots \text{ UNION } P_n)$ is well designed if every P_i $(1 \le i \le n)$ is a UNION-free well-designed graph pattern.

It should be noticed that to prove the PSPACE lower bound of Theorem 5, it is used in [40] a graph pattern that is not well designed. Thus, an immediate question is whether the complexity of evaluating well-designed graph pattern expressions is lower than in the general case. In [41] (the extended version of [40]), the authors showed that this is indeed the case, in fact, they proved a coNP upper bound for the case of well-designed graph patterns. In [40,41], the authors also considered the problem of optimizing well-designed graph patterns. Since the beginning of the relational model, several techniques for optimizing the evaluation of relational algebra expressions have been developed. In fact, one of the reasons why relational algebra is so extensively used to implement SQL is the existence of simple reordering and optimization rules for this language. Unfortunately, the development of this type of rules for SPARQL is limited by the presence of the OPT operator. However, it was shown in [40,41] that well-designed patterns are suitable for reordering and optimization, demonstrating the significance of this

class of queries from the practical point of view. In the rest of this section, we review some of the results in [40,41] regarding well-designed patterns.

We note first that the property of being well-designed can be checked efficiently by a straightforward procedure. Let P be a pattern. Then for every sub-pattern P' of P of the form $(P_1 \text{ OPT } P_2)$, we construct three sets: sets V_{P_1} and V_{P_2}, containing the variables occurring in P_1 and P_2, respectively, and set $O_{P'}$ containing the variables that occur *outside* P'. To construct V_{P_1}, we collect variables by making a bottom-up traversal of the sub-patterns of P_1. We repeat this procedure in P_2 to construct V_{P_2}. To construct $O_{P'}$, we make a bottom-up traversal of the entire pattern P, but not taking into consideration P'. Having these three sets, we check whether $V_{P_2} \cap O_{P'} \subseteq V_{P_1}$, that is, we check whether every variable that occurs inside P_2 and outside P' also occurs inside P_1, which is exactly the well-designed condition. We must repeat this test for every OPT sub-pattern of P. Notice that the test for every OPT sub-pattern takes linear time in the size of P, and then, the entire process takes time proportional to the size of P times the number of OPT sub-patterns of P. We can then state the following proposition:

Proposition 3 ([41]). *Testing if a pattern P is well designed can be done in time $O(|P|^2)$.*

4.4   Complexity of Evaluating Well-Designed Patterns

Intuitively, if we *delete* some optional parts of a pattern P to obtain a new pattern P', the mappings in the evaluation of P' over a graph G could not be more informative than the mappings in the evaluation of P over G. That is, the optional matchings of a pattern must only serve to *extend* solutions with new information, but not to reject solutions if some information is not provided. In [41], the authors showed that the intuition is indeed correct for the case of well-designed graph patterns. In this section, we present the formalization of this intuition given in [41], and use it to develop a characterization of the evaluation of well-designed graph patterns.

We say that a mapping μ is *subsumed* by a mapping μ', denoted by $\mu \sqsubseteq \mu'$, if μ and μ' are compatible and $\text{dom}(\mu) \subseteq \text{dom}(\mu')$. That is, μ is subsumed by μ' if μ agrees with μ' in every variable for which μ is defined. For sets of mappings Ω and Ω', we write $\Omega \sqsubseteq \Omega'$ if for every mapping $\mu \in \Omega$, there exists a mapping $\mu' \in \Omega'$ such that $\mu \sqsubseteq \mu'$.

We say that a pattern P' is a *reduction* of a pattern P, if P' can be obtained from P by replacing a sub-formula $(P_1 \text{ OPT } P_2)$ of P by P_1, that is, if P' is obtained by deleting some optional part of P. For example,

$$P' = (t_1 \text{ AND } (t_2 \text{ OPT } (t_3 \text{ AND } t_4)))$$

is a reduction of

$$P = ((t_1 \text{ OPT } t_2) \text{ AND } (t_2 \text{ OPT } (t_3 \text{ AND } t_4)))$$

since P' can be obtained from P by replacing $(t_1$ OPT $t_2)$ by t_1. The reflexive and transitive closure of the reduction relation is denoted by $\trianglelefteq$. Thus, for example, if $P'' = (t_1$ AND $t_2)$, then $P'' \trianglelefteq P$ since P'' is a reduction of P' and P' is a reduction of P. We note that if $P' \trianglelefteq P$ and P is well designed, then P' is well designed.

We can now state the result that formalizes the intuition mentioned at the beginning of this section.

Lemma 2 ([41]). *Let P be a* UNION-*free well-designed graph pattern, and P' a pattern such that $P' \trianglelefteq P$. Then $[\![P']\!]_G \sqsubseteq [\![P]\!]_G$ for every graph G.*

It should be noticed that the property stated in Lemma 2 does not hold for patterns that are not well designed. For example, consider a graph $G = \{(1, \mathsf{a}, 1),$ $(2, \mathsf{a}, 2), (3, \mathsf{a}, 3)\}$ and non well-designed pattern:

$$P = ((?X, \mathsf{a}, 1) \text{ OPT } ((?Y, \mathsf{a}, 2) \text{ OPT } (?X, \mathsf{a}, 3))).$$

The evaluation of P results in the set $\{\{?X \to 1\}\}$. By deleting the optional part $(?X, \mathsf{a}, 3)$ of P, we obtain the reduction $P' = ((?X, \mathsf{a}, 1) \text{ AND } (?Y, \mathsf{a}, 2))$ of P. The evaluation of P' results in the set $\{\{?X \to 1, ?Y \to 2\}\}$. Thus, we have that $[\![P']\!]_G \not\sqsubseteq [\![P]\!]_G$.

We have mentioned that, when evaluating an optional part of a pattern, one is trying to extend mappings with optional information. Another intuition behind the OPT operator is that, when a pattern has several optional parts, one wants to extend the solutions *as much as possible*, that is, one does not want to lose information when the information is present. We formalize this intuition with the notion of *partial solution* for a pattern. Informally, a partial solution for a pattern P is a mapping that is an *exact match* for some P' such that $P' \trianglelefteq P$. We show then, in Proposition 4, that the evaluation of a well-designed graph pattern P is exactly the set of *maximal partial solutions* for P w.r.t. $\sqsubseteq$, that is, the solutions that retrieve as much information as possible. This proposition gives an alternative characterization of the evaluation of well-designed graph patterns.

Given a pattern P, define $\text{and}(P)$ to be the pattern obtained from P by replacing every OPT operator in P by an AND operator. For example, if P is the pattern:

$$P = ((t_1 \text{ OPT } t_2) \text{ AND } (t_2 \text{ OPT } (t_3 \text{ AND } t_4))),$$

then we have that:

$$\text{and}(P) = ((t_1 \text{ AND } t_2) \text{ AND } (t_2 \text{ AND } (t_3 \text{ AND } t_4))).$$

Notice that, by the semantics of the OPT operator, for every (not necessarily well designed) pattern P and every graph G, we have that $[\![\text{and}(P)]\!]_G \subseteq [\![P]\!]_G$.

A mapping μ is a *partial solution* for a pattern P over a graph G if $\mu \in [\![\text{and}(P')]\!]_G$, for some $P' \trianglelefteq P$. Partial solutions and the notion of subsumption of mappings give the following characterization of the evaluation of well-designed graph patterns.

Proposition 4 ([41]). *Given a* UNION-*free well-designed graph pattern P, a graph G, and a mapping μ, we have that $\mu \in [\![P]\!]_G$ if and only if μ is a maximal (w.r.t. $\sqsubseteq$) partial solution for P over G.*

In [41], the authors use this characterization to prove that the complexity of the evaluation problem for well-designed patterns is lower than for general patterns.

Theorem 8 ([41]). EVALUATION *is coNP-complete for the case of* UNION-*free well-designed graph pattern expressions.*

The characterization of the evaluation of well-designed graph patterns in Proposition 4 can be extended to patterns in UNION normal form. For a well-designed pattern $P = (P_1 \text{ UNION } P_2 \text{ UNION } \cdots \text{ UNION } P_n)$ in UNION normal form, a mapping μ, and a graph G, it holds that $\mu \in [\![P]\!]_G$ if and only if μ is a maximal partial solution (w.r.t. $\sqsubseteq$) for some P_i $(1 \leq i \leq n)$. Then the evaluation problem for well-designed patterns in UNION normal form is still in coNP.

Corollary 2 ([41]). EVALUATION *is coNP-complete for well-designed graph pattern expressions in* UNION *normal form.*

4.5   Optimization of Well-Designed Patterns

Due to the evident similarity between certain operators of SPARQL and relational algebra, a natural question is whether the classical results of normal forms and optimization for relational algebra are applicable in the SPARQL context. The answer is not straightforward, at least for the case of optional patterns and its relational counterpart, the left outer join. The classical results about outer-join query reordering and optimization by Galindo-Legaria and Rosenthal [21] are not directly applicable in the SPARQL context, as they assume constraints on the relational queries that are rarely satisfied in SPARQL. The first, and most problematic issue, is the assumption on predicates used for joining/outer-joining relations to be *null-rejecting* [21]. A predicate p is null-rejecting if it evaluates to *false* (or *undefined*) whenever a null value is used in p. In SPARQL, those predicates are implicit in the variables that graph patterns share and, by the definition of compatible mappings, they are never *null-rejecting*. In fact, people who have developed algorithms for translating SPARQL queries into relational algebra and SQL queries (e.g. [20]) have used `NULL` to represent unbound variables, `IS NULL` in predicates for joining/outer-joining, and `COALESCE` for merging the values of different columns into a single column. These features are explicitly prohibited in [21] since they may imply a violation of the null-rejecting requirement.

Since the application of classical results in relational query optimization is not straightforward, it would be desirable to develop specific techniques in the SPARQL context. In [40], the authors proved that the property of being well designed has important consequences for the study of normalization and optimization for SPARQL.

Proposition 5 ([40]). *Let P_1, P_2 and P_3 be graph pattern expressions and R a built-in condition. Consider the rewriting rules:*

$$((P_1 \text{ OPT } P_2) \text{ FILTER } R) \longrightarrow ((P_1 \text{ FILTER } R) \text{ OPT } P_2), \tag{3}$$

$$(P_1 \text{ AND } (P_2 \text{ OPT } P_3)) \longrightarrow ((P_1 \text{ AND } P_2) \text{ OPT } P_3), \tag{4}$$

$$((P_1 \text{ OPT } P_2) \text{ AND } P_3) \longrightarrow ((P_1 \text{ AND } P_3) \text{ OPT } P_2). \tag{5}$$

Let P be a UNION-free well-designed pattern, and assume that P' is a pattern obtained from P by applying either Rule (3), or Rule (4), or Rule (5). Then P' is a UNION-free well-designed pattern equivalent to P.

