Querying Semantic Web Data with SPARQL

Marcelo Arenas
Department of Computer Science
PUC Chile
marenas@ing.puc.cl

Jorge Pérez
Department of Computer Science
Universidad de Chile
jperez@dcc.uchile.cl

ABSTRACT

The Semantic Web is the initiative of the W3C to make information on the Web readable not only by humans but also by machines. RDF is the data model for Semantic Web data, and SPARQL is the standard query language for this data model. In the last ten years, we have witnessed a constant growth in the amount of RDF data available on the Web, which have motivated the theoretical study of some fundamental aspects of SPARQL and the development of efficient mechanisms for implementing this query language.

Some of the distinctive features of RDF have made the study and implementation of SPARQL challenging. First, as opposed to usual database applications, the semantics of RDF is open world, making RDF databases inherently incomplete. Thus, one usually obtains partial answers when querying RDF with SPARQL, and the possibility of adding optional information if present is a crucial feature of SPARQL. Second, RDF databases have a graph structure and are interlinked, thus making graph navigational capabilities a necessary component of SPARQL. Last, but not least, SPARQL has to work at Web scale!

RDF and SPARQL have attracted interest from the database community. However, we think that this community has much more to say about these technologies, and, in particular, about the fundamental database problems that need to be solved in order to provide solid foundations for the development of these technologies. In this paper, we survey some of the main results about the theory of RDF and SPARQL putting emphasis on some research opportunities for the database community.

Categories and Subject Descriptors
H.2.3 [**Database Management**]: Query languages

General Terms
Algorithms, Theory

Keywords
SPARQL, RDF, RDFS, Semantic Web, Linked Data

1. INTRODUCTION

The Resource Description Framework (RDF) [26] 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 [37]. Since then, SPARQL has been rapidly adopted as the standard for querying Semantic Web data. In January 2008, SPARQL became a W3C Recommendation.

In the last ten years, we have witnessed a constant growth in the amount of RDF data available on the Web, which has motivated the theoretical study of some fundamental aspects of SPARQL, and the development of efficient mechanisms for implementing this query language. RDF and SPARQL have attracted interest from the database community. However, we think that this community has much more to say about these technologies, and, in particular, about the fundamental database problems that need to be solved in order to provide solid foundations for their development.

This paper is a compendium of the material to be presented in a tutorial about RDF and SPARQL, targeted to database researchers that are new to Semantic Web data management. Next, we briefly discuss the topics to be covered in this paper, putting an emphasis on the distinctive features that make the problem of querying Semantic Web data challenging.

RDF and SPARQL: An atomic piece of data in RDF is a *Uniform Resource Identifier* (URI), which is syntactically similar to a URL but identifies an abstract *resource* (which can be, literally, anything). For example, `http://dbpedia.org/resource/Ronald_Fagin` is the URI used by DBpedia (the Semantic Web version of Wikipedia [13]) to identify Ronald Fagin. In the RDF data model, URIs are organized in the form of so called RDF graphs, which are labeled directed graphs, where node labels and edge labels are URIs. Formally, an RDF graph is a finite set of *triples* of the form (*subject*, *predicate*, *object*). Figure 1 shows an example of an RDF graph that stores data from DBpedia [13] and an RDF representation of DBLP [24]. Shorthands for URIs prefixes are usually defined to avoid overcrowding RDF specifications and queries. In Figure 1, prefixes are shown at the top of the figure. For example, the following is a triple in the RDF graph in Figure 1:

$$(\texttt{inPods:FaginLN01},\ \texttt{dct:isPartOf},\ \texttt{inPods:2001})$$

SPARQL is essentially a graph-matching query language. A SPARQL query is composed of: (1) a *body*, which is a

PODS'11, June 13–15, 2011, Athens, Greece.

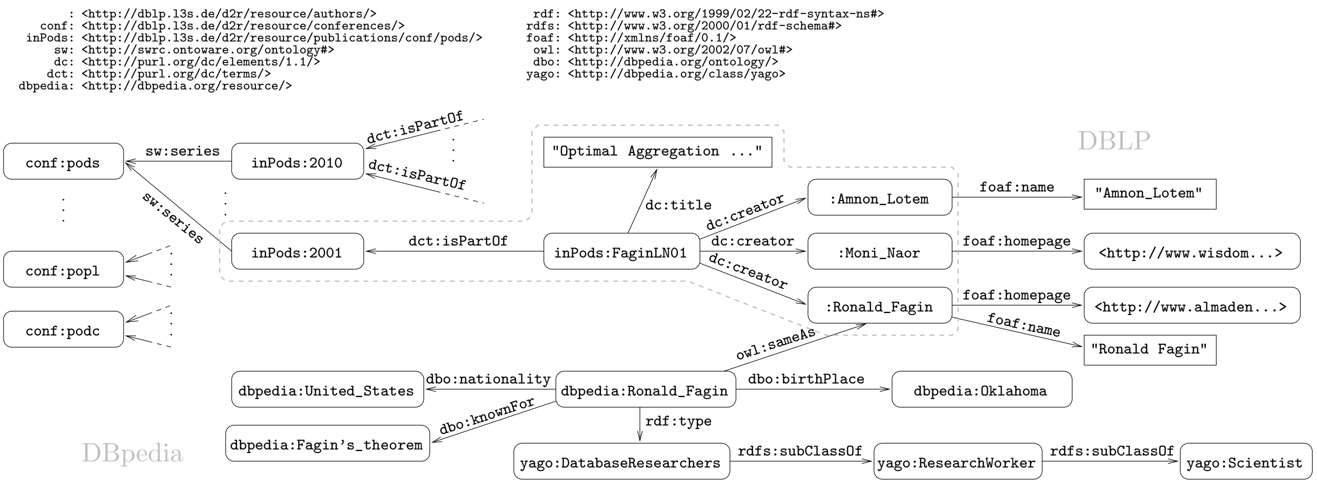
: <http://dblp.l3s.de/d2r/resource/authors/>
conf: <http://dblp.l3s.de/d2r/resource/conferences/>
inPods: <http://dblp.l3s.de/d2r/resource/publications/conf/pods/>
sw: <http://swrc.ontoware.org/ontology#>
dc: <http://purl.org/dc/elements/1.1/>
dct: <http://purl.org/dc/terms/>
dbpedia: <http://dbpedia.org/resource/>
rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
rdfs: <http://www.w3.org/2000/01/rdf-schema#>
foaf: <http://xmlns/foaf/0.1/>
owl: <http://www.w3.org/2002/07/owl#>
dbo: <http://dbpedia.org/ontology/>
yago: <http://dbpedia.org/class/yago>
DBLP
conf:pods
sw:series
inPods:2010
dct:isPartOf
dct:isPartOf
"Optimal Aggregation ..."
sw:series
dc:title
dc:creator
:Amnon_Lotem
foaf:name
"Amnon_Lotem"
conf:popl
inPods:2001
dct:isPartOf
inPods:FaginLN01
dc:creator
:Moni_Naor
foaf:homepage
<http://www.wisdom...>
dc:creator
conf:podc
:Ronald_Fagin
foaf:homepage
<http://www.almaden...>
foaf:name
"Ronald Fagin"
owl:sameAs
dbpedia:United_States
dbo:nationality
dbpedia:Ronald_Fagin
dbo:birthPlace
dbpedia:Oklahoma
dbo:knownFor
dbpedia:Fagin's_theorem
rdf:type
DBpedia
yago:DatabaseResearchers
rdfs:subClassOf
yago:ResearchWorker
rdfs:subClassOf
yago:Scientist

Figure 1: An RDF graph storing DBLP and DBpedia information.

complex RDF *graph pattern-matching expression*, and (2) a *head*, which is an expression that indicates how to construct the answer to the query. The following is an example of a simple SPARQL query that retrieves the authors that have published in PODS and were born in Oklahoma (prefixes are the same as in Figure 1). The body of the query is just a set of triples with variables, which are prefixed by the symbol ?, and the head is a projection (via a SELECT operator):

```
SELECT ?Author
WHERE {
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:isPartOf   ?InPods .
  ?InPods     sw:series      conf:pods .
  ?DbpPerson  owl:sameAs     ?Author .
  ?DbpPerson  dbo:birthPlace dbpedia:Oklahoma .  }
```

The evaluation of a query as the one shown above is done in two steps. First, values are given to variables in such a way that the triples in the body of the query match the triples in the queried graph. Second, the values assigned to the variables are processed to construct the answer. In this case, some variables are projected out to keep only the authors. According to the RDF graph in Figure 1, `:Ronald_Fagin` is an answer to this query. SPARQL also considers the possibility of adding optional information. For example, the following query retrieves the authors that have published in PODS, and their Web Pages if this information is available:

```
SELECT ?Author ?WebPage
WHERE {
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:isPartOf   ?InPods .
  ?InPods     sw:series      conf:pods .
  OPTIONAL { ?Author  foaf:homePage  ?WebPage . }  }
```

In our example graph, the answer to the query is:

?Author	?WebPage
:Ronald_Fagin	<http://www.almaden...>
:Moni_Naor	<http://www.wisdom...>
:Amnon_Lotem	

Notice that `?WebPAge` is *unbounded* in the last answer. One distinctive feature of SPARQL is that one naturally obtains partial answers to queries. With SPARQL one can also make queries with complex join sequences. For example, the following query retrieves the authors that have published in PODS, PODC, and POPL conferences.[1]

```
SELECT ?Author
WHERE {
  ?Paper1      dc:creator      ?Author .
  ?Paper1      dct:isPartOf    ?InPods .
  ?InPods      sw:series       conf:pods .

  ?Paper2      dc:creator      ?Author .
  ?Paper2      dct:isPartOf    ?InPodc .
  ?InPodc      sw:series       conf:podc .

  ?Paper3      dc:creator      ?Author .
  ?Paper3      dct:isPartOf    ?InPopl .
  ?InPopl      sw:series       conf:popl .  }
```

In Section 2, we give a formalization of RDF and SPARQL that follows the approach proposed in [32, 33], and we present some results regarding the complexity of evaluating SPARQL queries. In Section 3, we study the expressiveness of SPARQL and, in particular, we put forward the question of whether SPARQL is the right query language for RDF data. As opposed to the usual database applications, the semantics of RDF is open world, making RDF databases inherently incomplete. Nevertheless, SPARQL has adopted a semantics based on a closed world assumption. Thus, we explore in Section 3 the relationship between the open nature of RDF data and the closed-world semantics underlying SPARQL.

