Federation and Navigation in SPARQL 1.1

Marcelo Arenas[1] and Jorge Pérez[2]

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

Abstract. SPARQL is now widely used as the standard query language for RDF. Since the release of its first version in 2008, the W3C group in charge of the standard has been working on extensions of the language to be included in the new version, SPARQL 1.1. These extensions include several interesting and very useful features for querying RDF.

In this paper, we survey two key features of SPARQL 1.1: Federation and navigation capabilities. We first introduce the SPARQL standard presenting its syntax and formal semantics. We then focus on the formalization of federation and navigation in SPARQL 1.1. We analyze some classical theoretical problems such as expressiveness and complexity, and discuss algorithmic properties. Moreover, we present some important recently discovered issues regarding the normative semantics of federation and navigation in SPARQL 1.1, specifically, on the impossibility of answering some unbounded federated queries and the high computational complexity of the evaluation problem for queries including navigation functionalities. Finally, we discuss on possible alternatives to overcome these issues and their implications on the adoption of the standard.

1 Introduction

Jointly with the RDF release in 1998 as a W3C Recommendation, 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. Since then, SPARQL has been rapidly adopted as the standard for querying Semantic Web data. In January 2008, SPARQL became a W3C Recommendation [28]. But this recommendation is not the last step towards the definition of the right language for querying RDF, and the W3C groups involved in the design of the language are currently working on the new version of the standard, the upcoming SPARQL 1.1 [16]. This new version will include several interesting and useful features for querying RDF.

Among the multiple design issues to be considered, there are two important problems that have been in the focus of attention: *Federation* and *Navigation*. Since the release of the first version of SPARQL, the Web has witnessed a constant growth in the amount of RDF data publicly available on-line. Several of these RDF repositories also provide SPARQL interfaces to directly querying their data, which has led the W3C to standardize a set of protocols plus some language constructs to access RDF repositories

T. Eiter and T. Krennwallner (Eds.): Reasoning Web 2012, LNCS 7487, pp. 78–111, 2012.

by means of what are called SPARQL *endpoints*. All these constructs are part of the *federation extensions* of SPARQL [27], planned to be included in the new version of the standard [16]. Somewhat orthogonally, we have the issue of navigating data. 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. Nevertheless, the first release of SPARQL included very limited navigational capabilities. This is one of the motivations of the W3C to include the *property-path* feature in the upcoming version of the SPARQL standard. Property paths are essentially regular expressions, that retrieve pairs of nodes from an RDF graph that are connected by paths conforming to those expressions, and provide a very powerful formalism to query RDF data.

In this paper, intended to be a companion for a short course during the 8th Reasoning Web Summer School, we survey some recent developments regarding federation and navigation in SPARQL 1.1. We first present an introduction to the SPARQL standard presenting its syntax and formal semantics. We then focus on the formalization of the federation and navigation features. We analyze classical theoretical problems such as expressiveness and complexity, and discuss some algorithmic properties.

The formalization of the SPARQL language presented in this paper is based on the official SPARQL 1.1 specification published on January 2012 [16].[1] In this paper, we present some important recently discovered issues regarding this normative semantics, specifically, on the impossibility of answering some unbounded federated queries [7,8] and the high computational complexity of the evaluation problem for queries including navigation functionalities [5,19]. It should be noticed that the semantics of SPARQL 1.1 is currently under discussion, and the standardization group is still receiving input from the community. Hence, some of the issues surveyed in this paper will probably be revisited in the final version of the standard, thus, we also discuss on possible alternatives (some of them currently under discussion) and their implications on the adoption of the standard.

The rest of paper is organized as follows. In Section 2, we introduce the query language SPARQL, and we formalize its syntax and semantics including the SPARQL 1.1 federation extension. In Section 3, we study some boundedness issues associated to this federation extension, and, in particular, we introduce the notions of service-boundedness and service-safeness. In Section 4, we formalize the navigation functionalities of SPARQL 1.1. In Section 5, we present some results on the complexity of evaluating these expressions, and we also present some alternatives to deal with the high complexity of the evaluation problem. Finally, we give some concluding remarks in Section 6.

2 The Query Language SPARQL

In this section, we give an algebraic formalization of SPARQL including the SPARQL 1.1 federation extension. For now, we do not consider other features of SPARQL 1.1 such as property paths [16], but we introduce them later in Section 4. We also consider a set semantics for SPARQL, and extend it to a bag semantics in Section 4.

[1] As of June 2012, this is the last version officially published by the W3C.