It is worth mentioning that the previous rules are not applicable to non well-designed graph patterns. For example, consider the graph $G = \{(1, \mathsf{a}, 1), (2, \mathsf{a}, 2), (3, \mathsf{a}, 3)\}$ and non well-designed pattern:

$$P = ((?X, \mathsf{a}, 1) \text{ AND } ((?Y, \mathsf{a}, 2) \text{ OPT } (?X, \mathsf{a}, 3))).$$

The evaluation of P results in the empty set of mappings. If we apply rule (4) to P, we obtain pattern $P' = (((?X, \mathsf{a}, 1) \text{ AND } (?Y, \mathsf{a}, 2)) \text{ OPT } (?X, \mathsf{a}, 3))$. The evaluation of P' results in the set $\{\{?X \to 1, ?Y \to 2\}\}$ and, thus, we have that $[\![P]\!]_G \neq [\![P']\!]_G$.

We say that a UNION-free graph pattern P is in OPT *normal form* if either: (1) P is constructed by using only the AND and FILTER operators, or (2) $P = (O_1 \text{ OPT } O_2)$, with O_1 and O_2 patterns in OPT normal form. For example, consider a pattern P:

$$\left[\left(((t_1 \text{ AND } t_2) \text{ FILTER } R_1) \right. \right.$$

$$\left. \left. \text{OPT } (t_3 \text{ OPT } ((t_4 \text{ FILTER } R_2) \text{ AND } t_5)) \right) \text{ OPT } \left(t_6 \text{ FILTER } R_3 \right) \right],$$

where every t_i is a triple pattern, and every R_j is a built-in condition. Then P is in OPT normal form. The following theorem shows that for every well-designed graph pattern, an equivalent pattern in OPT normal form can be efficiently obtained.

Theorem 9 ([41]). *For every UNION-free well-designed pattern P, an equivalent pattern in OPT normal form can be obtained after $O(|P|^2)$ applications of Rules (3)-(5).*

The application of Rules (3)-(5) may have a considerable impact in the cost of evaluating graph patterns. One can measure this impact by analyzing the intermediate sizes of the sets of mappings produced when evaluating a pattern. By the semantics of the OPT operator, when evaluating an expression of the form $(P_1 \text{ OPT } P_2)$ over a graph G, the number of mappings obtained is at least the number of mappings obtained when evaluating P_1 over D. That is, the application of the OPT operator never implies a reduction in the size of the

intermediate results in the evaluation of a graph pattern expression. In contrast, it is clear that operators AND and FILTER may imply a reduction in the size of intermediate results. Thus, for optimization purposes, it would be convenient to perform all the AND and FILTER operations first, delaying the OPT operations to the last step of the evaluation. A pattern in OPT normal form has its operators ordered in a way that, the bottom-up evaluation of the pattern follows exactly this strategy: AND and FILTER operations are executed prior to the execution of the OPT operations.

5   On the Expressiveness of SPARQL

Determining the expressive power of a query language is crucial for understanding its capabilities, that is, what types of queries a user can pose in this language, and how complex the evaluation of such queries is. In this section, we study the expressive power of SPARQL. The main goal is to show that SPARQL is equivalent, from an expressive-power point of view, to Relational Algebra.

In order to determine the expressive power of a query language $\mathcal{L}$, one usually chooses a well-studied query language $\mathcal{L}'$, and then compares the expressiveness of $\mathcal{L}$ and $\mathcal{L}'$. In particular, one says that two query languages have the same expressive power if they express exactly the same set of queries. In this section, we present an overview of the results in [7], that show that the query language SPARQL SELECT has the same expressiveness as non-recursive Datalog with negation (nr-Datalog$^\neg$) and Relational Algebra.

We start with an overview of Datalog (for further details see [1,33]). A *term* is either a *variable* or a *constant*. An *atom* is either a *predicate formula* $p(x_1, ..., x_n)$, where p is a predicate name and each x_i is a term, or an *equality formula* $t_1 = t_2$, where t_1 and t_2 are terms. A *literal* is either an atom (a *positive literal*), or the negation of an atom (a *negative literal*). A *fact* is a predicate formula containing only constants. A *substitution* θ for variables $x_1, \ldots, x_k$ is a set of assignments $\{x_1 \rightarrow t_1, \ldots, x_k \rightarrow t_k\}$ where each t_i is a term. Given a literal L, we denote by $\theta(L)$ the literal that results by replacing in L each variable x_i by the term t_i.

A Datalog *rule* is an expression $H \leftarrow L_1, \ldots, L_n$, where H is a predicate formula containing only variables and each L_i is a literal. H is called the *head* of the rule, and the sequence $L_1, \ldots, L_n$ is called its *body*. A *Datalog program* Π is a finite set of Datalog rules. A predicate is *extensional* in Π if it does not occur in the head of any rule of Π, otherwise it is called *intensional*. A Datalog program is *non-recursive* if there is some ordering $r_1, \ldots, r_m$ of its rules so that, the predicate name in the head of r_i does not occur in the body of a rule r_j for every $j \leq i$. We further impose the following *safety* condition to rules: every variable occurring in a rule r must occur in at least one (positive) predicate formula in the body of r. In what follows, we only consider non-recursive and safe programs. Moreover, we may assume that all heads of rules in a program have distinct variables, since repeated variables can always be replaced by adding equalities. For example, the rule $p(X, X) \leftarrow t(X)$ can be replaced by $p(X, Y) \leftarrow t(X), t(Y), X = Y$.

Let D be a set of facts over the extensional predicates of a Datalog program Π. We define the *meaning* of Π given D, denoted by $\text{facts}^*(\Pi, D)$, as the set of facts that results from the following process. Fix an order $r_1, \ldots, r_m$ of the rules that satisfies the aforementioned non-recursive property. The set $\text{facts}^*(\Pi, D)$ is obtained evaluating the rules by following that order. Formally, we denote by $\text{facts}^i(\Pi, D)$ the total set of facts obtained after evaluating rule r_i. Initially, $\text{facts}^0(\Pi, D) = D$. In order to compute $\text{facts}^{i+1}(\Pi, D)$, assume that rule r_{i+1} is $H \leftarrow L_1, \ldots L_n$. Then $\text{facts}^{i+1}(\Pi, D)$ is obtained by adding to $\text{facts}^i(\Pi, D)$ all the facts of the form $\theta(H)$, where θ is a substitution such that $\theta(L_1), \ldots, \theta(L_n)$ hold in $\text{facts}^i(\Pi, D)$. The process stops when all rules have been considered.

A *Datalog query* Q is a pair (Π, L) where Π is a Datalog program and L is a predicate formula (the *goal* of the program). The *answer* to a Datalog query $Q = (\Pi, L)$ over a database D, denoted by $\underline{\text{answer}}(Q, D)$, is the set of all substitutions θ for the variables occurring in L, such that $\theta(L) \in \text{facts}^*(\Pi, D)$.

5.1   From SPARQL to nr-Datalog⌐

In this section, we show that nr-Datalog⌐ is at least as expressive as SPARQL SELECT, that is, we show that every SPARQL SELECT query can be expressed as an nr-Datalog⌐ program. More specifically, we first define a one-to-one transformation $\mathcal{T}_1$ that assigns to every RDF graph G a set of Datalog facts $\mathcal{T}_1(G)$. We then define a one-to-one transformation $\mathcal{T}_2$ that assigns to every SPARQL SELECT query Q, a Datalog query $\mathcal{T}_2(Q)$, and show that for every SPARQL SELECT query Q and RDF graph G, the evaluation of Q over G corresponds to the evaluation of the Datalog query $\mathcal{T}_2(Q)$ over the set of facts $\mathcal{T}_1(G)$.

The transformation $\mathcal{T}_1$ from RDF graphs into Datalog facts essentially transform triples into facts, but taking special care of encoding unbounded values as nulls. Formally, given an RDF graph G, the transformation $\mathcal{T}_1(G)$ works as follows: every element a occurring in G is encoded by a fact $term(a)$; each triple (s, p, o) is encoded by a fact $triple(s, p, o)$; additionally, we include a special fact $\mathbf{N}(null)$, where $null$ is a constant value used to represent unbounded variables.

We now have to show how graph patterns are transformed into Datalog rules. We show here some examples of this transformation to highlight the intuition of the process. We refer the reader to [44,7] for the details on the general transformation. Consider first the graph pattern $P_1 = ((?X, a, 1) \text{ OPT } (?X, b, ?Z))$. Then the transformation $\mathcal{T}_2$ generates the following Datalog program with goal predicate p to express P_1:

$$p(?X, ?Z) \leftarrow triple(?X, a, 1), triple(?X, b, ?Z) \tag{6}$$

$$p(?X, ?Z) \leftarrow triple(?X, a, 1), \mathbf{N}(?Z), \neg q(?X) \tag{7}$$

$$q(?X) \leftarrow triple(?X, b, ?V) \tag{8}$$

The first rule is encoding the *join* operation between sets of mappings, while the second and third rules are encoding the *difference*. The left outer-join, which defines the semantics of the OPT operator, is then obtained by considering rules (6), (7) and (8), that is, considering the union between the results of the join

and the difference. Notice that predicate **N** is used in the second rule to encode unbounded variables.

Second, consider SPARQL SELECT query $(\{?Z\}, P_1)$, where P_1 is the pattern defined above. To express the SELECT operator, one only needs to perform a *projection* in Datalog, that is, one can express query $(\{?Z\}, P_1)$ by using rules (6), (7), (8) and the following projection rule:

$$r(?Z) \leftarrow p(?X, ?Z).$$

Notice that in this case r is the new goal predicate.

Finally, consider SPARQL pattern:

$$P_2 = \Big((?X, a, 1) \text{ AND } \big((?X, b, 1) \text{ UNION } (?Y, c, 1) \big) \Big).$$

The main difficulty in translating P_2 into an nr-Datalog$^\neg$ program is the encoding of the notion of *compatible mapping*. To see why this is the case, first notice that one can easily express pattern $P_2' = ((?X, b, 1) \text{ UNION } (?Y, c, 1))$ as an nr-Datalog$^\neg$ program:

$$p'(?X, ?Y) \leftarrow triple(?X, b, 1), \mathbf{N}(?Y),$$
$$p'(?X, ?Y) \leftarrow triple(?Y, c, 1), \mathbf{N}(?X).$$

But if we now want to translate pattern $P_2 = ((?X, a, 1) \text{ AND } P_2')$, one cannot directly use the previous two rules together with a rule like the following:

$$p(?X, ?Y) \leftarrow triple(?X, a, 1), p'(?X, ?Y),$$

as this rule does not take into consideration the fact that the occurrence of $?X$ in p' could be instantiated with value *null*. In fact, if this is the case, then the rule does not generate any facts as either there is no value $d \in U$ such that $triple(d, a, 1)$ holds, or there is such a value d but then d is different from *null*. Notice that this failure is due to the fact that the previous rule does not correctly encode the notion of compatible mapping. To solve this problem, one needs to replace the previous rule by:

$$p(?X, ?Y) \leftarrow triple(?X, a, 1), p'(?U, ?Y), compatible(?X, ?U),$$

where $compatible(\cdot, \cdot)$ is defined as:

$$compatible(?X, ?Y) \leftarrow term(?X), term(?Y), ?X = ?Y$$
$$compatible(?X, ?Y) \leftarrow term(?X), \mathbf{N}(?Y)$$
$$compatible(?X, ?Y) \leftarrow \mathbf{N}(?X), term(?Y)$$
$$compatible(?X, ?Y) \leftarrow \mathbf{N}(?X), \mathbf{N}(?Y)$$

To conclude this section, it only remains to show how SPARQL mappings are represented as Datalog substitutions. Notice that a mapping μ is a partial function. To represent the fact that a mapping is not defined for some variables, we

use the special value *null*. Given a mapping μ and a set of variables W such that $\text{dom}(\mu) \subseteq W$, we define $\theta_{(\mu,W)}$ as a substitution for variables in W such that (1) $\theta_{(\mu,W)}(?X) = \mu(?X)$ for every variable $?X \in \text{dom}(\mu)$, and (2) $\theta_{(\mu,W)}(?X) = null$ for every variable $?X$ such that $?X \in W$ and $?X \notin \text{dom}(\mu)$.