Vocabulary with predefined semantics: One of the distinctive features of the Semantic Web data is the existence of vocabularies with predefined semantics: the RDF *Schema* (RDFS [12]) and the *Ontology Web Language* (OWL [31]). Both languages can be used to derive logical conclusions from RDF graphs. For example, we have used some RDFS and OWL special URIs in Figure 1 [2]: `rdf:type` to denote that Ronald Fagin is a Database Researcher, and `rdfs:subClassOf` to denote that the class of Database Researchers is a subclass of Research Worker, which in turn is a subclass of Scientist. The RDFS semantics allows one to conclude from these explicit data that Ronald Fagin is a Scientist. Notice that we

[1] In fact, one can execute this query against real DBLP data at `http://dblp.l3s.de/d2r/`, and obtain 17 authors (among them, Philip Bernstein, Jeffrey Ullman, and Moshe Vardi).

[2] Under the prefixes `rdf:`, `rdfs:` and `owl:`.

have also used `owl:sameAs` to indicate that the URIs used by DBpedia and DBLP to represent Ronald Fagin denote the same resource. Consider now the following SPARQL query:

```
SELECT ?Author
WHERE {
  ?Author     rdf:type       yago:Scientist .
  ?Author     dbo:birthPlace dbpedia:Oklahoma .
  ?Paper      dc:creator     ?Author .
  ?Paper      dct:isPartOf   ?InPods .
  ?InPods     sw:series      conf:pods . }
```

This query intuitively asks for scientists that were born in Oklahoma and that have published in PODS. When matching the body of the above query against the graph in Figure 1, one obtains no valid binding of the variables. In fact, there is no triple in the graph with `rdf:type` as predicate and `yago:Scientist` as object, thus no URI assigned to variable `?Author` could make the first triple in the query to match the graph. Nevertheless, if one considers the predefined semantics of RDFS and OWL (in particular, the semantics of `rdf:type`, `rdfs:subClassOf`, and `owl:sameAs`), then one obtains `:Ronald_Fagin` as an answer to the query. Evaluating queries considering the semantics of RDFS and OWL is challenging, and there is not yet consensus in the Semantic Web community on how to deal with this problem [39, 37, 35, 17]. In Section 5, we present a simple approach for querying RDFS data that was proposed in [18, 29, 34]. Dealing with queries that consider the OWL semantics is a very interesting topic of research [39, 17, 25], but it is out of the scope of this paper.

Navigating RDF graphs: As with any data model, a natural question is what are the desired features for an RDF 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 an explicit tree or graph structure (like XML and RDF). However, SPARQL only provides limited navigational functionalities, as recognized by the W3C and the working group behind the upcoming specification of SPARQL (SPARQL 1.1 [19]). In Section 5.2, we introduce the notion of *nested regular expression*, which was proposed in [34] as a language to specify how to navigate RDF data. Interestingly, these expressions allows one to pose many natural queries, which do not seem to be expressible even in classical path languages such as *Conjunctive Regular Path Queries* (CRPQs) [10, 14] or the recently proposed language of *Extended* CRPQs [5]. For example, there exists a nested regular expression that can retrieve from the graph in Figure 1 the complete network of co-authorship of PODS papers: the expression retrieves all the pairs (a_1, a_n) of authors for which there exists a list of authors $a_2, \ldots, a_{n-1}$ such that a_i is a co-author of a_{i+1} $(1 \leq i < n)$ in a PODS paper (see Example 5.1 for details). Another interesting property of nested regular expressions is that they can be used to answer queries considering the semantics of RDFS by directly traversing the input graph. In fact, this was the motivation in [34] to propose nested regular expressions. We also discuss this issue in Sections 5.2 and 5.3, and present the language nSPARQL [34] which is obtained from SPARQL by including nested regular expressions.

Linked Data: Most of the research on RDF and SPARQL assumes a classical database perspective in which the RDF data reside in a single repository to which queries have full access. This assumption is becoming outdated with the advent of the *Web of Linked Data* [8]. The Web of Linked Data is the materialization of a set of principles for publishing Semantic Web data [6]. Essentially, these principles state that each piece of data should be published as a Web-accessible URI in such a way that when this URI is accessed by some application, an RDF specification describing the URI should be provided, which in turn should point to other accessible URIs. URIs satisfying this principle are called *dereferenceable*. For example,

`http://dblp.l3s.de/d2r/resource/publications/conf/pods/FaginLN01`

is a dereferenceable URI currently published (April 2011) as Linked Data. In fact, its description contains the links inside the dashed area in Figure 1.

Linked Data poses many interesting research questions. In particular, there is still plenty of room for research on the foundations of querying Linked Data, although some progress has been made in the area [20, 21, 9]. One particular open issue is the definition of the right querying model for this highly distributed scenario, as it is not clear whether SPARQL can deal with the features of Linked Data. Scalability issues are also an imperative topic of research in this context. As of May 2009, the Web of Linked Data was estimated to contain more than 4.7 billion RDF triples interlinked by approximately 142 million RDF links [8]. Today, just the DBpedia project [13] describe more than 3.5 million things (URIs), which descriptions sum-up to more than 600 million RDF triples containing 6.5 millions of links to other RDF stores. In Section 6, we briefly describe Linked Data and some of the research opportunities in this area.

Most of the material of this paper is the result of a fruitful collaboration with our co-authors Claudio Gutierrez, Carlos Hurtado, Alberto Mendelzon, and Sergio Muñoz. We also want to thank Juan Sequeda for many insightful discussions about Linked Data. Arenas was supported by FONDECYT grant 1090565.

2. RDF AND SPARQL

The RDF specification considers two types of values: *resource identifiers* (in the form of URIs [7]) to denote Web resources, and *literals* to denote values such as natural numbers, Booleans, and strings. In this paper, we use **U** to denote the set of all URIs and **L** to denote the set of all literals, and we assume that these two sets are disjoint. RDF considers also a special type of objects to describe *anonymous resources*, called blank nodes in the RDF data model. Essentially, blank nodes are existentially quantified variables that can be used to make statements about unknown (but existent) resources. In this paper, we do not consider blank nodes, that is, we focus on what are called *ground* RDF graphs. Formally, an RDF triple is a tuple:

$$(s, p, o) \quad \in \quad \mathbf{U} \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L}),$$

where s is the *subject*, p the *predicate* and o the *object*. An RDF graph is a finite set of RDF triples.

Figure 1 shows an example of an RDF graph with data from DBpedia [13] and the RDF version of DBLP [24]. For example, the graph states that Ronald Fagin is the author of a paper with title “Optimal Aggregation Algorithms for Middleware” by using the triples:

```
    (inPods:FaginLN01, dc:creator, :Ronald_Fagin)
(inPods:FaginLN01, dc:title, "Optimal Aggregation ...")
```

For simplicity, and when there is no ambiguity, we will omit the *prefixes* in URIs, and thus, we denote the last two triples simply as:

$$\text{(FaginLN01, creator, Ronald\_Fagin)}$$
$$\text{(FaginLN01, title, ``Optimal Aggregation \ldots'')}$$

Jointly with the release of RDF in 1999 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 [15]. In 2004, the RDF Data Access Working Group released a first public working draft of a query language for RDF, called SPARQL [37]. Currently, SPARQL is a W3C recommendation, and has become the standard language for querying RDF data. In this section, we give an algebraic formalization of the core fragment of SPARQL. Following [32, 33], we focus on the body of SPARQL queries, i.e., in its graph pattern matching facility.

2.1 Syntax of SPARQL graph patterns

The official syntax of SPARQL [37] considers operators `OPTIONAL`, `UNION`, and `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, we present the syntax of SPARQL graph patterns in a more traditional algebraic formalism following [32, 33]. More specifically, we use binary operators AND (`.`), UNION (`UNION`), OPT (`OPTIONAL`), and FILTER (`FILTER`), and we fully parenthesize expressions making explicit the precedence and association of operators.