We restrict ourselves to SPARQL over simple RDF, that is, we disregard higher entailment regimes (see [14]) such as RDFS or OWL. Our starting point is the existing formalization of SPARQL described in [23], to which we add the operators SERVICE and BINDINGS proposed in [27].[2]

We introduce first the necessary notions about RDF. Assume there are pairwise disjoint infinite sets I, B, and L (IRIs [12], Blank nodes, and Literals, respectively). Then a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an *RDF triple*, where s is called the *subject*, p the *predicate* and o the *object*. An *RDF graph* is a set of RDF triples. Moreover, assume the existence of an infinite set V of variables disjoint from the above sets, and leave UNBOUND to be a reserved symbol that does not belong to any of the previously mentioned sets.

2.1 Syntax

The official syntax of SPARQL [26] considers operators `OPTIONAL`, `UNION`, `FILTER`, `GRAPH`, `SELECT` and concatenation via a point symbol (`.`), to construct graph pattern expressions. Operators `SERVICE` and `BINDINGS` are introduced in the SPARQL 1.1 federation extension, the former for allowing users to direct a portion of a query to a particular SPARQL endpoint, and the latter for transferring results that are used to constrain a query. The syntax of the language also considers `{ }` to group patterns, and some implicit rules of precedence and association. In order to avoid ambiguities in the parsing, we follow the approach proposed in [23], and we first present the syntax of SPARQL graph patterns in a more traditional algebraic formalism, using operators AND (`.`), UNION (`UNION`), OPT (`OPTIONAL`), FILTER (`FILTER`), GRAPH (`GRAPH`) and SERVICE (`SERVICE`), then we introduce the syntax of BINDINGS queries, which use the BINDINGS operator (`BINDINGS`), and we conclude by defining the syntax of SELECT queries, which use the SELECT operator (`SELECT`). More precisely, a SPARQL graph pattern expression is defined recursively as follows:

(1) A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern (a triple pattern).
(2) If P_1 and P_2 are graph patterns, then 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.
(4) If P is a graph pattern and $a \in (I \cup V)$, then $(\text{GRAPH } a\ P)$ is a graph pattern.
(5) If P is a graph pattern and $a \in (I \cup V)$, then $(\text{SERVICE } a\ P)$ is a graph pattern.

As we will see below, despite the similarity between the syntaxes of GRAPH and SERVICE operators, they behave semantically quite differently.

For the exposition of this paper, we leave out further more complex graph patterns from SPARQL 1.1 including aggregates, and subselects (property paths will be formalized in Section 4), but only mention one additional feature which is particularly relevant for federated queries, namely, BINDINGS queries. A SPARQL BINDINGS query is defined as follows:

[2] It is important to notice that SPARQL 1.1 is still under development, and BINDINGS will likely be renamed to VALUES in future specifications.

(6) If P is a graph pattern, $\boldsymbol{W} \in V^n$ is a nonempty sequence of pairwise distinct variables of length $n > 0$ and $\{\boldsymbol{A_1}, \ldots, \boldsymbol{A_k}\}$ is a nonempty set of sequences $\boldsymbol{A_i} \in (I \cup L \cup \{\text{UNBOUND}\})^n$, then $(P \text{ BINDINGS } \boldsymbol{W} \{\boldsymbol{A_1}, \ldots, \boldsymbol{A_k}\})$ is a BINDINGS query.

Finally, a SPARQL SELECT query is defined as:

(7) If P is either a graph pattern or a BINDINGS query, and W is a set of variables, then $(\text{SELECT } W\ P)$ is a SELECT query.

It is important to notice that the rules (1)–(4) above were introduced in [23], while rules (5)–(7) were introduced in [7,8] to formalize the federation extension of SPARQL proposed in [27].

We used the notion of built-in conditions for the FILTER operator above. A SPARQL built-in condition is constructed using elements of the set $(I \cup L \cup V)$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), the binary equality predicate ($=$) as well as unary predicates like bound, isBlank, isIRI, and *isLiteral*.[3] That is: (1) if $?X, ?Y \in V$ and $c \in (I \cup L)$, then $\text{bound}(?X)$, $\text{isBlank}(?X)$, $\text{isIRI}(?X)$, *isLiteral*$(?X)$, $?X = c$ and $?X = ?Y$ are built-in conditions, and (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 either a graph pattern or a BINDINGS query or a SELECT query. In what follows, we use $\text{var}(P)$ to denote 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