With the above transformations, we can show that nr-Datalog$^{\neg}$ is at least as expressive as the language SPARQL SELECT. More precisely, let G be an RDF graph and $Q = (W, P)$ a SPARQL SELECT query, with W a set of variables and P a SPARQL graph pattern. Then a mapping μ is in $[\![Q]\!]_G$ if and only if the substitution $\theta_{(\mu,W)}$ is in $\underline{\text{answer}}(\mathcal{T}_1(Q), \mathcal{T}_2(G))$. Thus, we have that:

Theorem 10 ([44,7]). *nr-Datalog$^{\neg}$ is at least as expressive as the language* SPARQL SELECT.

5.2 From Datalog to SPARQL

In this section, we show that SPARQL is at least as expressive as nr-Datalog$^{\neg}$, that is, we provide transformations from Datalog facts into RDF graphs, Datalog substitutions into SPARQL mappings, and nr-Datalog$^{\neg}$ programs into SPARQL graph patterns. But before presenting these transformations, we give a technical result that is used to encode negated literals of Datalog rules. Let MINUS be a binary operator defined as follows. Given SPARQL graph patterns P_1, P_2 and an RDF graph G:

$$[\![(P_1 \ \text{MINUS} \ P_2)]\!]_G = [\![P_1]\!]_G \smallsetminus [\![P_2]\!]_G,$$

where $\smallsetminus$ denotes the difference between sets of mappings defined in Section 3. Then the following proposition shows that the MINUS operator can be expressed in SPARQL:

Proposition 6. *Let P_1 and P_2 be graph patterns. Then pattern $(P_1 \ \text{MINUS} \ P_2)$ is equivalent to:*

$$\left(\left(P_1 \ \text{OPT} \ (P_2 \ \text{AND} \ (?X_1, ?X_2, ?X_3)) \right) \ \text{FILTER} \ \neg \, \text{bound}(?X_1) \right), \quad (9)$$

where $?X_1, ?X_2, ?X_3$ are fresh variables mentioned neither in P_1 nor in P_2.

Thus, from now on we use SPARQL patterns including the operator MINUS, as they can be translated into usual SPARQL patterns.

We now describe the transformations used to show that nr-Datalog$^{\neg}$ is contained in SPARQL. Given a fact $f = p(c_1, ..., c_n)$, let $\text{desc}(f)$ be the set of triples $\{(b, \text{predicate}, p), (b, 1, c_1), \ldots, (b, n, c_n)\}$, where b is a *fresh* value in U. Moreover, given a set of facts D, define a one-to-one transformation $\mathcal{T}_1'$ as $\mathcal{T}_1'(D) = \{\text{desc}(f) \mid f \in D\}$.

Transformation $\mathcal{T}_1'$ allows one to represent a set of facts as an RDF graph. Thus, to show that SPARQL SELECT is at least as expressive as nr-Datalog$^{\neg}$, it remains to provide a one-to-one mapping $\mathcal{T}_2'$ that transforms nr-Datalog$^{\neg}$ programs into SPARQL SELECT queries. As we did for the other direction, we

show the intuition of the transformation with an example, and refer the reader to [7] for a detailed description of this transformation. Let Π be an nr-Datalog⁻ program, and L a predicate formula $p(x_1, \ldots, x_n)$. For the sake of readability, we assume that all the variables in Π are in V (that is, they can be used as variables in SPARQL graph patterns). We define $\mathrm{gp}(\Pi, L)$ as a function which returns a graph pattern that encodes the program (Π, L). The function $\mathrm{gp}(\Pi, L)$ works as follows:

(a) If predicate p is extensional in Π, then $\mathrm{gp}(\Pi, L)$ returns the graph pattern $((?Y, \mathrm{predicate}, p) \text{ AND } (?Y, 1, x_1) \text{ AND } \cdots \text{ AND } (?Y, 1, x_n))$, where $?Y$ is a fresh variable.

(b) If predicate p is intensional in Π, then for each rule $L \leftarrow L_1, \cdots, L_s, \neg K_1, \cdots, \neg K_t, L_1^{eq}, \cdots, L_u^{eq}$ in Π having p in its head, where each L_i is a positive literal and each L_j^{eq} is a literal of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$, the following SPARQL pattern is generated:

$$\left[\left(\left(\cdots \left(\left(\mathrm{gp}(\Pi, L_1) \text{ AND } \cdots \text{ AND } \mathrm{gp}(\Pi, L_s) \right) \right. \right. \right. \right.$$
$$\left. \left. \text{MINUS } \mathrm{gp}(\Pi, K_1) \right) \cdots \right) \text{ MINUS } \mathrm{gp}(\Pi, K_t) \right)$$
$$\left. \text{FILTER } \left(L_1^{eq} \wedge \cdots \wedge L_u^{eq} \right) \right].$$

Assume that there are k rules in Π having p in their heads, and that P_1, ..., P_k are the SPARQL patterns generated from these rules as above. Then $\mathrm{gp}(\Pi, L)$ is defined as $(P_1 \text{ UNION } \cdots \text{ UNION } P_k)$.

Function $\mathrm{gp}(\cdot, \cdot)$ is used to define transformation T_2'. More precisely, if the set of variables mentioned in L is W, then $T_2'((\Pi, L))$ is the SPARQL SELECT query $(W, \mathrm{gp}(\Pi, L))$.

Example 5. Consider the following Datalog program Π:

$$p(?X, ?Y) \leftarrow r(?X, ?Y, ?Z), \neg s(?X, ?X)$$
$$p(?X, ?Y) \leftarrow t(?X, ?Y)$$

In order to translate this program into a SPARQL SELECT query, the first rule is transformed into the pattern:

$$P_1 = \left[\left((?U, \mathrm{predicate}, r) \text{ AND } (?U, 1, ?X) \text{ AND } (?U, 2, ?Y) \text{ AND } (?U, 3, ?Z) \right) \right.$$
$$\left. \text{MINUS } \left((?V, \mathrm{predicate}, s) \text{ AND } (?V, 1, ?X) \text{ AND } (?V, 2, ?X) \right) \right],$$

and the second rule is transformed into the pattern:

$$P_2 = \left((?W, \mathrm{predicate}, t) \text{ AND } (?W, 1, ?X) \text{ AND } (?W, 2, ?Y) \right).$$

Thus, we have that $\text{gp}(\Pi, p(?X, ?Y))$ is the pattern $(P_1 \text{ UNION } P_2)$, from which we conclude that $\mathcal{T}_2'((\Pi, p(?X, ?Y)))$ is the SPARQL SELECT query $(\{?X, ?Y\}, (P_1 \text{ UNION } P_2))$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

To conclude this section, it only remains to show how Datalog substitutions are represented as SPARQL mappings. Given a substitution θ over a set W of variables, define μ_θ as a mapping such that: (1) $?X \in \text{dom}(\mu_\theta)$ if and only if $?X \to t$ is in θ and $t \neq \textit{null}$, and (2) for every $?X \in \text{dom}(\mu_\theta)$, mapping μ_θ assigns to $?X$ the value assigned by θ to this variable. This transformation together with $\mathcal{T}_1'$ and $\mathcal{T}_2'$ can be used to show that the language SPARQL SELECT is at least as expressive as nr-Datalog$^\neg$. More precisely, given a set D of Datalog facts and an nr-Datalog$^\neg$ query $Q = (\Pi, L)$, we have that a substitution θ is in $\underline{\text{answer}}(Q, D)$ if and only if the mapping μ_θ is in $[\![\mathcal{T}_2'(Q)]\!]_{\mathcal{T}_1'(D)}$. Thus, we have that:

Theorem 11 ([7]). *The language* SPARQL SELECT *is at least as expressive as nr-Datalog$^\neg$.*

From Theorems 10 and 11, and using the well-known fact that Relational Algebra has the same expressive power as nr-Datalog$^\neg$ [1], we obtain that SPARQL SELECT and Relational Algebra have the same expressive power.

Corollary 3 ([7]). *The language* SPARQL SELECT *has the same expressive power as Relational Algebra.*

6 A Query Language for RDFS Data

The RDF specification includes a set of reserved keywords with its own semantics, the RDFS vocabulary. This vocabulary is designed to describe special relationships between resources like typing and inheritance of classes and properties [11]. As with any data structure designed to model information, a natural question that arises is what the desiderata are for an RDFS query language. Among the multiple design issues to be considered, it has been largely recognized that navigational capabilities are of fundamental importance for data models with explicit tree or graph structure (like XML and RDF [6,12]).

SPARQL has been designed much in the spirit of classical relational languages such as SQL. In particular, it has been noted that, although RDF is a directed labeled graph data format, SPARQL only provides limited navigational functionalities. This is more notorious when one considers the RDFS vocabulary (which current SPARQL specification does not cover [45]), where testing conditions like being a subclass of or a subproperty of naturally requires navigating the RDF data. A good illustration of this is shown by the following query, which cannot be expressed in SPARQL without some navigational capabilities. Consider the RDF graph shown in Fig. 3. This graph stores information about cities, transportation services between cities, and further relationships among those transportation services (in the form of RDFS annotations). For instance, in the graph we have that a "Seafrance" service is a subproperty of a "ferry" service, which in turn is a subproperty of a general "transport" service. Assume that we
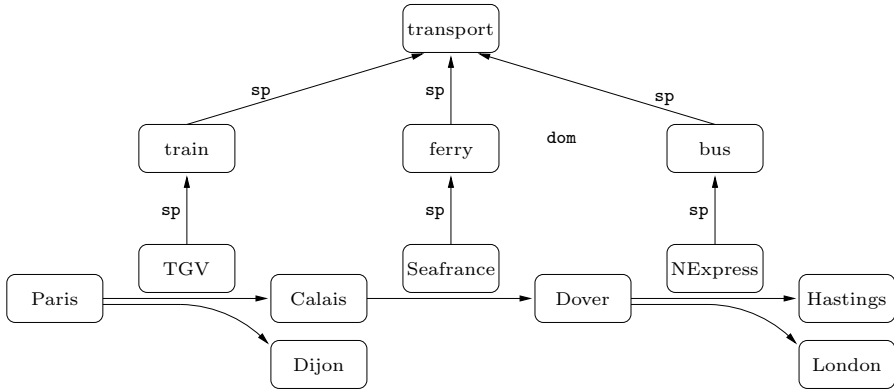
Fig. 3. An RDF graph storing information about transportation services between cities

want to test whether a pair of cities A and B are connected by a sequence of transportation services, but without knowing in advance what services provide those connections. We can answer such a query by testing whether there is a path connecting A and B in the graph, such that every edge in that path is connected with "transport" by following a sequence of subproperty relationships. For instance, for "Paris" and "Calais" the condition holds, since "Paris" is connected with "Calais" by an edge with label "TGV", and "TGV" is a subproperty of "train", which in turn is a subproperty of "transport". Notice that the condition also holds for "Paris" and "Dover".