Let $\mathbf{V}$ be an infinite set of variables disjoint from $\mathbf{U}$ and $\mathbf{L}$. In this paper, we assume that the elements from $\mathbf{V}$ are prefixed by the symbol ?. Then SPARQL graph patterns are recursively defined as follows:

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

A SPARQL built-in condition is a Boolean combination of terms constructed by using equality (=) among elements in $(\mathbf{U} \cup \mathbf{L} \cup \mathbf{V})$, and the unary predicate bound over variables. Formally,

- if $?X$, $?Y \in \mathbf{V}$ and $c \in (\mathbf{U} \cup \mathbf{L})$, then $\text{bound}(?X)$, $?X = c$ and $?X = ?Y$ are built-in conditions; and
- 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.

Example 2.1. The following is a SPARQL graph pattern

$$\Big(\big((?P, \text{creator}, ?A) \text{ AND } (?A, \text{name}, ?N)\big) \\ \text{FILTER } (?N = \text{``Ronald Fagin''})\Big). \quad (1)$$

Intuitively, it retrieves the papers of an author with name "Ronald Fagin". □

Given a graph pattern P, we denote by $\text{var}(P)$ the set of variables occurring in P. In particular, if t is a triple pattern, then $\text{var}(t)$ denotes the set of variables occurring in the components of t. Similarly, for a built-in condition R, we use $\text{var}(R)$ to denote the set of variables occurring in R.

2.2 Semantics of SPARQL graph patterns

To define the semantics of SPARQL graph pattern expressions, we need to introduce some terminology from [32, 33]. A *mapping* μ is a partial function $\mu : \mathbf{V} \to (\mathbf{U} \cup \mathbf{L})$. Abusing notation, for a triple pattern t such that $\text{var}(t) \subseteq \text{dom}(\mu)$, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, denoted by $\text{dom}(\mu)$, is the subset of $\mathbf{V}$ where μ is defined. Two mappings μ_1 and μ_2 are *compatible* when for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, 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:

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

Moreover, we define the left outer-join as:

$$\Omega_1 \mathbin{⟕} \Omega_2 \;=\; (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).$$

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 $\Omega_1 \mathbin{⟕} \Omega_2$ 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).

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

Definition 2.2 ([32, 33]) *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 $[\![P]\!]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(t)$ 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 $[\![P]\!]_G = [\![P_1]\!]_G \mathbin{⟕} [\![P_2]\!]_G$.*
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 graph patterns. Consider graph 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. 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.

It is evident from the definitions of the operators AND and UNION that these two operators are associative and commutative, thus permitting us to avoid parenthesis when writing sequences of either AND operators or UNION operators. In the following sections, we show some other algebraic properties of graph patterns.

The semantics of filter expressions is defined as follows. Given a mapping μ and a built-in condition R, we say that μ satisfies R, denoted by $\mu \models R$, if (omitting the usual rules for boolean connectives): (1) R is bound$(?X)$ and $?X \in \text{dom}(\mu)$; (2) R is $?X = c$, $?X \in \text{dom}(\mu)$ and $\mu(?X) = c$; and (3) R is $?X =?Y$, $?X \in \text{dom}(\mu)$, $?Y \in \text{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$. Then we have that:

Definition 2.3 ([32, 33]) *Given an RDF graph G and a filter expression $(P \text{ FILTER } R)$,*

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

From now on, 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

	$?X$	$?Y$	$?Z$
μ_1 :	a	b	
μ_2 :		c	

represents the set $\Omega = \{\mu_1, \mu_2\}$ where (1) $\text{dom}(\mu_1) = \{?X, ?Y\}$, $\mu_1(?X) = a$, and $\mu_1(?Y) = b$, and (2) $\text{dom}(\mu_2) = \{?Y\}$ and $\mu_2(?Y) = c$. Sometimes we also use notation $\{\{?X \to a, ?Y \to b\}, \{?Y \to c\}\}$ for the above set of mappings.

Example 2.4. The following are examples of SPARQL graph patterns and their evaluations over the RDF graph in Figure 1.

$$P_1 = \big((\text{FaginLN01}, \text{creator}, ?A) \text{ AND } (?A, \text{name}, ?N)\big)$$

$?A$	$?N$
Ronald_Fagin	"Ronald Fagin"
Amnon_Lotem	"Amnon Lotem"

$$P_2 = \big[\big((\text{FaginLN01}, \text{creator}, ?A) \text{ OPT } (?A, \text{name}, ?N)\big) \\ \text{OPT } (?A, \text{homePage}, ?H)\big]$$

$?A$	$?N$	$?H$
Ronald_Fagin	"Ronald Fagin"	http://www.almaden...
Amnon_Lotem	"Amnon Lotem"	
Moni_Naor		http://www.wisdom...

$$P_3 = \big[(\text{FaginLN01}, \text{creator}, ?A) \text{ OPT } \\ \big((?A, \text{name}, ?N) \text{ OPT } (?A, \text{homePage}, ?H)\big)\big]$$

$?A$	$?N$	$?H$
Ronald_Fagin	"Ronald Fagin"	http://www.almaden...
Amnon_Lotem	"Amnon Lotem"	
Moni_Naor		

2.3 The complexity of evaluating 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 survey of the results on the complexity of the evaluation of SPARQL graph patterns. 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 [32], and that this high complexity is obtained as a consequence of unlimited use of nested optional parts.

As is customary when studying the complexity of the evaluation problem for a query language [40], 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	:	Does $\mu \in [\![P]\!]_G$?

Notice that the pattern and the graph are both input for Evaluation. Thus, we study the *combined complexity* of the query language [40].

We start this study by considering the fragment consisting of graph pattern expressions constructed by using only the operators AND and FILTER. In what follows, we call AND-FILTER to this fragment [3]. 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 [32]. 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 2.5 ([32, 33]) Evaluation *can be solved in time $O(|P| \cdot |G|)$ for the* AND-FILTER *fragment of* SPARQL.

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 was one of the most controversial issues in the definition of this language. The following theorem shows that the inclusion of this operator makes the evaluation problem for SPARQL considerably harder.

Theorem 2.6 ([32, 33]) Evaluation *is* NP*-complete for the* AND-FILTER-UNION *fragment of* SPARQL.

In [38], the authors strengthen the above result by showing that the complexity of evaluating graph pattern expressions

[3] We use a similar notation for other combinations of SPARQL operators. For example, the AND-FILTER-UNION fragment of SPARQL is the fragment consisting of all the graph patterns constructed by using only the operators AND, FILTER and UNION.

constructed by using only AND and UNION operators is already NP-hard. Thus, we have the following result.

Theorem 2.7 ([38]) EVALUATION *is* NP*-complete for the* AND-UNION *fragment of* SPARQL.

We now consider the OPT operator. The following theorem proved in [32] shows that when considering all the operators in SPARQL graph patterns, the evaluation problem becomes considerably harder.

Theorem 2.8 ([32, 33]) EVALUATION *is* PSPACE*-complete.*

To prove the PSPACE-hardness of EVALUATION, the authors show in [33] 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 [16]. In the encoding presented in [33], 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 [38] how to encode formulas φ and ψ by using only the OPT operator, thus strengthening Theorem 2.8.

Theorem 2.9 ([38]) EVALUATION *is* PSPACE*-complete even for the* OPT *fragment of* SPARQL.

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 [40], 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 2.10 ([33]) EVALUATION(P) *is in* LOGSPACE *for every* SPARQL *graph pattern expression* P.

3. IS SPARQL THE RIGHT QUERY LANGUAGE FOR RDF?

As we pointed out in the introduction, one of the distinctive features of RDF is the fact that its semantics is open world, making RDF databases inherently incomplete. More precisely, if we are given an RDF graph G, we know that all the tuples in G hold, but we have no information about the tuples that are not included in this graph. This gives rise to the following notion of interpretation of an RDF graph G: If H is an RDF graph such that $G \subseteq H$, then H is a possible *interpretation* of G as all the tuples in G also hold in H.

The open nature of RDF leads to an infinite number of possible interpretations for each RDF graph. However, the semantics of SPARQL proposed by the W3C [37, 32, 33], which is presented in Section 2, does not take into consideration these possible interpretations. Thus, it is natural to ask whether the semantics of SPARQL is appropriate for the open-world semantics of RDF. In this section, we provide both positive and negative answers to this question, depending on what SPARQL operators are considered. In particular, we show in Section 3.1 that the semantics of SPARQL is appropriate for the open-world semantics of RDF if the OPT operator is not considered, and we show in Section 3.2 that the situation is completely different if OPT is used.

3.1 A positive answer: SPARQL without the OPT operator

As we mentioned above, the open nature of an RDF graph G leads to an infinite number of possible interpretations of G, which immediately raises the question of how a query can be answered over such a graph. The standard way to deal with this problem is to consider the answers to a query that hold in every such interpretation. Formally, given an RDF graph G and a SPARQL graph pattern P, define the set of certain answers of P over G as follows:

$$\textsc{CertainAnswers}(P, G) \quad = \bigcap_{H \,:\, G \subseteq H} [\![P]\!]_H .$$

According to the semantics of SPARQL defined by the W3C [37, 32, 33], the answer to a query over an RDF graph G should be computed by considering only the triples in G (without taking into account the possible interpretations of G). Thus, it is natural to ask what is the relationship of this semantics with the certain answers semantics defined above. Next we show that these two semantics coincide for the AND-FILTER-UNION fragment of SPARQL.