To define the semantics of SPARQL queries, we need to introduce some extra terminology from [23]. A mapping μ from V to $(I \cup B \cup L)$ is a partial function $\mu : V \rightarrow (I \cup B \cup L)$. Abusing notation, for a triple pattern t, we denote by $\mu(t)$ the pattern obtained by replacing the variables in t according to μ. The domain of μ, denoted by $\text{dom}(\mu)$, is the subset of V where μ is defined. We sometimes write down concrete mappings in square brackets, for instance, $\mu = [?X \rightarrow a, ?Y \rightarrow b]$ is the mapping with $dom(\mu) = \{?X, ?Y\}$ such that, $\mu(?X) = a$ and $\mu(?Y) = b$. Two mappings μ_1 and μ_2 are compatible, denoted by $\mu_1 \sim \mu_2$, 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 compatible if μ_1 can be extended with μ_2 to obtain a new mapping, and vice versa [23]. We will use the symbol $\mu_\emptyset$ to represent the mapping with empty domain (which is compatible with any other mapping).

Let Ω_1 and Ω_2 be sets of mappings. Then the join of, the union of, the difference between and the left outer-join between Ω_1 and Ω_2 are defined as follows [23]:

[3] For simplicity, we omit here other features such as comparison operators ('$<$', '$>$','$\leq$','$\geq$'), data type conversion and string functions, see [26, Section 11.3] for details. It should be noted that the results of the paper can be easily extended to the other built-in predicates in SPARQL.

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \\
&\qquad\qquad\qquad \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}, \\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}, \\
\Omega_1 \smallsetminus \Omega_2 &= \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2 : \mu \not\sim \mu'\}, \\
\Omega_1 ⟕ \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).
\end{aligned}$$

Next we use these operators to give semantics to graph pattern expressions, BINDINGS queries and SELECT queries. More specifically, we define this semantics in terms of an evaluation function $[\![\cdot]\!]_G$, which takes as input any of these types of queries and returns a set of mappings, depending on the *active dataset* DS and the *active graph* G within DS.

Here, we use the notion of a dataset from SPARQL, i.e. a *dataset*

$$DS = \{(\mathit{def}, G), (g_1, G_1), \ldots, (g_k, G_k)\},$$

with $k \geq 0$ is a set of pairs of symbols and graphs associated with those symbols, where the default graph G is identified by the special symbol $\mathit{def} \notin I$ and the remaining so-called "named" graphs (G_i) are identified by IRIs ($g_i \in I$). Without loss of generality (there are other ways to define the dataset such as via explicit FROM and FROM NAMED clauses), we assume that any query is evaluated over a fixed dataset DS and that any SPARQL endpoint that is identified by an IRI $c \in I$ evaluates its queries against its own dataset $DS_c = \{(\mathit{def}, G_c), (g_{c,1}, G_{c,1}), \ldots, (g_{c,k_c}, G_{c,k_c})\}$. That is, we assume given a partial function ep from the set I of IRIs such that for every $c \in I$, if $\mathit{ep}(c)$ is defined, then $\mathit{ep}(c) = DS_c$ is the dataset associated with the endpoint accessible via IRI c. Moreover, we assume (i) a function $\mathit{graph}(g, DS)$ which – given a dataset $DS = \{(\mathit{def}, G), (g_1, G_1), \ldots, (g_k, G_k)\}$ and a graph name $g \in \{\mathit{def}, g_1, \ldots, g_k\}$ – returns the graph corresponding to symbol g within DS, and (ii) a function $\mathrm{names}(DS)$ which given a dataset DS as before returns the set of names $\{g_1, \ldots, g_k\}$.

The evaluation of a graph pattern P over a dataset DS with active 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 $(P_1$ AND $P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.
(3) If P is $(P_1$ OPT $P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G ⟕ [\![P_2]\!]_G$.
(4) If P is $(P_1$ UNION $P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$.
(5) If P is (GRAPH $c\, P_1$) with $c \in I \cup V$, then

$$[\![P]\!]_G = \begin{cases} [\![P_1]\!]^{DS}_{\mathit{graph}(c,DS)} & \text{if } c \in \mathrm{names}(DS) \\ \{\mu_\emptyset\} & \text{if } c \in I \setminus \mathrm{names}(DS) \\ \{\mu \cup \mu_c \mid \exists g \in \mathrm{names}(DS) : & \\ \qquad \mu_c = [c \to g], & \\ \qquad \mu \in [\![P_1]\!]^{DS}_{\mathit{graph}(g,DS)} \text{ and } \mu_c \sim \mu\} & \text{if } c \in V \end{cases}$$