In this section, we present a language for navigating RDF data grounded on paths expressed with regular expressions, which was proposed in [43]. This language takes advantage of the special features of RDF, and besides regular expressions, it borrows the notion of *branching* from XPath [17], to obtain what is called *nested regular expressions*. We also show how these navigational capabilities can be incorporated into SPARQL, which gives rise to the query language nSPARQL [43].

Furthermore, in this section we consider two fundamental questions about these new navigational capabilities and the language nSPARQL. First, we deal with the problem of whether these new navigational capabilities can be implemented efficiently. In this section, we present the evaluation algorithm for nested regular expressions that was proposed in [43], and which works in time $O(|G| \cdot |E|)$ for an RDF graph G and a nested regular expression E. Second, we consider the issue of whether nSPARQL is a *good* query language from an expressiveness point of view. In this section, we provide evidence that the capabilities of nSPARQL can be used to pose many interesting and natural queries over RDF data. For the sake of presentation, in this section we consider RDF graphs constructed by using only elements from U, that is, we do not consider blank nodes.
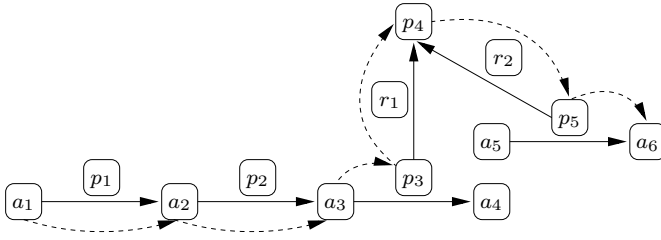
Fig. 4. Nodes a_1 and a_6 are connected by a path that follows the sequence of navigational axes `next/next/edge/next/next`$^{-1}$`/node`

6.1   Nested Regular Expressions for RDF Data

As usual for graph query languages [36,14,6], the language presented in this section uses regular expressions to define paths on graph structures, but taking advantage of the special features of RDF graphs.

The navigation of a graph is usually done by using an operator *next*, which allows one to move from one node to an adjacent one. In our setting, we have RDF "graphs", which are sets of triples, not classical graphs. In particular, instead of classical edges (pair of nodes), we have directed triples of nodes (*hyperedges*). Hence, a language for navigating RDF graphs should be able to deal with this type of objects. In this section, we present the notion of *nested regular expression* to navigate through an RDF graph, which was introduced in [43]. This notion takes into account the special features of the RDF data model. In particular, nested regular expressions use three different *navigation axes* `next`, `edge` and `node`, and their inverses `next`$^{-1}$, `edge`$^{-1}$ and `node`$^{-1}$, to move through an RDF triple. These axes are shown in the following figure:
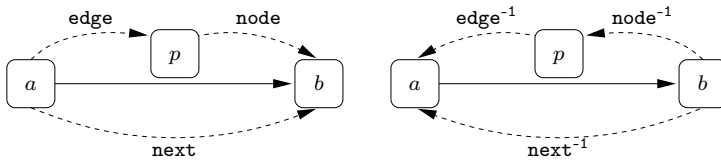
A navigation axis allows one to move one step forward (or backward) in an RDF graph. Thus, a sequence of these axes defines a path in an RDF graph. For instance, in the graph of Fig. 4, the sequence of axes:

$$\texttt{next/next/edge/next/next}^{-1}\texttt{/node}$$

defines a path between nodes a_1 and a_6 (the path is shown with dashed lines in the figure). Moreover, one can use classical regular expressions over these axes to define a set of paths that can be used in a query. The language proposed in [43] considers an additional axis `self` that is used not to actually navigate, but instead to test the label of a specific node in a path. The language also allows *nested expressions* that can be used to test for the existence of certain paths

starting at any axis. The following grammar defines the syntax of nested regular expressions:

$$exp \quad := \quad \text{axis} \mid \text{axis}::a \ (a \in U) \mid \text{axis}::[exp] \mid$$
$$exp/exp \mid exp|exp \mid exp^* \quad (10)$$

where axis $\in \{\text{self}, \text{next}, \text{next}^{-1}, \text{edge}, \text{edge}^{-1}, \text{node}, \text{node}^{-1}\}$. Before introducing the formal semantics of nested regular expressions, we give some intuition about how these expressions are evaluated in an RDF graph. The most natural navigation axis is $\text{next}::a$, with a an arbitrary element from U. Given an RDF graph G, the expression $\text{next}::a$ is interpreted as the $a\text{-}neighbor$ relation in G, that is, the pairs of nodes (x, y) such that $(x, a, y) \in G$. Given that in the RDF data model, a node can also be the label of an edge, the language allows one to navigate from a node to one of its leaving edges by using the edge axis. More formally, the interpretation of $\text{edge}::a$ is the pairs of nodes (x, y) such that $(x, y, a) \in G$. The nesting construction $[exp]$ is used to check for the existence of a path defined by expression exp. For instance, when evaluating nested expression $\text{next}::[exp]$ in a graph G, we retrieve the pairs of nodes (x, y) such that there exists z with $(x, z, y) \in G$, and such that there is a path in G that follows expression exp starting in z.

The evaluation of a nested regular expression exp in a graph G is formally defined as a binary relation $[\![exp]\!]_G$, denoting the pairs of nodes (x, y) such that y is reachable from x in G by following a path that conforms to exp [43]. The formal semantics of the language is shown in Tab. 2. In this table, G is an RDF graph, $a \in U$, $\text{voc}(G)$ is the set of all the elements from U that are mentioned in G, and exp, exp_1, exp_2 are nested regular expressions.

Example 6. Let G be the graph in Fig. 3, and consider expression

$$exp_1 = \text{next}::[\text{next}::\text{sp}/\text{self}::\text{train}].$$

The expression $\text{next}::\text{sp}/\text{self}::\text{train}$ defines the pairs of nodes (z, w) such that from z one can reach w by following an edge labeled sp, and furthermore the label of w is train (expression $\text{self}::\text{train}$ is used to perform this test). Thus, the nested expression $[\text{next}::\text{sp}/\text{self}::\text{train}]$ performs an existential test; it is satisfied by the nodes in G from which there exists a path that follows an edge labeled sp and reaches a node labeled train. TGV is the only such node in G and, thus, we have that $[\![exp_1]\!]_G = \{(\text{Paris}, \text{Calais}), (\text{Paris}, \text{Dijon})\}$. □

6.2 An Efficient Algorithm for Evaluating Nested Regular Expressions

In [43], it was introduced the language nSPARQL that combines the operators of SPARQL with the navigational capabilities of nested regular expressions. As pointed out in that paper, an essential requirement to use nSPARQL in large applications is that nested regular expressions could be evaluated efficiently.

Table 2. Formal semantics of nested regular expressions

$$[\![\mathtt{self}]\!]_G = \{(x,x) \mid x \in \mathrm{voc}(G)\}$$
$$[\![\mathtt{self}\text{::}a]\!]_G = \{(a,a)\}$$
$$[\![\mathtt{next}]\!]_G = \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,z,y) \in G\}$$
$$[\![\mathtt{next}\text{::}a]\!]_G = \{(x,y) \mid (x,a,y) \in G\}$$
$$[\![\mathtt{edge}]\!]_G = \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,y,z) \in G\}$$
$$[\![\mathtt{edge}\text{::}a]\!]_G = \{(x,y) \mid (x,y,a) \in G\}$$
$$[\![\mathtt{node}]\!]_G = \{(x,y) \mid \text{there exists } z \text{ s.t. } (z,x,y) \in G\}$$
$$[\![\mathtt{node}\text{::}a]\!]_G = \{(x,y) \mid (a,x,y) \in G\}$$
$$[\![\mathrm{axis}^{-1}]\!]_G = \{(x,y) \mid (y,x) \in [\![\mathrm{axis}]\!]_G\} \quad \text{with axis} \in \{\mathtt{next}, \mathtt{node}, \mathtt{edge}\}$$
$$[\![\mathrm{axis}^{-1}\text{::}a]\!]_G = \{(x,y) \mid (y,x) \in [\![\mathrm{axis}\text{::}a]\!]_G\} \quad \text{with axis} \in \{\mathtt{next}, \mathtt{node}, \mathtt{edge}\}$$
$$[\![exp_1/exp_2]\!]_G = \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,z) \in [\![exp_1]\!]_G \text{ and } (z,y) \in [\![exp_2]\!]_G\}$$
$$[\![exp_1|exp_2]\!]_G = [\![exp_1]\!]_G \cup [\![exp_2]\!]_G$$
$$[\![exp^*]\!]_G = [\![\mathtt{self}]\!]_G \cup [\![exp]\!]_G \cup [\![exp/exp]\!]_G \cup [\![exp/exp/exp]\!]_G \cup \cdots$$
$$[\![\mathtt{self}\text{::}[exp]]\!]_G = \{(x,x) \mid x \in \mathrm{voc}(G) \text{ and there exists } z \text{ s.t. } (x,z) \in [\![exp]\!]_G\}$$
$$[\![\mathtt{next}\text{::}[exp]]\!]_G = \{(x,y) \mid \text{there exist } z,w \text{ s.t. } (x,z,y) \in G \text{ and } (z,w) \in [\![exp]\!]_G\}$$
$$[\![\mathtt{edge}\text{::}[exp]]\!]_G = \{(x,y) \mid \text{there exist } z,w \text{ s.t. } (x,y,z) \in G \text{ and } (z,w) \in [\![exp]\!]_G\}$$
$$[\![\mathtt{node}\text{::}[exp]]\!]_G = \{(x,y) \mid \text{there exist } z,w \text{ s.t. } (z,x,y) \in G \text{ and } (z,w) \in [\![exp]\!]_G\}$$
$$[\![\mathrm{axis}^{-1}\text{::}[exp]]\!]_G = \{(x,y) \mid (y,x) \in [\![\mathrm{axis}\text{::}[exp]]\!]_G\} \quad \text{with axis} \in \{\mathtt{next}, \mathtt{node}, \mathtt{edge}\}$$

In this section, we present an efficient algorithm for this task, which works in time proportional to the size of the input graph times the size of the expression being evaluated. As is customary when studying the complexity of the evaluation problem for a query language [51], we consider its associated decision problem. For nested regular expressions, this problem is defined as:

> PROBLEM : Evaluation problem for nested regular expressions.
> INPUT : An RDF graph G, a nested regular expression exp, and a pair (a,b).
> QUESTION : Is $(a,b) \in [\![exp]\!]_G$?