In order to prove the equivalence of the two semantics, we first provide a simple characterization of the notion of certain answers in terms of the notion of monotonicity of a query language. Following the usual database terminology, a SPARQL graph pattern P is said to be monotone if for every pair G_1, G_2 of RDF graphs such that $G_1 \subseteq G_2$, it holds that $[\![P]\!]_{G_1} \subseteq [\![P]\!]_{G_2}$. Then we have that:

Proposition 3.1 *A* SPARQL *graph pattern P is monotone if and only if* $[\![P]\!]_G = \textsc{CertainAnswers}(P, G)$ *for every RDF graph G.*

It is straightforward to prove that every graph pattern in the AND-FILTER-UNION fragment of SPARQL is monotone.

Lemma 3.2 *Every graph pattern P in the* AND-UNION-FILTER *fragment of* SPARQL *is monotone.*

Thus, as a corollary of Proposition 3.1 and Lemma 3.2, we obtain the following result:

Theorem 3.3 *Let P be a graph pattern in the* AND-UNION-FILTER *fragment of* SPARQL*. Then for every RDF graph G, it holds that* $[\![P]\!]_G = \textsc{CertainAnswers}(P, G)$.

That is, when the semantics of SPARQL is restricted to the AND-UNION-FILTER fragment, it properly handles the possible interpretations of an RDF graph. In fact, Theorem 3.3 tells one that in order to compute the certain answers of a graph pattern P in this fragment over an RDF graph G, it is enough to compute the answers to P over G according to the semantics proposed by the W3C.

3.2 A negative answer: The OPT operator

In the previous section, we show that the open-world semantics of RDF is properly handled by the semantics of SPARQL when restricted to the AND-UNION-FILTER fragment of this query language. This fact, which is shown in

Theorem 3.3, holds essentially because the AND-UNION-FILTER fragment of SPARQL is a positive query language; because of monotonicity, it cannot represent a difference operator between sets of mappings (like the MINUS operator of SQL). Thus, one may be tempted to think that a graph pattern including the OPT operator should also satisfy Theorem 3.3, as OPT is, at least conceptually, a positive operator. However, we show in this section that the inclusion of the OPT operator leads to a completely different scenario.

In the previous section, it was shown that the notion of monotonicity characterizes the equivalence of the certain answers semantics with the semantics of SPARQL proposed by the W3C. However, this notion of monotonicity is not appropriate for the OPT operator, as shown in the following example.

Example 3.4. Assume that G_1 is the RDF graph shown in Figure 1, and let G_2 be a graph obtained by adding to G_1 the triple (Ronald_Fagin, email, ron@fagin.com). Moreover, let P be the following SPARQL graph pattern:

$$(\text{Ronald\_Fagin}, \text{homepage}, ?X) \text{ OPT } (\text{Ronald\_Fagin}, \text{email}, ?Y). \quad (2)$$

The answer to this query in G_1 is $\{\mu_1\}$, where μ_1 is the mapping $\{?X \to$ <http://www.almaden...>$\}$, while the answer to this query in G_2 is $\{\mu_2\}$, where μ_2 is the mapping $\{?X \to$ <http://www.almaden...>, $?Y \to$ ron@fagin.com$\}$. Hence, we have that $[\![P]\!]_{G_1} \not\subseteq [\![P]\!]_{G_2}$ (since $\mu_1 \notin [\![P]\!]_{G_2}$), from which we conclude that P is not monotone as $G_1 \subseteq G_2$. □

Graph pattern (2) in the previous example is not monotone. However, one can claim that this pattern is at least monotone in terms of the retrieved information; for the graphs G_1, G_2 given in Example 3.4, we have that the information in $[\![P]\!]_{G_1}$ is contained in the information in $[\![P]\!]_{G_2}$, as the latter set includes not only the homepage of Ronald_Fagin but also his email. This idea gives rise to a weaker notion of monotonicity that is more appropriate for the study of the OPT operator. Formally, given mappings μ_1 and μ_2, we say that μ_1 is subsumed by μ_2, denoted by $\mu_1 \preceq \mu_2$, if $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ and $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1)$. Moreover, given sets of mappings Ω_1 and Ω_2, we say that Ω_1 is subsumed by Ω_2, denoted by $\Omega_1 \sqsubseteq \Omega_2$, if for every for every $\mu_1 \in \Omega_1$, there exists $\mu_2 \in \Omega_2$ such that $\mu_1 \preceq \mu_2$. Then a SPARQL graph pattern P is said to be *weakly monotone* if for every pair G_1, G_2 of RDF graphs such that $G_1 \subseteq G_2$, it holds that $[\![P]\!]_{G_1} \sqsubseteq [\![P]\!]_{G_2}$. For instance, graph pattern (2) is weakly monotone.

The notion of weak monotonicity captures the intuition that if some extra triples are included in an RDF graph, then the use of the operator OPT should allows one to extend some mappings without losing any information. Thus, in order to answer the question of whether the semantics of SPARQL properly handles the open-world semantics of RDF, and following the approach developed in Section 3.2, we now look for a notion of certain answers that is properly characterized by the notion of weak monotonicity. Formally, let P be a SPARQL graph pattern, G an RDF graph, Ω a set of mappings and Υ a family of sets of mappings. Then Ω is said to be a lower bound of Υ w.r.t the preorder $\sqsubseteq$ if $\Omega \sqsubseteq \Omega'$ for every $\Omega' \in \Upsilon$, and Ω is said to be a greatest lower bound of Υ w.r.t. $\sqsubseteq$ if Ω is a lower bound of Υ w.r.t. $\sqsubseteq$, and for every other lower bound Ω' of Υ w.r.t. $\sqsubseteq$, it holds that $\Omega' \sqsubseteq \Omega$. In the following proposition, we show that there exists a tight connection between the notions of weak monotonicity and greatest lower bound w.r.t. $\sqsubseteq$.

Proposition 3.5 *A* SPARQL *query P is weakly monotone if and only if for every RDF graph G, it holds that $[\![P]\!]_G$ is a greatest lower bound of $\{[\![P]\!]_H \mid G \subseteq H\}$ w.r.t. $\sqsubseteq$.*

Proposition 3.5 gives us a natural definition of what it means for a SPARQL graph pattern to properly handle the open-world semantics of RDF: Given a SPARQL graph pattern P and an RDF graph G, $[\![P]\!]_G$ should be the largest set in terms of information content that is contained in the answer to P over every possible interpretation of G. Thus, we have a criteria to formally answer the question of whether the semantics of SPARQL is appropriate for the open nature of RDF. Unfortunately, the answer to this question is negative as shown in the following example.

Example 3.6. Assume that G_1 is the RDF graph shown in Figure 1, and let G_2 be a graph obtained by adding to G_1 the triple (Ronald_Fagin, email, ron@fagin.com). Moreover, let P be the following SPARQL graph pattern:

$$\Bigg[(?X, \text{ name, ``Moni Naor''}) \text{ AND } \bigg(\big((?Y, \text{ name, ``Ronald Fagin''}) \text{ OPT } (?X, \text{ email, } ?Z)\big)\bigg)\Bigg] \quad (3)$$

Graph pattern P stores in variable $?X$ a URI with name "Moni Naor", in variable $?Y$ a URI with name "Ronald Fagin", and it optionally stores in $?Z$ the email of the URI stored in $?X$. The answer to this query in G_1 is $\{\mu_1\}$, where μ_1 is the mapping $\{?X \to \text{Moni\_Naor}, ?Y \to \text{Ronald\_Fagin}\}$. The answer to this query in G_2 is the empty set of mappings since (a) the answer to the triple pattern (?X, name, "Moni Naor") over G_2 is $\{\nu_1\}$, where ν_1 is the the mapping $\{X \to \text{Moni\_Naor}\}$, (b) the answer to the graph pattern ((?Y, name, "Ronald Fagin") OPT (?X, email, ?Z)) over G_2 is $\{\nu_2\}$, where ν_2 is the mapping $\{?Y \to \text{Ronald\_Fagin}, ?X \to \text{Ronald\_Fagin}, ?Z \to \text{ron@fagin.com}\}$, and (c) ν_1 is not compatible with ν_2. Hence, we have that $[\![P]\!]_{G_1} \not\sqsubseteq [\![P]\!]_{G_2}$, from which we conclude that P is not weakly monotone as $G_1 \subseteq G_2$. □

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 match. However, this intuition fails in the simple graph pattern (3) given in Example 3.6, as this pattern is not weakly monotone. And not only that, it can also be shown that the intuition that SPARQL is a positive query language fails, as OPT can be used to express a difference operator between sets of mappings. More precisely, given SPARQL graph patterns P_1, P_2 and an RDF graph G, define operator MINUS as $[\![(P_1 \text{ MINUS } P_2)]\!]_G = [\![P_1]\!]_G \smallsetminus [\![P_2]\!]_G$, where $\smallsetminus$ is the difference operator between sets of mappings defined in Section 2.2. Then we have that:

Proposition 3.7 ([3]) *Let P_1 and P_2 be* SPARQL *graph patterns. Then (P_1* MINUS *P_2) is equivalent to:*

$$\Bigg[\bigg(P_1 \text{ OPT } (P_2 \text{ AND } (?X_1, ?X_2, ?X_3))\bigg) \text{ FILTER } \neg\, \text{bound}(?X_1)\Bigg],$$

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

In light of these negative results, we present in Section 4 a fragment of SPARQL that is defined by a syntactic restriction (proposed in [32]) over the scope of the variables used in the OPT operator. The graph patterns in this fragment are shown to be weakly monotone in Section 4 and, thus, the semantics of these patterns is appropriate for the open nature of RDF.