(6) If P is (SERVICE $c\ P_1$) with $c \in I \cup V$, then

$$[\![P]\!]_G = \begin{cases} [\![P_1]\!]^{ep(c)}_{graph(def, ep(c))} & \text{if } c \in \mathrm{dom}(ep) \\ \{\mu_\emptyset\} & \text{if } c \in I \setminus \mathrm{dom}(ep) \\ \{\mu \cup \mu_c \mid \exists s \in \mathrm{dom}(ep) : & \\ \qquad\qquad \mu_c = [c \to s], & \\ \qquad\qquad \mu \in [\![P_1]\!]^{ep(s)}_{graph(def, ep(s))} \text{ and } \mu_c \sim \mu\} & \text{if } c \in V \end{cases}$$

(7) If P is (P_1 FILTER R), then $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G \mid \mu \models R\}$.

In the previous definition, the semantics of the FILTER operator is based on the definition of the notion of satisfaction of a built-in condition by a mapping. More precisely, given a mapping μ and a built-in condition R, we say that μ satisfies R, denoted by $\mu \models R$, if: [4]

- R is bound($?X$) and $?X \in \mathrm{dom}(\mu)$;
- R is isBlank($?X$), $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) \in B$;
- R is isIRI($?X$), $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) \in I$;
- R is *isLiteral*($?X$), $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) \in L$;
- R is $?X = c$, $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) = c$;
- R is $?X = ?Y$, $?X \in \mathrm{dom}(\mu)$, $?Y \in \mathrm{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$;
- R is $(\neg R_1)$, and it is not the case that $\mu \models R_1$;
- R is $(R_1 \vee R_2)$, and $\mu \models R_1$ or $\mu \models R_2$;
- R is $(R_1 \wedge R_2)$, $\mu \models R_1$ and $\mu \models R_2$.

Moreover, the semantics of BINDINGS queries is defined as follows. Given a sequence $\boldsymbol{W} = [?X_1, \ldots, ?X_n]$ of pairwise distinct variables, where $n \geq 1$, and a sequence $\boldsymbol{A} = [a_1, \ldots, a_n]$ of values from $(I \cup L \cup \{\text{UNBOUND}\})$, let $\mu_{\boldsymbol{W} \mapsto \boldsymbol{A}}$ be a mapping with domain $\{?X_i \mid i \in \{1, \ldots, n\}$ and $a_i \in (I \cup L)\}$ and such that $\mu_{\boldsymbol{W} \mapsto \boldsymbol{A}}(?X_i) = a_i$ for every $?X_i \in \mathrm{dom}(\mu_{\boldsymbol{W} \mapsto \boldsymbol{A}})$. Then

(8) If $P = (P_1$ BINDINGS $\boldsymbol{W}\ \{\boldsymbol{A_1}, \ldots, \boldsymbol{A_k}\})$ is a BINDINGS query:

$$[\![P]\!]_G = [\![P_1]\!]_G \bowtie \{\mu_{\boldsymbol{W} \mapsto \boldsymbol{A_1}}, \ldots, \mu_{\boldsymbol{W} \mapsto \boldsymbol{A_k}}\}.$$

Finally, the semantics of SELECT queries is defined as follows. Given a mapping $\mu : V \to (I \cup B \cup L)$ and a set of variables $W \subseteq V$, the restriction of μ to W, denoted by $\mu_{|_W}$, is a mapping such that $\mathrm{dom}(\mu_{|_W}) = (\mathrm{dom}(\mu) \cap W)$ and $\mu_{|_W}(?X) = \mu(?X)$ for every $?X \in (\mathrm{dom}(\mu) \cap W)$. Then

(9) If $P = $ (SELECT $W\ P_1$) is a SELECT query, then:

$$[\![P]\!]_G = \{\mu_{|_W} \mid \mu \in [\![P_1]\!]_G\}.$$

[4] For the sake of presentation, we use here the two-valued semantics for built-in conditions from [23], instead of the three-valued semantics including errors used in [26]. It should be noticed that the results of the paper can be easily extended to this three-valued semantics.