It is important to note that the evaluation problem that we study considers the pair of nodes (a,b) as part of the input. That is, similar to the complexity study presented in Section 4, we study the complexity by measuring how difficult it is to verify whether a given pair of nodes is in the evaluation of a nested regular expression over an RDF graph.

Following the terminology introduced in [43], we assume that an RDF graph G is stored as an adjacency list that makes explicit the navigation axes (and their inverses). Thus, every $u \in \mathrm{voc}(G)$ is associated with a list of pairs $\alpha(u)$, where every pair contains a navigation axis and the destination node. For instance, if (s,p,o) is a triple in G, then $(\mathtt{next}\text{::}p, o) \in \alpha(s)$ and $(\mathtt{edge}^{-1}\text{::}o, s) \in \alpha(p)$. Moreover, we assume that $(\mathtt{self}\text{::}u, u) \in \alpha(u)$ for every $u \in \mathrm{voc}(G)$. Notice that if the number of triples in G is N, then the adjacency list representation uses space $O(N)$. Thus, when measuring the size of G, we use $|G|$ to denote the size of its adjacency list representation. We further assume that given an

element $u \in \mathrm{voc}(G)$, we can access its associated list $\alpha(u)$ in time $O(1)$. This is a standard assumption for graph data-structures in a RAM model [19].

Example 7. The following figure shows an example of an adjacency-list representation of an RDF graph.
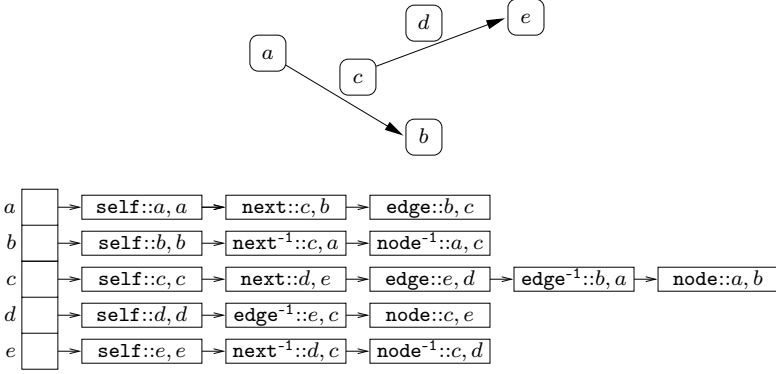
The algorithm in [43] for the evaluation of nested regular expressions was inspired by some of the algorithms for the evaluation of temporal logics [18] and propositional dynamic logic [2,24]. To present this algorithm, we need to introduce some terminology. An expression exp' is a *nested subexpression* of an expression exp if axis::$[exp']$ occurs in exp, with axis $\in$ {self, next, next^{-1}, edge, edge^{-1}, node, node^{-1}}. Given an RDF graph G and a nested regular expression exp, the algorithm proceeds by recursively considering the nested subexpressions of exp, labeling every node u of G with a set label(u) of nested expressions. Initially, label(u) is the empty set. Then at the end of the execution of the algorithm, it holds that $exp \in$ label(u) if and only if there exists z such that $(u, z) \in [\![exp]\!]_G$. Before giving any technical details, let us show the general idea of this process with an example. Figure 5 exemplifies the process for a graph G and the nested expression:

$$\beta = \texttt{next::}a/(\texttt{next::}[\texttt{next::}b/\texttt{self::}c])^*/(\texttt{edge::}[\texttt{next::}d] \mid \texttt{next::}a)^+. \quad (11)$$

The process first considers the nested subexpressions $\gamma = \texttt{next::}b/\texttt{self::}c$ and $\lambda = \texttt{next::}d$, and marks the nodes in G according to which ones of these subexpressions they satisfy. Thus, after this stage we have that $\gamma \in$ label(r_3) since $(r_3, c) \in [\![\gamma]\!]_G$, and $\lambda \in$ label(r_6) since $(r_6, r_7) \in [\![\lambda]\!]_G$ (see Fig. 5). Using this information, the nodes are marked according to whether they satisfy β, but considering the previously computed labels (γ and λ) and the expression $\beta' = \texttt{next::}a/(\texttt{next::}\gamma)^*/(\texttt{edge::}\lambda \mid \texttt{next::}a)^+$. In the example of Fig. 5, we have that $(r_1, r_5) \in [\![\beta]\!]_G$ and, thus, $\beta \in$ label(r_1).

We now explain how to efficiently carry out the labeling process by using some tools from automata theory (here we assume some familiarity with this theory). A key idea in the algorithm presented in [43] is to associate to each nested regular expression a nondeterministic finite automaton with ε-transitions
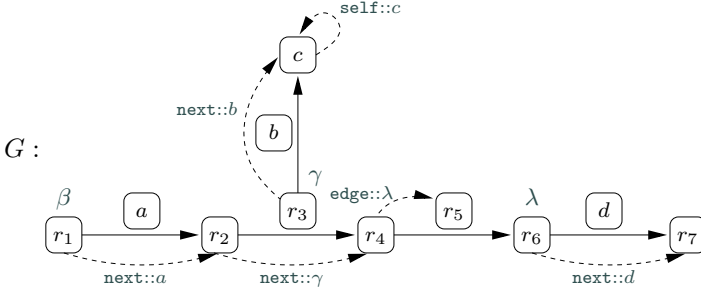
Fig. 5. Example of the labeling process of the RDF graph G according to expression $\beta = \texttt{next::}a/(\texttt{next::}[\texttt{next::}b/\texttt{self::}c])^*/(\texttt{edge::}[\texttt{next::}d] \mid \texttt{next::}a)^+$. First, node r_3 is marked with label $\gamma = \texttt{next::}b/\texttt{self::}c$ (since $(r_3, c) \in [\![\gamma]\!]_G$), and node r_6 with label $\lambda = \texttt{next::}d$ (since $(r_6, r_7) \in [\![\lambda]\!]_G$). Finally, node r_1 is labeled with β (since $(r_1, r_5) \in [\![\beta]\!]_G$). This last label is obtained by considering the expression $\beta' = \texttt{next::}a/(\texttt{next::}\gamma)^*/(\texttt{edge::}\lambda \mid \texttt{next::}a)^+$.

(ε-NFA). Given a nested regular expression *exp*, the set of *depth-0 terms* of *exp*, denoted by $\mathbf{D}_0(exp)$, is recursively defined as follows:

$\mathbf{D}_0(exp) = \{exp\}$ if *exp* is either axis, or axis::a, or axis::$[exp']$,
$\mathbf{D}_0(exp_1/exp_2) = \mathbf{D}_0(exp_1|exp_2) = \mathbf{D}_0(exp_1) \cup \mathbf{D}_0(exp_2)$,
$\mathbf{D}_0(exp^*) = \mathbf{D}_0(exp)$,

where axis $\in \{\texttt{self}, \texttt{next}, \texttt{next}^{\texttt{-1}}, \texttt{edge}, \texttt{edge}^{\texttt{-1}}, \texttt{node}, \texttt{node}^{\texttt{-1}}\}$. For instance, for the nested expression β in (11), we have that:

$$\mathbf{D}_0(\beta) = \{\,\texttt{next::}a, \ \texttt{next::}[\texttt{next::}b/\texttt{self::}c], \ \texttt{edge::}[\texttt{next::}d]\,\}.$$

Notice that a nested regular expression *exp* can be viewed as a classical regular expression over the alphabet $\mathbf{D}_0(exp)$. We denote by $\mathcal{A}_{exp}$ the ε-NFA that accepts the language generated by the regular expression *exp* over the alphabet $\mathbf{D}_0(exp)$. For example, Fig. 6 shows an ε-NFA $\mathcal{A}_\beta$ that accepts the language generated by expression β in (11) over the alphabet $\mathbf{D}_0(\beta)$. As for the case of RDF graphs, ε-NFAs are stored using an adjacency-list representation.

An essential ingredient in the algorithm presented in [43] is the use of the product automaton $G \times \mathcal{A}_{exp}$, which is constructed as follows. Assume that we have the graph G labeled with respect to the nested subexpressions of *exp*, that is, for every node u of G and nested subexpression exp' of *exp*, we have that $exp' \in \text{label}(u)$ if and only if there exists a node v such that $(u, v) \in [\![exp']\!]_G$. Let Q be the set of states of $\mathcal{A}_{exp}$, and $\delta : Q \times (\mathbf{D}_0(exp) \cup \{\varepsilon\}) \to 2^Q$ the transition function of $\mathcal{A}_{exp}$. Then the set of states of $G \times \mathcal{A}_{exp}$ is $\text{voc}(G) \times Q$, and its transition function $\delta' : (\text{voc}(G) \times Q) \times (\mathbf{D}_0(exp) \cup \{\varepsilon\}) \to 2^{\text{voc}(G) \times Q}$ is defined as follows. For every $(u, p) \in \text{voc}(G) \times Q$ and $s \in \mathbf{D}_0(exp)$, we have that $(v, q) \in \delta'((u, p), s)$ if and only if $q \in \delta(p, s)$ and one of the following cases hold:
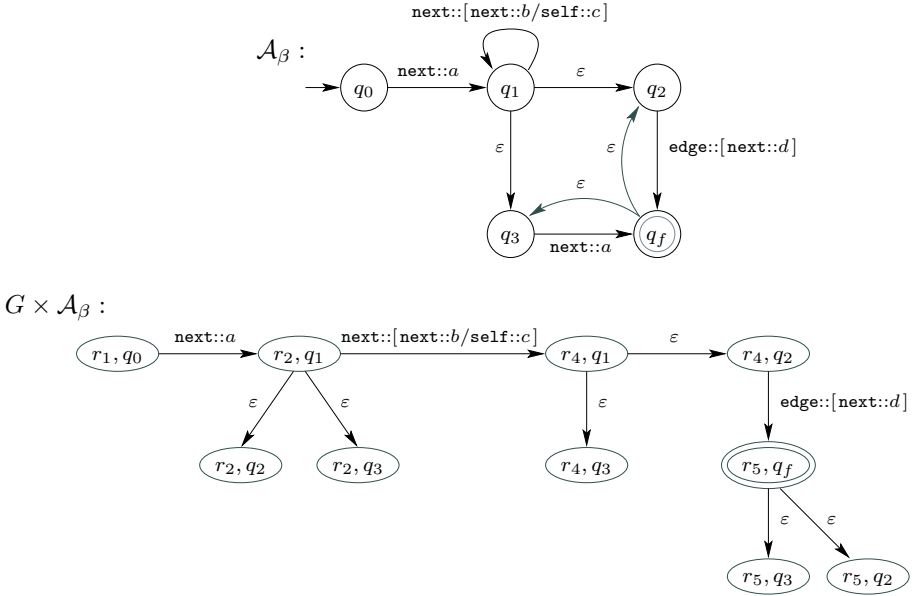
Fig. 6. Automaton $\mathcal{A}_\beta$ for the nested regular expression β in (11), and product automaton $G \times \mathcal{A}_\beta$