4. WELL-DESIGNED GRAPH PATTERNS

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 this 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 match. However, this intuition fails in some simple examples.

EXAMPLE 4.1. Let P_1 be the graph pattern:

$$((?X, \text{name}, \text{“Moni Naor”}) \text{ AND } (?Y, \text{name}, \text{“Ronald Fagin”})),$$

and P_2 be the graph pattern (3), which is obtained by replacing $(?Y$, name, “Ronald Fagin”) in P_1 by optional pattern $((?Y$, name, “Ronald Fagin”) OPT $(?X$, email, $?Z))$. Moreover, let G be the RDF graph obtained by adding the triple (Ronald_Fagin, email, ron@fagin.com) to the RDF graph in Figure 1. It is easy to see that $\llbracket P_1 \rrbracket_G = \{\mu_1\}$, where μ_1 is the mapping $\{?X \to \text{Moni\_Naor}, ?Y \to \text{Ronald\_Fagin}\}$. On the other hand, although P_2 is obtained by adding optionality to P_1, it is shown in Example 3.6 that $\llbracket P \rrbracket_G$ is the empty set of mappings □

The graph pattern in the previous example is unnatural as the triple pattern $(?X, \text{email}, ?Z)$ seems to be giving optional information for $(?X, \text{name}, \text{Moni Naor”})$ (they share variable $?X$), but in the graph pattern appears as giving optional information for $(?Y, \text{name}, \text{“Ronald Fagin”})$. In fact, it is possible to find a common pattern in the examples that contradict the intuition behind the definition of the OPT operator: 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 [32], the authors focus on the AND-FILTER-OPT fragment of SPARQL and introduce a syntactic restriction that forbids the form of interaction between variables discussed above. Next we present this restriction, for which we need to introduce some terminology. A graph pattern Q is said to be *safe* if for every sub-pattern $(P \text{ FILTER } R)$ of Q, it holds that $\text{var}(R) \subseteq \text{var}(P)$.

Definition 4.2 ([32]) *Let P be a graph pattern in the* AND-FILTER-OPT *fragment of* SPARQL. *Then P is* well designed *if (1) P is safe, and (2) for every sub-pattern $P' = (P_1 \text{ OPT } P_2)$ of P and variable $?X$, if $?X$ occurs both inside P_2 and outside P', then it also occurs in P_1.*

For instance, pattern (3) in Example 3.6 is not well designed.

The condition of being well-designed imposes a natural restriction over the scope of variables in the OPT operator. In [32], the notion of being well designed was introduced in an attempt to regulate the scope of variables in the OPT operator and, in particular, to forbid some unnatural SPARQL graph patterns that violate the intuition behind the definition of the OPT operator. Next we show that this restriction also leads to weak monotonicity, thus showing that well-designed graph patterns are appropriate for the open-world assumption underlying RDF.

1. Sub-property:
   (a) $\dfrac{(\mathcal{A},\texttt{sp},\mathcal{B}) \;\; (\mathcal{B},\texttt{sp},\mathcal{C})}{(\mathcal{A},\texttt{sp},\mathcal{C})}$
   (b) $\dfrac{(\mathcal{A},\texttt{sp},\mathcal{B}) \;\; (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\mathcal{B},\mathcal{Y})}$
2. Subclass:
   (a) $\dfrac{(\mathcal{A},\texttt{sc},\mathcal{B}) \;\; (\mathcal{B},\texttt{sc},\mathcal{C})}{(\mathcal{A},\texttt{sc},\mathcal{C})}$
   (b) $\dfrac{(\mathcal{A},\texttt{sc},\mathcal{B}) \;\; (\mathcal{X},\texttt{type},\mathcal{A})}{(\mathcal{X},\texttt{type},\mathcal{B})}$
3. Typing:
   (a) $\dfrac{(\mathcal{A},\texttt{dom},\mathcal{B}) \;\; (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\texttt{type},\mathcal{B})}$
   (b) $\dfrac{(\mathcal{A},\texttt{range},\mathcal{B}) \;\; (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{Y},\texttt{type},\mathcal{B})}$

Table 1: RDFS inference rules.

Theorem 4.3 *Every well-designed graph pattern is weakly monotone.*

Well-designed graph patterns also have good properties regarding the complexity of the evaluation problem. In [32, 33, 38], it was proved that the evaluation problem for SPARQL is PSPACE-complete, even if only the OPT operator is considered [38]. In these papers, the PSPACE lower bounds were proved by using graph patterns that are 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 [33], the authors showed that this is indeed the case.

Theorem 4.4 ([33]) EVALUATION *is coNP-complete for the fragment of* SPARQL *consisting of well-designed patterns.*

It is important to notice that it was also shown in [32, 33, 9] that well-designed patterns are suitable for reordering and optimization, demonstrating the significance of this class of queries from the practical point of view.

Well-designed patterns form a well-behaved fragment of SPARQL that we think deserves future investigation. In particular, given the result in Theorem 4.3, it would be interesting to explore whether weakly monotone graph patterns coincide with well-designed graph patterns. The question of whether an AND-FILTER-OPT SPARQL pattern is weakly monotone if and only if it is well designed, remains open. Notice that we are considering only AND-FILTER-OPT SPARQL patterns, since the notion of being well-designed is defined only for this fragment. Thus, an interesting subject to explore is how to extend this notion to patterns containing the UNION operator. A first proposal is presented in [36], but the study of fundamental properties such as weak monotonicity, complexity of evaluation, and optimization has not been carried out.

5. RDFS VOCABULARY AND A NAVIGATIONAL LANGUAGE FOR RDF

The RDF specification includes a set of reserved URIs (reserved elements from $\mathbf{U}$) with predefined semantics, the RDFS vocabulary [12]. This set of reserved URIs is designed to deal with inheritance of classes and properties, as well as typing, among other features [12]. In Figure 1, we use two of these URIs, namely `rdf:type` and `rdfs:subClassOf`, when describing the resource `dbpedia:Ronald_Fagin`. Intuitively, it is possible to conclude from the graph in Figure 1 that "Ronald Fagin" is a "Scientist", as it is stated in the graph that "Ronald Fagin" is a "Database Researcher", which is a subclass of "Research Worker", which in turn is a subclass of "Scientist".

RDFS was designed to deal with the the kind of deductions shown in the previous paragraph. In this section, we present a formalization of RDFS taken from [18, 29], and then we study the problem of answering queries over RDFS data. In particular, we introduce the notion of nested regular expression in Section 5.2, which has shown to be appropriate to navigate RDF data [34], and then we introduce in Section 5.3 the query language nSPARQL, which extends SPARQL with nested regular expressions and has shown to be appropriate to deal with the RDFS vocabulary [34]. Interestingly, we also show that nSPARQL can be used to pose many interesting and natural queries over RDF data, which require of its navigational capabilities.

5.1 RDFS vocabulary

The semantics of RDFS was defined in [22] by borrowing some notions from mathematical logic. We consider here a simplified version of this semantics, which is obtained by focusing on the subset of RDFS consisting of the reserved URIs `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:range`, and `rdfs:domain`. This fragment was studied in [29], where the authors provide a formal semantics for it, and show it to be equivalent to the semantics of RDFS defined in [22] (when restricted to the five keywords just mentioned).

From now on, we use shorthands `type`, `sc`, `sp`, `range`, `dom` for the reserved URIs `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:range`, and `rdfs:domain`, respectively. The semantics of these keywords can be given in terms of the rule system shown in Table 1. In every rule, letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{X}$, $\mathcal{Y}$ stand for *variables* to be replaced by actual terms. Formally, an *instantiation* of a rule is a replacement of its variables by elements of $(\mathbf{U} \cup \mathbf{L})$. Then an RDF graph G' is said to be obtained by an *application* of a rule r to G, if there is an instantiation $\frac{R}{R'}$ of r such that $R \subseteq G$ and $G' = (G \cup R')$. Moreover, an RDF triple t can be *inferred* from G, if there exists a graph G' that is obtained from G by successively applying the rules in Table 1 and such that $t \in G'$. In [29], it is proved that the rule system given in Table1 is sound and complete for the inference problem in the presence of the vocabulary `type`, `sc`, `sp`, `range` and `dom`.

When dealing with RDFS vocabulary, a particular property that separates RDF graphs from standard graphs comes into play: edge labels may also be considered as nodes in the graph. For example, consider the RDF graph in Figure 2, which stores information about transportation services between cities. It states that TGV provides a transportation service from Paris to Calais by using the triple $(\text{Paris}, \text{TGV}, \text{Calais})$. It also states that a TGV service is a sub-property of train_service by using the triple (TGV, `sp`, train_service). Thus, TGV is simultaneously acting as an edge label and as a node label in the graph of Figure 2.

The RDF graph in Figure 2 also contains RDFS annotations to describe the relationship between transportation services. For instance, (Seafrance, `sp`, ferry_service) states that Seafrance is a sub-property of ferry_service. Thus, by using this triple and (Calais, Seafrance, Dover), we can conclude that there is a ferry going from Calais to Dover. Formally, this conclusion can be obtained by a single application of rule (1b) to triples (Seafrance, `sp`, ferry_service) and (Calais, Seafrance, Dover), from which we obtain triple (Calais, ferry_service, Dover). Moreover, by applying the rule (3b) to this last triple and (ferry_service, `range`, coastal_city), we obtain triple (Dover, `type`, coastal_city) and, thus, we conclude that Dover is a coastal city.

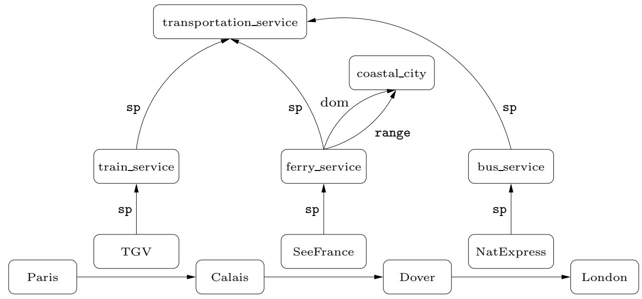

Figure 2: RDF graph storing information about cities and transportation services between cities.

For the RDFS vocabulary consisting of `type`, `sc`, `sp`, `range` and `dom`, it was shown in [29] that one can determine whether a triple is implied by a graph just by checking the existence of some paths in the graph. For example, a query that asks whether a resource A is a sub-class of a resource B can be answered just by checking the existence of a path from A to B in the graph where each edge has label `sc`. This observation has motivated the extension of SPARQL with path expressions [2, 34, 19], that allow the user to specify ways to navigate RDF graphs. For instance, in the query languages proposed in [2, 34, 19], an extended triple pattern of the form $(A, \texttt{sc}^{+}, B)$ can be used to check whether A is a sub-class of B, as $\texttt{sc}^{+}$ is a path expression denoting paths of length at least 1 and where each edge has label `sc`. In the rest of this section, we present the query language nSPARQL that was introduced in [34], and which uses the notion of *nested regular expression* to specify how to navigate an RDF graph. More specifically, nested regular expressions are introduced in Section 5.2, and nSPARQL is introduced in Section 5.3.

5.2 Nested regular expressions

The query language proposed in [34] uses, as usual for graph query languages [28, 4], regular expressions to define paths on graph structures, but taking into consideration the special features of RDF graphs. In particular, the authors extend in [34] regular expressions by borrowing the notion of *branching* from XPath [11], to obtain the language of *nested regular expressions* to specify how to navigate RDF data.

The navigation of a graph is usually done by using an operator *next*, which allows one to move from a node to an adjacent node. In our setting, we have RDF "graphs", which are sets of triples, not classical graphs. In particular, the sets of node labels and edge labels of an RDF graph can have a nonempty intersection. The notion of nested regular expression proposed in [34] 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 $\texttt{next}^{-1}$, $\texttt{edge}^{-1}$ and $\texttt{node}^{-1}$, to move through an RDF triple. These axes are shown in Figure 3.

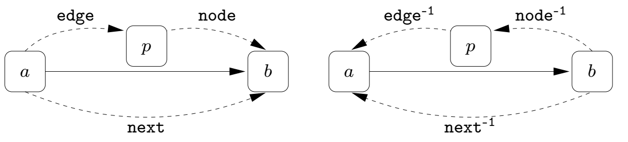

Figure 3: Axes for an RDF triple (a, p, b).

In the language of nested regular expressions, a navigation axis allows one to move one step forward (or backward) in an RDF graph and, thus, a sequence of these axes defines a path. Moreover, one can use classical regular expressions over these axes to define a set of paths that can be used in a query. Nested regular expressions also include an additional axis `self` that is used not to actually navigate, but instead to perform a test over a node in a path. Finally, 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:

$$\begin{aligned} \mathit{exp} \quad := \quad & \text{axis} \;\mid\; \text{axis::}a \ (a \in \mathbf{U}) \;\mid\; \text{axis::}[\mathit{exp}] \;\mid \\ & \mathit{exp}/\mathit{exp} \;\mid\; \mathit{exp}|\mathit{exp} \;\mid\; \mathit{exp}^* \qquad (4) \end{aligned}$$

with axis $\in \{$`self`, `next`, `next`$^{-1}$, `edge`, `edge`$^{-1}$, `node`, `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 `next`::a, with a an element from $\mathbf{U}$. Given a graph G, the expression `next`::a is interpreted as the a-*neighbor* relation in G, that is, the pairs of nodes (x, y) such that $(x, a, y) \in G$. The language also allows one to navigate from a node to one of its leaving edges by using the `edge` axis. Formally, the interpretation of `edge`::a is the pairs of nodes (x, y) such that $(x, y, a) \in G$. The nesting construction $[\mathit{exp}]$ is used to check for the existence of a path defined by expression exp. For instance, when evaluating nested expression `next`::$[\mathit{exp}]$ in G, one retrieves the pairs of nodes (x, y) such that there exists z satisfying that $(x, z, y) \in G$ and there exists a path in G starting in z and conforming to expression exp.

The evaluation of a nested regular expression exp in an RDF graph G is formalized as a binary relation $[\![\mathit{exp}]\!]_G$, denoting the pairs of nodes (x, y) such that y is reachable from x in G by following a path that conforms to exp. The semantics of the language is shown in Table 2. In this table, G is an RDF graph, $a \in \mathbf{U}$, voc(G) is the set of all the elements from $\mathbf{U}$ that are mentioned in G, "axis" represents any of the navigational axes in $\{$`next`, `node`, `edge`$\}$, and exp, exp_1, exp_2 are nested regular expressions. Notice that the semantics of our expressions considers only URIs (elements in $\mathbf{U}$), and not literals, as the navigation of an RDF graph has to be done through the actual nodes in the graph, and not through the data values. As is customary for regular expressions, given a nested regular expression exp, we use exp^+ as a shorthand for the expression $\mathit{exp}/\mathit{exp}^*$.

$[\![\texttt{self}]\!]_G$	$=$	$\{(x, x) \mid x \in \text{voc}(G)\}$
$[\![\texttt{self}::a]\!]_G$	$=$	$\{(a, a)\}$
$[\![\texttt{next}]\!]_G$	$=$	$\{(x, y) \mid \text{there exists } z \text{ s.t. } (x, z, y) \in G\}$
$[\![\texttt{next}::a]\!]_G$	$=$	$\{(x, y) \mid (x, a, y) \in G\}$
$[\![\texttt{edge}]\!]_G$	$=$	$\{(x, y) \mid \text{there exists } z \text{ s.t. } (x, y, z) \in G\}$
$[\![\texttt{edge}::a]\!]_G$	$=$	$\{(x, y) \mid (x, y, a) \in G\}$
$[\![\texttt{node}]\!]_G$	$=$	$\{(x, y) \mid \text{there exists } z \text{ s.t. } (z, x, y) \in G\}$
$[\![\texttt{node}::a]\!]_G$	$=$	$\{(x, y) \mid (a, x, y) \in G\}$
$[\![\text{axis}^{-1}]\!]_G$	$=$	$\{(x, y) \mid (y, x) \in [\![\text{axis}]\!]_G\}$
$[\![\text{axis}^{-1}::a]\!]_G$	$=$	$\{(x, y) \mid (y, x) \in [\![\text{axis}::a]\!]_G\}$
$[\![\mathit{exp}_1/\mathit{exp}_2]\!]_G$	$=$	$\{(x, y) \mid \text{there exists } z \text{ s.t. } (x, z) \in [\![\mathit{exp}_1]\!]_G$ and $(z, y) \in [\![\mathit{exp}_2]\!]_G\}$
$[\![\mathit{exp}_1 \mid \mathit{exp}_2]\!]_G$	$=$	$[\![\mathit{exp}_1]\!]_G \cup [\![\mathit{exp}_2]\!]_G$
$[\![\mathit{exp}^*]\!]_G$	$=$	$[\![\texttt{self}]\!]_G \cup [\![\mathit{exp}]\!]_G \cup [\![\mathit{exp}/\mathit{exp}]\!]_G \cup \cdots$
$[\![\texttt{self}::[\mathit{exp}]]\!]_G$	$=$	$\{(x, x) \mid x \in \text{voc}(G) \text{ and there exists } z$ s.t. $(x, z) \in [\![\mathit{exp}]\!]_G\}$
$[\![\texttt{next}::[\mathit{exp}]]\!]_G$	$=$	$\{(x, y) \mid \text{there exist } z, w \text{ s.t. } (x, z, y) \in G$ and $(z, w) \in [\![\mathit{exp}]\!]_G\}$
$[\![\texttt{edge}::[\mathit{exp}]]\!]_G$	$=$	$\{(x, y) \mid \text{there exist } z, w \text{ s.t. } (x, y, z) \in G$ and $(z, w) \in [\![\mathit{exp}]\!]_G\}$
$[\![\texttt{node}::[\mathit{exp}]]\!]_G$	$=$	$\{(x, y) \mid \text{there exist } z, w \text{ s.t. } (z, x, y) \in G$ and $(z, w) \in [\![\mathit{exp}]\!]_G\}$
$[\![\text{axis}^{-1}::[\mathit{exp}]]\!]_G$	$=$	$\{(x, y) \mid (y, x) \in [\![\text{axis}::[\mathit{exp}]]\!]_G\}$

Table 2: Semantics of nested regular expressions.