- $s = \text{axis}$ and there exists a such that $(\text{axis::}a, v) \in \alpha(u)$,
- $s = \text{axis::}a$ and $(\text{axis::}a, v) \in \alpha(u)$,
- $s = \text{axis::}[exp]$ and there exists b such that $(\text{axis::}b, v) \in \alpha(u)$ and $exp \in \text{label}(b)$,

where axis $\in \{\texttt{self}, \texttt{next}, \texttt{next}^{\text{-1}}, \texttt{edge}, \texttt{edge}^{\text{-1}}, \texttt{node}, \texttt{node}^{\text{-1}}\}$. Additionally, if $q \in \delta(p, \varepsilon)$ we have that $(u, q) \in \delta'((u, p), \varepsilon)$ for every $u \in \text{voc}(G)$. That is, $G \times \mathcal{A}_{exp}$ is the standard product automaton of G and $\mathcal{A}_{exp}$ if G is viewed as an ε-NFA over the alphabet $\mathbf{D}_0(exp)$. Figure 6 shows the product automaton $G \times \mathcal{A}_\beta$ for the nested expression β in (11) and the graph G of Fig. 5 (labeled with respect to the nested subexpressions of β). In this figure, we have only depicted the states of $G \times \mathcal{A}_\beta$ that are reachable from the initial state. For instance, we have that there is a transition from (r_2, q_1) to (r_4, q_1) with symbol $\texttt{next::}[\texttt{next::}b/\texttt{self::}c]$ since: (i) there is a transition from q_1 to q_1 with $\texttt{next::}[\texttt{next::}b/\texttt{self::}c]$ in $\mathcal{A}_\beta$, and (ii) $(\texttt{next::}r_3, r_4) \in \alpha(r_2)$ and $\gamma = \texttt{next::}b/\texttt{self::}c \in \text{label}(r_3)$.

Two key observations about the product automaton defined above should be made. Let G be a graph labeled with respect to the nested subexpressions of exp, and $\mathcal{A}_{exp}$ an ε-NFA for exp. Assume that q_0 is the initial state of $\mathcal{A}_{exp}$ and q_f is one of its final states. The first observation is that if there exists two elements $u, v \in \text{voc}(G)$ such that from (u, q_0) one can reach state (v, q_f) in $G \times \mathcal{A}_{exp}$, then $(u, v) \in [\![exp]\!]_G$. In the example of Fig. 6, we have that $(r_1, r_5) \in [\![\beta]\!]_G$ since we can reach state (r_5, q_f) from state (r_1, q_0) in $G \times \mathcal{A}_\beta$. The second observation is

that given a nested regular expression exp, one can construct in linear time an ε-NFA for exp by using standard techniques [28]. Thus, given a nested regular expression exp and an RDF graph G that has been labeled with respect to the nested subexpressions of exp, it is easy to see that automaton $G \times \mathcal{A}_{exp}$ can be constructed in time $O(|G| \cdot |\mathcal{A}_{exp}|)$.

Now we have all the necessary ingredients to present the algorithm for the evaluation problem for nested regular expressions given in [43]. This algorithm is split in two procedures: LABEL labels G according to the nested subexpressions of exp as explained above, and EVAL returns YES if $(a, b) \in [\![exp]\!]_G$ and NO otherwise.

LABEL(G, exp):
1. **for each** axis::$[exp']$ $\in$ $\mathbf{D}_0(exp)$ **do**
2.   call LABEL(G, exp')
3. construct $\mathcal{A}_{exp}$, and assume that q_0 is its initial state and F is its set of final states
4. construct $G \times \mathcal{A}_{exp}$
5. **for each** (u, q_0) that is connected to a state (v, q_f) in $G \times \mathcal{A}_{exp}$, with $q_f \in F$ **do**
6.   label$(u) := $ label$(u) \cup \{exp\}$

EVAL$(G, exp, (a, b))$:
1. **for each** $u \in \text{voc}(G)$ **do**
2.   label$(u) := \emptyset$
3. call LABEL(G, exp)
4. construct $\mathcal{A}_{exp}$, and assume that q_0 is its initial state and F is its set of final states
5. construct $G \times \mathcal{A}_{exp}$
6. **if** a state (b, q_f), with $q_f \in F$, is reachable from (a, q_0) in $G \times \mathcal{A}_{exp}$
7.   **then return** YES
8.   **else return** NO

In [43], it is formally proved that procedure EVAL can be implemented efficiently. More precisely, assuming that $|exp|$ denotes the size of a nested regular expression exp, it is shown in [43] that:

Theorem 12 ([43]). *Procedure* EVAL *solves the evaluation problem for nested regular expressions in time* $O(|G| \cdot |exp|)$.

6.3   The Navigational Language nSPARQL

We conclude this section by presenting the query language nSPARQL introduced in [43], and showing that the navigational capabilities of nSPARQL can be used to pose many interesting and natural queries over RDF data. In particular, we formally show that these capabilities can be used to evaluate queries according to the semantics of the RDFS vocabulary.

The language nSPARQL (*nested* SPARQL) is obtained by using triple patterns with nested regular expressions in the predicate position, plus SPARQL operators AND, OPT, UNION, and FILTER. Formally, a *nested-regular-expression triple* (or just nre-triple) is a tuple t of the form (x, exp, y), where $x, y \in U \cup V$ and exp is a nested regular expression. nSPARQL patterns are recursively defined from nre-triples:

- An nre-triple is an nSPARQL pattern.
- If P_1 and P_2 are nSPARQL patterns and R is a built-in condition, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, $(P_1 \text{ UNION } P_2)$, and $(P_1 \text{ FILTER } R)$ are nSPARQL patterns.

To define the semantics of nSPARQL, we just need to define the semantics of nre-triples. The evaluation of an nre-triple $t = (?X, exp, ?Y)$ over an RDF graph G is defined as the following set of mappings:

$$\llbracket t \rrbracket_G = \{\mu \mid \text{dom}(\mu) = \{?X, ?Y\} \text{ and } (\mu(?X), \mu(?Y)) \in \llbracket exp \rrbracket_G\}.$$

Similarly, the evaluation of an nre-triple $t = (?X, exp, a)$ over an RDF graph G, where $a \in U$, is defined as $\{\mu \mid \text{dom}(\mu) = \{?X\} \text{ and } (\mu(?X), a) \in \llbracket exp \rrbracket_G\}$, and likewise for $(a, exp, ?X)$ and (a, exp, b) with $b \in U$.

Notice that every SPARQL triple $(?X, p, ?Y)$ with $p \in U$ is equivalent to nSPARQL triple $(?X, \texttt{next::}p, ?Y)$. Also notice that, since variables are not allowed in nested regular expressions, the occurrence of variables in the predicate position of triple patterns is forbidden in nSPARQL. Nevertheless, every SPARQL triple of the form $(?X, ?Y, a)$, with $a \in U$, is equivalent to nSPARQL pattern $(?X, \texttt{edge::}a, ?Y)$, and every triple of the form $(a, ?X, ?Y)$ is equivalent to $(?X, \texttt{node::}a, ?Y)$. Thus, what one loses in nSPARQL is only the possibility of using variables in the three positions of a triple pattern [43].

In the following examples, we show that the navigational capabilities of nSPARQL can be used to pose queries that are likely to occur in the Semantic Web, and which cannot be expressed in SPARQL without using nested regular expressions.

Example 8. Assume that we want to obtain the pairs of cities $(?X, ?Y)$ such that there is a way to travel from $?X$ to $?Y$ by using either Seafrance or NExpress, with an intermediate stop in a city that has a direct NExpress trip to London. Consider nested expression:

$$\begin{aligned} exp_1 = (\texttt{next::Seafrance} \mid \texttt{next::NExpress})^+ / \\ \texttt{self::}[\texttt{next::NExpress/self::London}] / \\ (\texttt{next::Seafrance} \mid \texttt{next::NExpress})^+ \end{aligned}$$

Then pattern $P = (?X, exp_1, ?Y)$ answers our initial query. Notice that expression $\texttt{self::}[\texttt{next::NExpress/self::London}]$ is used to perform the intermediate existential test of having a direct NExpress trip to London.   □

Example 9. Let G be the graph in Fig. 3 and P_1 the following pattern:

$$P_1 = (?X, \ \texttt{next::}[(\texttt{next::sp})^*/\texttt{self::transport}], \ ?Y). \tag{12}$$

Pattern P_1 defines the pairs of cities $(?X, ?Y)$ such that, there exists a triple $(?X, p, ?Y)$ in the graph and a path from p to transport where every edge has label $\texttt{sp}$. Thus, nested expression $[(\texttt{next::sp})^*/\texttt{self::transport}]$ is used to emulate
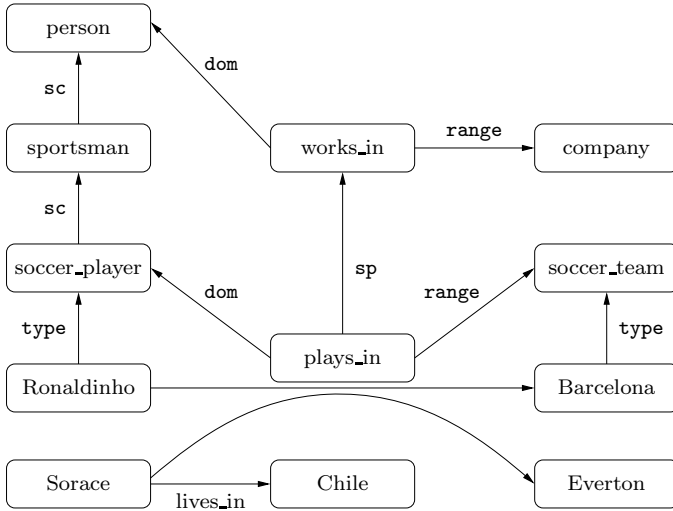
Fig. 7. An RDF graph storing information about soccer players

the process of inference in RDFS; it retrieves all the nodes that are *sub-properties* of transport. Hence, pattern P_1 retrieves the pairs of cities that are connected by a direct transportation service, which could be a train, ferry, bus, etc. In general, if we want to obtain the pairs of cities such that there is a way to travel from one city to the other, we can use the following nSPARQL pattern:

$$P_2 = (?X, \ (\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+, \ ?Y). \qquad (13)$$

□

The previous example shows that nSPARQL can be used to emulate some of the inference rules of RDFS. In [43], it is shown that this is not a particular phenomenon, that is, it is formally proved that if one wants to answer a SPARQL query P according to the semantics of RDFS, then one can rewrite P into an nSPARQL query Q such that Q retrieves the answer to P by directly traversing the input graph. In the remaining of this section, we show how this is done.