EXAMPLE 5.1. Let G be the graph in Figure 1, and consider the following nested regular expression:

$$\texttt{next}::\text{isPartOf}/\texttt{next}::\text{series}$$

The evaluation of this expression over G is the set of all pairs (x, y) such that x is a paper that was published in conference y. For example (FaginLN01, pods) is in the evaluation of the expression. One can use the `self` axis to test that the conference in the above expression is actually PODS:

$$\mathit{exp}_1 \quad = \quad \texttt{next}::\text{isPartOf}/\texttt{next}::\text{series}/\texttt{self}::\text{pods}$$

By using expression exp_1, it is easy to construct a nested regular expression that has as evaluation all the pairs (x, y) such that x and y are co-authors of a PODS paper:

$$\mathit{exp}_2 \quad = \quad \texttt{next}^{-1}::\text{creator}/\texttt{self}::[\,\mathit{exp}_1]/\texttt{next}::\text{creator}$$

Notice that `next`$^{-1}$::creator is used to navigate from an author x to a paper p created by x, then the nested expression `self`::$[\,\mathit{exp}_1]$ is used to test that p is actually a PODS paper, and then `next`::creator is used to navigate from p to one of its authors y. For example, the pair (Ronald_Fagin, Moni_Naor) is in $[\![\mathit{exp}_2]\!]_G$. Moreover, one can use the power of nested regular expressions to obtain the complete network of co-authorship of PODS papers as follows:

$$\mathit{exp}_3 \quad = \quad (\texttt{next}^{-1}::\text{creator}/\texttt{self}::[\,\mathit{exp}_1]/\texttt{next}::\text{creator})^+$$

This expression defines the set of all pairs (x_1, x_k) for which there exists a sequence $x_2, \ldots, x_{k-1}$ of authors such that x_i is a co-author of x_{i+1} $(1 \leq i < k)$ in a PODS paper. □

5.2.1 On the complexity and expressiveness of nested regular expressions

In [34], the authors study several properties of nested regular expressions. In the first place, they study the complexity of evaluating these expressions, and considered two problems. The first one is the standard decision problem considered when studying the complexity of a query language:

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

The second problem considered in [34] is the following *computation* problem associated to nested regular expressions:

Problem	:	Computation problem for nested regular expressions.
Input	:	An RDF graph G, a nested regular expression *exp*, and a node a.
Output	:	List all elements b s.t. $(a, b) \in [\![exp]\!]_G$.

It turns out that both problems can be solved efficiently.

Theorem 5.2 ([34]) *The evaluation and computation problems for nested regular expressions can be solved in time* $O(|G| \cdot |exp|)$.

In the second place, it is studied in [34] whether nested regular expressions are expressive enough to deal with the RDFS vocabulary. Interestingly, it is shown in [34] that this is indeed the case:

Theorem 5.3 ([34]) *There exists a function* Trans *from* $\mathbf{U}$ *to the set of nested regular expressions such that, for every RDF graph* G *and triple* $(a, p, b) \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$, *it holds that* (a, p, b) *can be inferred from* G *according to the RDFS semantics if and only if* $(a, b) \in [\![\text{Trans}(p)]\!]_G$.

Due to the lack of space, we do not reproduce here the complete definition of Trans (see [34] for details), and we only show in an example the intuition behind this transformation.

Example 5.4. Let G be the RDF graph in Figure 2, and consider the following nested regular expression:

$$exp \;=\; \texttt{next}::[\,(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transportation\_service}\,]$$

A pair (x, y) is in the evaluation of the expression *exp* over G if there exists a triple (x, p, y) in G and a path from p to transportation_service where every edge has label sp. Thus, nested expression $[\,(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transportation\_service}\,]$ is used to emulate the inferencing process in RDFS; it retrieves all the nodes that are *sub-properties* of transportation_service (rule (1a) in Table 1). Therefore, we have that the triple $(x, \text{transportation\_service}, y)$ can be inferred from G if and only if (x, y) is in $[\![exp]\!]_G$, which tells one that the evaluation of expression *exp* results in the set of pairs of cities that are connected by a transportation service (which can be a train service, ferry service, bus service, etc). In fact, in the example we have that $(\text{Calais}, \text{Dover})$ is in $[\![exp]\!]_G$.

We can also use nested regular expressions to obtain the pairs of cities such that there is a way to travel from one city to the other with any number of transportation services:

$$(\texttt{next}::[\,(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transportation\_service}\,])^+$$

For example, the pair $(\text{Paris}, \text{London})$ is in the evaluation of this last expression over G. □

The previous example shows the power of nested regular expressions not only to deal with the RDFS vocabulary, but also to express interesting and natural queries over RDF data (which require of navigational capabilities to be expressed).

5.3 nSPARQL

In this section, we give a brief overview of the query language nSPARQL, which was proposed in [34] as an extension of SPARQL with navigational capabilities, and was designed to be able to answer queries considering the semantics of the RDFS vocabulary.

Let the closure of a graph G, denoted by $\text{cl}(G)$, be the graph obtained from G by successively applying the rules in Table 1 until the graph does not change. Then the RDFS evaluation of a SPARQL graph pattern P over an RDF graph G, denoted by $[\![P]\!]_G^{\text{rdfs}}$, is defined as $[\![P]\!]_G^{\text{rdfs}} = [\![P]\!]_{\text{cl}(G)}$.

As we have pointed out before, a SPARQL graph pattern P treats the RDFS vocabulary without considering its predefined semantics. Thus, we have in general that $[\![P]\!]_G \neq [\![P]\!]_G^{\text{rdfs}}$, which immediately raises the question of how the semantics of RDFS could be included into SPARQL. A simple approach to do this could be based on the materialization of the closure of a graph; once this closure has been computed, a query can be directly evaluated over it. Unfortunately, this approach has some practical drawbacks; among others, the closure of a graph can be of quadratic size thus making its computation and storage too expensive. In light of these limitations, it was proposed in [34] to replace triple patterns in SPARQL by patterns using nested regular expressions, which can be used to simulate the RDFS inferencing process without computing the closure of a graph (see Theorem 5.2). The resulting language was called nSPARQL in [34]. More specifically, nSPARQL is obtained by considering triple patterns with nested regular expressions in the predicate position. The evaluation of an extended triple pattern of the form $(?X, exp, ?Y)$ over an RDF graph G is defined just as the set of mappings $\mu = \{?X \to a, ?Y \to b\}$ such that $(a, b) \in [\![exp]\!]_G$. The language nSPARQL also includes the operators AND, FILTER, UNION and OPT, whose semantics is inherited from SPARQL. In [34], the authors showed that in order to evaluate a SPARQL graph pattern P over an RDF graph G according to the RDFS semantics, one does not need to materialize the closure of G, but instead one can just rewrite P into an nSPARQL pattern that is evaluated over the initial graph G. Formally, the following theorem is proved in [34]:

Theorem 5.5 ([34]) *Let* P *be a SPARQL graph pattern constructed by considering only triple patterns without a variable in the predicate position. Then there exists an nSPARQL pattern* Q, *that can be computed in linear time from* P, *such that for every RDF graph* G: $[\![P]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$.

It should be noticed that Theorem 5.5 does not hold for the triple pattern $(?X, ?Y, ?Z)$ (which has a variable in the predicate position), as the evaluation of $(?X, ?Y, ?Z)$ over an RDF graph G according to the RDFS semantics would inevitably imply the computation of the closure of G, which is something that cannot be computed by using the navigational approach of nSPARQL [34].

6. A BIG CHALLENGE: LINKED DATA

We conclude the paper by pointing out some challenges for the database community that come from the Linked Data project [6]. Up to this point, we have modeled RDF data assuming a classical centralized database approach: the data resides in a single repository to which queries have full access. In fact, the normative specifications of RDF and SPARQL follow the same assumption [22, 37]. The *Web of Linked Data* [6, 8] is a shift towards a data model in which every piece of RDF information describes itself (describes its relations with other pieces of data) in a decentralized way. Linked Data is based on the following set of principles to publish Web data [6, 23]: (1) URIs should be used to identify *things*, (2) URIs should be Web-accessible so that people/machines can *look up* those URIs, (3) when someone/something looks up a URI, an RDF graph with *useful* information should be provided, and (4) the RDF graph should include links to other Web-accessible URIs so that more things can be *discovered*.

There are several issues posed by these principles that are not covered from a classical database point of view, among the most important: data is highly distributed and at a *fine grain*, any URI can make statements about (provide links to) any other URI, and anyone can publish anything about a resource. These issues together with the scalability issues faced when querying Web data and the distinctive features of RDF (such as the inherent incompleteness of the information in an RDF database, the graph structure of RDF and the use of vocabularies with predefined semantics) make Linked Data a very challenging scenario from a database point of view. Although there has been some work on querying Linked Data [20, 21, 9], little research has been pursued towards the fundamental problems of developing appropriate data models and query languages in this context. Linked Data can be benefited from the large body of work done in database areas such as semi-structured data [41], graph databases [4], Web data models [27, 1] and distributed databases [30]. Thus, we think that the database community has much more to say about the fundamental database problems that need to be solved in the context of the Linked Data project.