SPARQL follows a *subgraph-matching* approach, and thus, a SPARQL query treats RDFS vocabulary without considering its predefined semantics. For example, consider the RDF graph G in Fig. 7, which stores information about soccer players, and consider the graph pattern $P = (?X, \text{works\_in}, ?C)$. Note that, although the triples (Ronaldinho, works_in, Barcelona) and (Sorace, works_in, Everton) can be *deduced* from G, one obtains the empty set as the result of evaluating P over G as there is no triple in G with "works_in" in the predicate position.

We are interested in defining the semantics of SPARQL over RDFS, that is, taking into account not only the explicit RDF triples of a graph G, but also
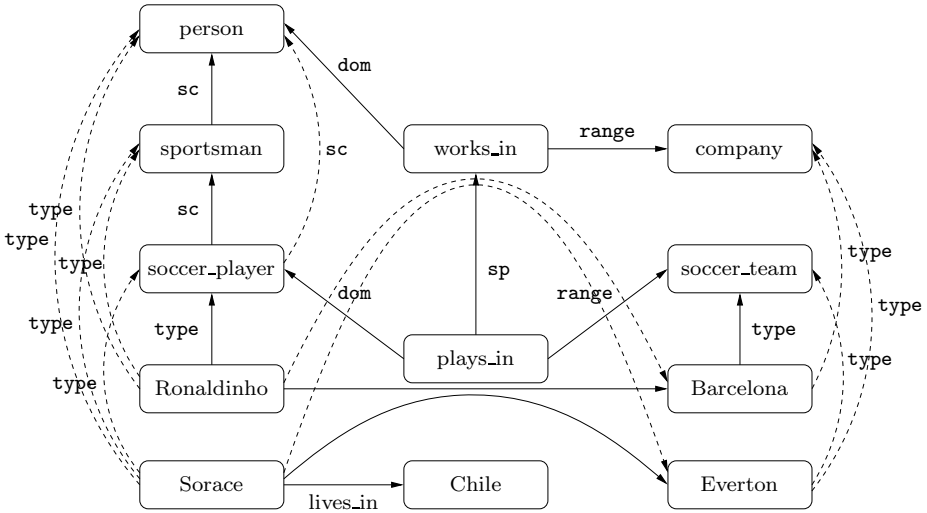
Fig. 8. The closure of the RDF graph of Fig. 7

the triples that can be derived from G according to the semantics of RDFS. We make an initial restriction. In the rest of the paper we assume that RDFS vocabulary cannot occur in subject or object position in RDF triples. Supported on Proposition 1 (2), we only consider rules (2)-(4) for the semantics of RDFS. Let the *closure* of an RDF graph G, denoted by cl(G), be the graph obtained from G by successively applying rules (2)-(4) in Tab. 1 until the graph does not change. For instance, Fig. 8 shows the closure of the RDF graph of Fig. 7. The solid lines in Fig. 8 represent the triples in the original graph, and the dashed lines the additional triples in the closure.

The most direct way to define the semantics of the RDFS evaluation of SPARQL patterns is by considering not the original graph but its closure. Thus, if we now evaluate pattern $P = (?X, \text{works\_in}, ?C)$ over the RDF graph in Fig. 8, we obtain the mappings $\{?X \rightarrow \text{Ronaldinho}, ?C \rightarrow \text{Barcelona}\}$ and $\{?X \rightarrow \text{Sorace}, ?C \rightarrow \text{Everton}\}$. The theoretical formalization of such an approach was studied in [23]. The following definition formalizes this notion.

Definition 6 ([43]). *Given a SPARQL graph pattern P, the RDFS evaluation of P over G, denoted by* $[\![P]\!]_G^{\text{rdfs}}$*, is defined as the set of mappings* $[\![P]\!]_{\text{cl}(G)}$*, that is, as the evaluation of P over the closure of G.*

Let us show with an example how nSPARQL can be used to obtain the RDFS evaluation of some patterns by directly traversing the input graph.

Example 10. Let G be the RDF graph in Fig. 7, and assume that we want to obtain the *type* information of Ronaldinho. This information can be obtained by computing the RDFS evaluation of the pattern $(\text{Ronaldinho}, \text{type}, ?C)$. By

simply inspecting the closure of G in Fig. 8, we obtain that the RDFS evaluation of $(\text{Ronaldinho}, \texttt{type}, ?C)$ is the set of mappings:

$$\{\{?C \rightarrow \text{soccer\_player}\}, \{?C \rightarrow \text{sportsman}\}, \{?C \rightarrow \text{person}\}\}.$$

However, if we directly evaluate this pattern over G, we obtain a single mapping $\{?C \rightarrow \text{soccer\_player}\}$. Consider now the nSPARQL pattern:

$$P = (\text{Ronaldinho}, \ \texttt{next::type/(next::sc)}^*, \ ?C).$$

The expression $\texttt{next::type/(next::sc)}^*$ is intended to obtain the pairs of nodes such that there is a path between them that starts with label $\texttt{type}$ followed by zero or more labels $\texttt{sc}$. When evaluating this expression in G, we obtain the set of pairs $\{(\text{Ronaldinho}, \text{soccer\_player}), (\text{Ronaldinho}, \text{sportsman}), (\text{Ronaldinho}, \text{person}), (\text{Barcelona}, \text{soccer\_team})\}$. Thus, the evaluation of P results in the set of mappings:

$$\{\{?C \rightarrow \text{soccer\_player}\}, \{?C \rightarrow \text{sportsman}\}, \{?C \rightarrow \text{person}\}\}.$$

In this case, pattern P is enough to obtain the type information of Ronaldinho in G according to the RDFS semantics, that is,

$$[\![(\text{Ronaldinho}, \texttt{type}, ?C)]\!]_G^{\text{rdfs}} = [\![(\text{Ronaldinho}, \ \texttt{next::type/(next::sc)}^*, \ ?C)]\!]_G.$$

Although the expression $\texttt{next::type/(next::sc)}^*$ is enough to obtain the type information for Ronaldinho in G, it cannot be used in general to obtain the type information of a resource. For instance, in the same graph, assume that we want to obtain the type information of Everton. In this case, if we evaluate the pattern $(\text{Everton}, \texttt{next::type/(next::sc)}^*, ?C)$ over G, we obtain the empty set. Consider now the nSPARQL pattern:

$$Q = (\text{Everton}, \ \texttt{node}^{-1}\texttt{/(next::sp)}^*\texttt{/next::range}, \ ?C).$$

With the expression $\texttt{node}^{-1}\texttt{/(next::sp)}^*\texttt{/next::range}$, we follow a path that first navigates from a node to one of its incoming edges by using $\texttt{node}^{-1}$, and then continues with zero or more $\texttt{sp}$ edges and a final $\texttt{range}$ edge. The evaluation of this expression over G results in the set $\{(\text{Everton}, \text{soccer\_team}), (\text{Everton}, \text{company}), (\text{Barcelona}, \text{soccer\_team}), (\text{Barcelona}, \text{company})\}$. Thus, the evaluation of Q in G is the set of mappings:

$$\{\{?C \rightarrow \text{soccer\_team}\}, \{?C \rightarrow \text{company}\}\}.$$

By looking at the closure of G in Fig. 8, we see that pattern Q obtains exactly the type information of Everton in G, that is, $[\![(\text{Everton}, \texttt{type}, ?C)]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$.
□

Next we show how the ideas in Examples 9 and 10 were generalized in [43] to obtain a way to evaluate a SPARQL query according to the RDFS semantics. More precisely, we show that if a SPARQL pattern P is constructed by using

triple patterns having at least one position with a non-variable element, then the RDFS evaluation of P can be obtained by directly traversing the input graph with an nSPARQL pattern.

Consider the following *translation* function from elements in U to nested regular expressions:

$$
\begin{aligned}
trans(\texttt{sc}) &= (\texttt{next::sc})^{+} \\
trans(\texttt{sp}) &= (\texttt{next::sp})^{+} \\
trans(\texttt{dom}) &= \texttt{next::dom} \\
trans(\texttt{range}) &= \texttt{next::range} \\
trans(\texttt{type}) &= (\ \texttt{next::type}/(\texttt{next::sc})^{*}\ | \\
&\qquad \texttt{edge}/(\texttt{next::sp})^{*}/\texttt{next::dom}/(\texttt{next::sc})^{*}\ | \\
&\qquad \texttt{node}^{-1}/(\texttt{next::sp})^{*}/\texttt{next::range}/(\texttt{next::sc})^{*}\ ) \\
trans(p) &= \texttt{next::}[(\texttt{next::sp})^{*}/\texttt{self::}p]\ \text{ for } p \notin \{\texttt{sc}, \texttt{sp}, \texttt{range}, \texttt{dom}, \texttt{type}\}.
\end{aligned}
$$

Notice that this translation function has been implicitly used in Examples 9 and 10. In the following lemma, it is shown that given an RDF graph G and a triple pattern t not containing a variable in the predicate position, the above translation function can be used to obtain the RDFS evaluation of t over G by navigating G through a nested regular expression.

Lemma 3 ([43]). *Let (x, p, y) be a SPARQL triple pattern with $x, y \in U \cup V$ and $p \in U$. Then $[\![(x, p, y)]\!]_G^{\mathrm{rdfs}} = [\![(x, trans(p), y)]\!]_G$ for every RDF graph G.*

Suppose now that we have a SPARQL triple pattern t with a variable in the predicate position, but such that the subject and object of t are not both variables. Next it is shown how to construct an nSPARQL pattern P_t such that $[\![t]\!]_G^{\mathrm{rdfs}} = [\![P_t]\!]_G$ [43]. Assume that $t = (x, ?Y, a)$ with $x \in U \cup V$, $?Y \in V$, and $a \in U$, that is, t does not contain a variable in the object position. Consider for every $p \in \{\texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}, \texttt{type}\}$, the pattern $P_{t,p}$ defined as:

$$((x, trans(p), a) \text{ AND } (?Y, \texttt{self::}p, ?Y)).$$

Then define pattern P_t as follows:

$$
P_t = ((x, \texttt{edge::}a/(\texttt{next::sp})^{*}, ?Y) \text{ UNION } P_{t,\texttt{sc}} \text{ UNION } P_{t,\texttt{sp}} \text{ UNION} \\
P_{t,\texttt{dom}} \text{ UNION } P_{t,\texttt{range}} \text{ UNION } P_{t,\texttt{type}}).
$$

In a similar way, it is possible to define pattern P_t for a triple pattern $t = (a, ?Y, x)$, where $a \in U$, $?Y \in V$ and $x \in U \cup V$. By using this construction, it is shown in [43] that:

Lemma 4 ([43]). *Let $t = (x, ?Y, z)$ be a triple pattern such that $?Y \in V$ and $x \notin V$ or $z \notin V$. Then $[\![t]\!]_G^{\mathrm{rdfs}} = [\![P_t]\!]_G$ for every RDF graph G.*