The amount of Web data published following the Linked Data principles has grown constantly since their proposal in 2006. With the advent of large projects such as `data.gov` and `data.gov.uk`, whose purposes are to increase public access to government data, we can only expect it to grow more in the future. Thus, we expect that the interest in managing highly-distributed large-scale RDF data will continue growing both in the Semantic Web and database communities.

7. REFERENCES

[1] S. Abiteboul and V. Vianu. Queries and computation on the Web. *Theor. Comput. Sci.*, 239(2):231–255, 2000.

[2] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73, 2009.

[3] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *ISWC*, pages 114–129, 2008.

[4] R. Angles and C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.

[5] P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, pages 3–14, 2010.

[6] T. Berners-Lee. Design issues: Linked Data. `http://www.w3.org/DesignIssues/LinkedData.html`, July 2006.

[7] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. `http://www.ietf.org/rfc/rfc3986.txt`, 2005.

[8] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[9] C. Buil-Aranda, M. Arenas, and O. Corcho. Semantics and optimization of the SPARQL 1.1 federation extension. In *ESWC*, 2011.

[10] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, 2000.

[11] J. Clark and S. DeRose. XML path language (XPath). W3C recommendation. `http://www.w3.org/TR/xpath`, November 2008.

[12] R.V. Guha D. Brickley. RDF vocabulary description language 1.0: RDF schema, W3C recommendation, February 2004.

[13] DBpedia. `http://dbpedia.org/`.

[14] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *DBPL*, pages 21–39, 2001.

[15] T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF querying: Language constructs and evaluation methods compared. In *Reasoning Web*, pages 1–52, 2006.

[16] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[17] B. Glimm and M. Krötzsch. SPARQL beyond subgraph matching. In *ISWC*, pages 241–256, 2010.

[18] C. Gutierrez, C. A. Hurtado, A. O. Mendelzon, and J. Pérez. Foundations of semantic web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.

[19] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C working draft. `http://www.w3.org/TR/sparql11-query/`, October 2010.

[20] O. Hartig and A. Langegger. A database perspective on consuming Linked Data on the Web. *Datenbank-Spektrum*, 10(2):57–66, 2010.

[21] O. Hartig, J. Sequeda, J. Taylor, and P. Sinclair. How to consume Linked Data on the Web: tutorial description. In *WWW*, pages 1347–1348, 2010.

[22] Pat Hayes. RDF semantics, W3C Recommendation, February 2004.

[23] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space.* Morgan & Claypool Publishers, 2011.

[24] D2R DBLP Bibliography Database hosted at L3S Research Center. `http://dblp.l3s.de/d2r/`.

[25] I. Kollia, B. Glimm, and I. Horrocks. SPARQL query answering over OWL ontologies. In *ESWC*, 2011.

[26] F. Manola and E. Miller. RDF primer, W3C recommendation, February 2004.

[27] A. O. Mendelzon and T. Milo. Formal models of Web queries. *Inf. Syst.*, 23(8):615–637, 1998.

[28] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In *VLDB*, pages 185–193, 1989.

[29] S. Muñoz, J. Pérez, and C. Gutiérrez. Minimal deductive systems for RDF. In *ESWC*, pages 53–67, 2007.

[30] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition.* Prentice-Hall, 2011.

[31] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL semantics and abstract syntax. W3C recommendation. `http://www.w3.org/TR/owl-semantics/`, February 2004.

[32] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *ISWC*, pages 30–43, 2006.

[33] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[34] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[35] A. Polleres. SPARQL1.1: New features and friends (OWL2, RIF). In *Rules and Reasoning*, pages 23–26, 2010.

[36] Axel Polleres. From SPARQL to rules (and back). In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 787–796, 2007.

[37] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation. `http://www.w3.org/TR/rdf-sparql-query/`, January 2008.

[38] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33, 2010.

[39] E. Sirin and B. Parsia. SPARQL-DL: SPARQL query for OWL-DL. In *OWLED*, 2007.

[40] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

[41] V. Vianu. A Web odyssey: from Codd to XML. *SIGMOD Record*, 32(2):68–77, 2003.

APPENDIX

A. PROOFS

A.1 Proof of Lemma 3.2

Assume that P is a SPARQL pattern not mentioning operator OPT, and let G_1, G_2 be RDF graphs such that $G_1 \subseteq G_2$. Next we show by induction on the structure of P that $[\![P]\!]_{G_1} \subseteq [\![P]\!]_{G_2}$.

- Assume that P is triple pattern t, and let $\mu \in [\![t]\!]_{G_1}$. Then we have that $\text{dom}(\mu) = \text{var}(t)$ and $\mu(t) \in G_1$. Thus, given that $G_1 \subseteq G_2$, we conclude that $\mu(t) \in G_2$, from which we conclude that $\mu \in [\![P]\!]_{G_2}$.
- Assume that $P = (P_1 \text{ AND } P_2)$ and that P_1, P_2 are monotone. Let $\mu \in [\![(P_1 \text{ AND } P_2)]\!]_{G_1}$. Then we have that $\mu = \mu_1 \cup \mu_2$, where $\mu_1 \in [\![P_1]\!]_{G_1}$ and $\mu_2 \in [\![P_2]\!]_{G_1}$. Thus, given that P_1 and P_2 are monotone, we conclude that $\mu_1 \in [\![P_1]\!]_{G_2}$ and $\mu_2 \in [\![P_2]\!]_{G_2}$ and, hence, $\mu = \mu_1 \cup \mu_2 \in [\![(P_1 \text{ AND } P_2)]\!]_{G_2}$.
- Assume that $P = (P_1 \text{ UNION } P_2)$ and that P_1, P_2 are monotone. Let $\mu \in [\![(P_1 \text{ UNION } P_2)]\!]_{G_1}$. Then we have that $\mu \in [\![P_1]\!]_{G_1}$ or $\mu \in [\![P_2]\!]_{G_1}$. Thus, given that P_1 and P_2 are monotone, we conclude that $\mu \in [\![P_1]\!]_{G_2}$ or $\mu \in [\![P_2]\!]_{G_2}$ and, hence, $\mu \in [\![(P_1 \text{ UNION } P_2)]\!]_{G_2}$.
- Assume that $P = (P_1 \text{ FILTER } R)$ and that P_1 is monotone. Let $\mu \in [\![(P_1 \text{ FILTER } R)]\!]_{G_1}$. Then we have that $\mu \in [\![P_1]\!]_{G_1}$ and $\mu \models R$. Thus, given that P_1 is monotone, we conclude that $\mu \in [\![P_1]\!]_{G_2}$ and, hence, $\mu \in [\![(P_1 \text{ FILTER } R)]\!]_{G_2}$.

A.2 Proof of Theorem 4.3

To prove the theorem, we need to introduce some terminology and results from [33]. Let P and P' be SPARQL patterns. P' is said to be a *reduction* of P if P' can be obtained from P by replacing a sub-formula $(P_1 \text{ OPT } P_2)$ of P by P_1. The reflexive and transitive closure of the reduction relation is denoted by $\trianglelefteq$. Moreover, $\text{and}(P)$ is defined to be the pattern obtained from P by replacing every OPT operator in P by an AND operator. Finally, a mapping μ is said to be a partial solution for P over an RDF graph G if $\mu \in [\![\text{and}(P')]\!]_G$, for some $P' \trianglelefteq P$.

Theorem A.1 ([33]) *Let P be a well-designed graph pattern and G an RDF graph. Then for every mapping μ, it holds that $\mu \in [\![P]\!]_G$ iff μ is a maximal (w.r.t. $\preceq$) partial solution for P over G.*

Let P be a well-designed SPARQL pattern. Next we use Theorem A.1 to prove that P is weakly monotone. Assume that G_1, G_2 is a pair of RDF graphs such that $G_1 \subseteq G_2$, and let $\mu \in [\![P]\!]_{G_1}$. We need to show that there exists $\mu' \in [\![P]\!]_{G_2}$ such that $\mu \preceq \mu'$. Given that $\mu \in [\![P]\!]_{G_1}$, we have from Theorem A.1 that there exists $P' \trianglelefteq P$ such that $\mu \in [\![\text{and}(P')]\!]_{G_1}$. Thus, given that $\text{and}(P')$ is a monotone SPARQL pattern (since $\text{and}(P')$ does not mention the OPT operator), we conclude that $\mu \in [\![\text{and}(P')]\!]_{G_2}$ and, hence, μ is a partial solution for P over G_2. If $\mu \in [\![P]\!]_{G_2}$, then for $\mu' = \mu$ it holds that $\mu' \in [\![P]\!]_{G_2}$ and $\mu \preceq \mu'$. If $\mu \notin [\![P]\!]_{G_2}$, then given that μ is a partial solution for P over G_2, we have from Theorem A.1 that there exists $\mu' \in [\![P]\!]_{G_2}$ such that $\mu \preceq \mu'$ and $\mu' \not\preceq \mu$. This concludes the proof of the theorem.