Let $\mathcal{T}$ be the set of triple patterns of the form (x, y, z) such that $x \notin V$ or $y \notin V$ or $z \notin V$. We have shown how to translate every triple pattern $t \in \mathcal{T}$ into an nSPARQL pattern P_t such that $[\![t]\!]_G^{\mathrm{rdfs}} = [\![P_t]\!]_G$. Moreover, for every triple pattern t, its translation is of size linear in the size of t. Given that the semantics of SPARQL is defined from the evaluation of triple patterns, the following results follows:
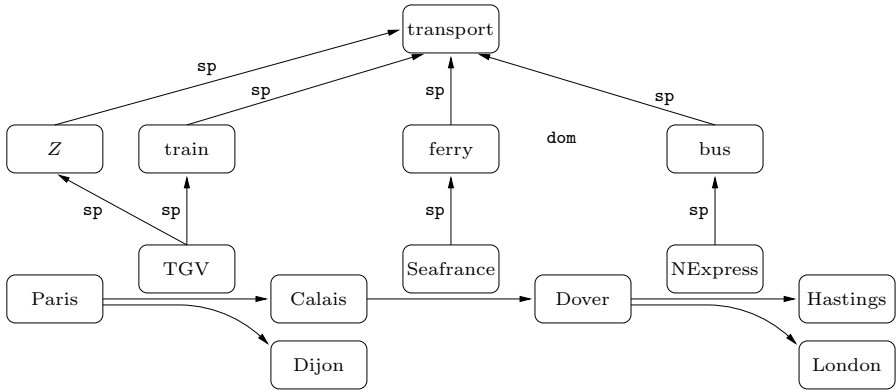
Fig. 9. An RDF graph with RDFS vocabulary and blank nodes

Theorem 13 ([43]). *Let P be a* SPARQL *pattern constructed from triple patterns in $\mathcal{T}$. Then there exists an* nSPARQL *pattern Q such that $[\![P]\!]_G^{\mathrm{rdfs}} = [\![Q]\!]_G$ for every RDF graph G. Moreover, the size of Q is linear in the size of P.*

7 Future Work: Dealing with Blank Nodes

Blank nodes, that is, existential objects, are not new in the area of databases [29,52]. And not only that, they have also been present in the RDF data model since the beginning of the Semantic Web initiative [34]. However, the design of SPARQL was made to keep the efficiency of the language and, in this direction, the current definition of this language does not consider the semantics of blank nodes recommended by the W3C [27]. To see why this is the case, let G_1 and G_2 be the RDF graphs in Figures 3 and 9, respectively, and assume that node Z in G_2 is a blank node. Consider the following SPARQL query:

$$P = \Big(\Big((?X, \mathtt{sp}, ?V) \text{ AND } (?V, \mathtt{sp}, ?Y) \text{ AND }$$
$$(?X, \mathtt{sp}, ?W) \text{ AND } (?W, \mathtt{sp}, ?Y)\Big) \text{ FILTER } \neg(?V = ?W)\Big).$$

Query P evaluated over an RDF graph G retrieves mappings $\{?X \to a, ?Y \to b, ?V \to c, ?W \to d\}$ such that $(a, \mathtt{sp}, c)$, $(c, \mathtt{sp}, b)$, $(a, \mathtt{sp}, d)$ and $(d, \mathtt{sp}, b)$ are all triples in G and c, d are distinct elements. Notice that the clause FILTER $\neg(?V = ?W)$ is used to indicate that $?V$ and $?W$ must take distinct values. Under the W3C semantics for blank nodes [27], G_1 and G_2 are equivalent as blank node Z in G_2 can be identified with node $\mathtt{train}$. Therefore, one would expect that the answer to P over G_1 is the same as over G_2. However, this is not the case; Z and $\mathtt{train}$ are considered to be distinct values under the semantics for SPARQL proposed in [45] and, thus, mapping

$\{?X \rightarrow \text{TGV}, ?Y \rightarrow \text{transport}, ?V \rightarrow \text{train}, ?W \rightarrow Z\}$ is in the answer of P over G_2 but not in the answer of P over G_1.

Evaluating queries which involve blank nodes is challenging, and there is not yet consensus in the Semantic Web community on how to define a query language for this type of data. As an important problem for future work, we identify the issue of extending SPARQL to consider RDF data with blank nodes. In practice, a considerable number of RDF databases include this type of nodes and, thus, this project is driven by the need to extend SPARQL to cope with this data. We hope that a project like this will help in bridging the gap between the current specification of SPARQL [45] and both the definition of the semantics of RDF data [27] and the way RDF data is used in real life.

We conclude this section by pointing out that blank nodes are used not only on RDF graphs but also in SPARQL patterns. They were introduced to make SPARQL compatible with future logical extensions. Nevertheless, they play no major role in the current semantics. In fact, it can be shown that each SPARQL query Q can be simulated by a SPARQL query Q' not mentioning any blank nodes. More precisely, it follows from the definitions of RDF instance mapping, solution mapping, and the order of evaluation of solution modifiers (see [45]), that if Q' is obtained from Q by replacing each blank node B by a fresh variable $?X_B$, then Q and Q' give the same results.

Acknowledgments

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Alechina, N., Immerman, N.: Reachability Logic: An Efficient Fragment of Transitive Closure Logic. Logic Journal of the IGPL 8(3), 325–338 (2000)
3. Alkhateeb, F.: Querying RDF(S) with Regular Expressions. PhD Thesis, Université Joseph Fourier, Grenoble, FR (2008)
4. Alkhateeb, F., Baget, J., Euzenat, J.: RDF with regular expressions. Research Report 6191, INRIA (2007)
5. Alkhateeb, F., Baget, J., Euzenat, J.: Constrained regular expressions in SPARQL. In: SWWS 2008, pp. 91–99 (2008)
6. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1), 1–39 (2008)
7. Angles, R., Gutierrez, C.: The Expressive Power of SPARQL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 114–129. Springer, Heidelberg (2008)

8. Anyanwu, K., Maduko, A., Sheth, A.: SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases. In: WWW 2007, pp. 797–806 (2007)

9. Arenas, M., Gutierrez, C., Parsia, B., Pérez, J., Polleres, A., Seaborne, A.: SPARQL - Where are we? Current state, theory and practice. Unit-2: SPARQL Formalization. In: Tutorial given at ESWC 2007, Innsbruck, Austria (2007), `http://axel.deri.ie/~axepol/sparqltutorial/`

10. Arenas, M., Gutierrez, C., Pérez, J.: An Extension of SPARQL for RDFS. In: Christophides, V., Collard, M., Gutierrez, C. (eds.) SWDB-ODBIS 2007. LNCS, vol. 5005, pp. 1–20. Springer, Heidelberg (2008)

11. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (February 2004), `http://www.w3.org/TR/rdf-schema/`

12. Benedikt, M., Koch, C.: XPath leashed. ACM Computing Surveys 41(1) (2008)

13. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)

14. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Rewriting of Regular Expressions and Regular Path Queries. J. Comput. Syst. Sci (JCSS) 64(3), 443–465 (2002)

15. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs. Journal of Web Semantics 3, 247–267 (2005)

16. Chandra, A.K., Merlin, P.M.: Optimal Implementation of Conjunctive Queries in Relational Data Bases. In: STOC 1977, pp. 77–90 (1977)

17. Clark, J., DeRose, S.: XML Path Language (XPath). W3C Recommendation (November 1999), `http://www.w3.org/TR/xpath`

18. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (2000)

19. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. McGraw-Hill, New York (2003)

20. Cyganiak, R.: A relational algebra for SPARQL. Tech. Rep. HPL-2005-170, HP-Labs (2005), `http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html`

21. Galindo-Legaria, C.A., Rosenthal, A.: Outerjoin simplification and reordering for query optimization. TODS 22(1), 43–73 (1997)

22. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)

23. Gutierrez, C., Hurtado, C., Mendelzon, A.: Foundations of Semantic Web Databases. In: PODS 2004, pp. 95–106 (2004)

24. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)

25. Harris, S., Gibbins, N.: 3store: Efficient bulk RDF storage. In: PSSS 2003, pp. 1–15 (2003)

26. Hayes, J., Gutierrez, C.: Bipartite Graphs as Intermediate Model for RDF. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 47–61. Springer, Heidelberg (2004)

27. Hayes, P.: RDF Semantics. W3C Recommendation (February 2004), `http://www.w3.org/TR/rdf-mt/`

28. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, Reading (2006)

29. Imielinski, T., Lipski Jr., W.: Incomplete Information in Relational Databases. J. ACM 31(4), 761–791 (1984)

30. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: a declarative query language for RDF. In: WWW 2002, pp. 592–603 (2002)

31. Kochut, K.J., Janik, M.: SPARQLeR: Extended Sparql for Semantic Association Discovery. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 145–159. Springer, Heidelberg (2007)
32. Lassila, O., Swick, R.: Resource description framework (RDF) model and syntax specification W3C Recommendation (February 1999), http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
33. Levene, M., Loizou, G.: A Guided Tour of Relational Databases and Beyond. Springer, Heidelberg (1999)
34. Manola, F., Miller, E., McBride, B.: RDF Primer, W3C Recommendation (February 10 , 2004), http://www.w3.org/TR/REC-rdf-syntax/
35. Marin, D.: RDF Formalization, Santiago de Chile, Technical Report Universidad de Chile, TR/DCC-2006-8 (2004), http://www.dcc.uchile.cl/~cgutierr/ftp/draltan.pdf
36. Mendelzon, A., Wood, P.: Finding Regular Simple Paths in Graph Databases. SIAM J. Comput. 24(6), 1235–1258 (1995)
37. Muñoz, S., Pérez, J., Gutierrez, C.: Minimal Deductive Systems for RDF. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 53–67. Springer, Heidelberg (2007)
38. Olson, M., Ogbuji, U.: The Versa Specification, http://uche.ogbuji.net/tech/rdf/versa/etc/versa-1.0.xml
39. ODP - Open Directory Project, http://www.dmoz.org/
40. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
41. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL (submitted for publication)
42. Pérez, J., Arenas, M., Gutierrez, C.: Semantics of SPARQL. Tech. Report Universidad de Chile, TR/DCC-2006-17 (2006)
43. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A Navigational Language for RDF. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 66–81. Springer, Heidelberg (2008)
44. Polleres, A.: From SPARQL to rules (and back). In: Proceedings of the 16th International World Wide Web Conference (WWW), pp. 787–796. ACM, New York (2007)
45. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (January 2008), http://www.w3.org/TR/rdf-sparql-query/
46. RDF Site Summary (RSS) 1.0, http://web.resource.org/rss/1.0/
47. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL Query Optimization. arXiv.org paper arXiv:0812.3788v1 (December 19, 2008)
48. The Dublin Core Metadata Initiative, http://dublincore.org/
49. The Friend of a Friend (FOAF) project, http://www.foaf-project.org/
50. Uniform Resource Identifier (URI): Generic Syntax, http://tools.ietf.org/html/rfc3986
51. Vardi, M.Y.: The Complexity of Relational Query Languages (Extended Abstract). In: STOC 1982, pp. 137–146 (1982)
52. Zaniolo, C.: Database Relations with Null Values. J. Comput. Syst. Sci. 28(1), 142–166 (1984)