It is important to notice that the rules (1)–(5), (7) and (9) were introduced in [23], while rules (6) and (8) were proposed in [7,8] to formalize the semantics for the operators SERVICE and BINDINGS introduced in [27]. Intuitively, if $c \in I$ is the IRI of a SPARQL endpoint, then the idea behind the definition of $(\text{SERVICE}\ \ c\, P_1)$ is to evaluate query P_1 in the SPARQL endpoint specified by c. On the other hand, if $c \in I$ is not the IRI of a SPARQL endpoint, then $(\text{SERVICE}\ \ c\, P_1)$ leaves all the variables in P_1 unbound, as this query cannot be evaluated in this case. This idea is formalized by making $\mu_\emptyset$ the only mapping in the evaluation of $(\text{SERVICE}\ \ c\, P_1)$ if $c \notin \text{dom}(ep)$. In the same way, $(\text{SERVICE}\ ?X\ P_1)$ is defined by considering that variable $?X$ is used to store IRIs of SPARQL endpoints. That is, $(\text{SERVICE}\ ?X\ P_1)$ is defined by assigning to $?X$ all the values s in the domain of function ep (in this way, $?X$ is also used to store the IRIs from where the values of the variables in P_1 are coming from). Finally, the idea behind the definition of $(P_1\ \ \text{BINDINGS}\ \boldsymbol{W}\ \{\boldsymbol{A_1}, \ldots, \boldsymbol{A_k}\})$ is to constrain the values of the variables in $\boldsymbol{W}$ to the values specified in $\boldsymbol{A_1}, \ldots, \boldsymbol{A_k}$.

The goal of the rules (6) and (8) is to define in an unambiguous way what the result of evaluating an expression containing the operators SERVICE and BINDINGS should be. As such, these rules should not be considered as a straightforward basis for an implementation of the language. In fact, a direct implementation of the rule (6), that defines the semantics of a pattern of the form $(\text{SERVICE}\ ?X\ P_1)$, would involve evaluating a particular query in every possible SPARQL endpoint, which is obviously infeasible in practice. In the next section, we face this issue and, in particular, we introduce a syntactic condition on SPARQL queries that ensures that a pattern of the form $(\text{SERVICE}\ ?X\ P_1)$ can be evaluated by only considering a finite set of SPARQL endpoints, whose IRIs are actually taken from the RDF graph where the query is being evaluated.

3 Federation

As we pointed out in the previous section, the evaluation of a pattern of the form $(\text{SERVICE}\ ?X\ P)$ is infeasible unless the variable $?X$ is bound to a finite set of IRIs. This notion of *boundedness* is one of the most significant and unclear concepts in the SPARQL federation extension. In fact, since agreement on such a boundedness notion could not yet be found, the current version of the specification of this extension [27] does not specify a formalization of the semantics of queries of the form $(\text{SERVICE}\ ?X\ P)$. Here, we present the formalization of this concept proposed in [7,8], and we study the complexity issues associated with it.

3.1 The Notion of Boundedness

Assume that G is an RDF graph that uses triples of the form $(a, \text{service\_address}, b)$ to indicate that a SPARQL endpoint with name a is located at the IRI b. Moreover, let P be the following SPARQL query:

$$\Bigg(\text{SELECT } \{?X, ?N, ?E\}$$

$$\Bigg(\big((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y\ (?N, \text{email}, ?E))\big)\Bigg)\Bigg).$$

Query P is used to compute the list of names and email addresses that can be retrieved from the SPARQL endpoints stored in an RDF graph. In fact, if $\mu \in [\![P]\!]_G$, then $\mu(?X)$ is the name of a SPARQL endpoint stored in G, $\mu(?N)$ is the name of a person stored in that SPARQL endpoint and $\mu(?E)$ is the email address of that person. It is important to notice that there is a simple strategy that ensures that query P can be evaluated in practice: first compute $[\![(?X, \text{service\_address}, ?Y)]\!]_G$, and then for every μ in this set, compute $[\![(\text{SERVICE}\ \ a\ \ (?N, \text{email}, ?E))]\!]_G$ with $a = \mu(?Y)$. More generally, SPARQL pattern $(\text{SERVICE } ?Y\ (?N, \text{email}, ?E))$ can be evaluated over DS in this case as only a finite set of values from the domain of G need to be considered as the possible values of $?Y$. This idea naturally gives rise to the following notion of boundedness for the variables of a SPARQL query. In the definition of this notion, $\text{dom}(G)$ refers to the domain of a graph G, that is, the set of elements from $(I \cup B \cup L)$ that are mentioned in G; $\text{dom}(DS)$ refers to the union of the domains of all graphs in the dataset DS; and finally, $\text{dom}(P)$ refers to the set of elements from $(I \cup L)$ that are mentioned in P.

Definition 1 (Boundedness [7,8]). *Let P be a* SPARQL *query and $?X \in \text{var}(P)$. Then $?X$ is bound in P if one of the following conditions holds:*

- *P is either a graph pattern or a BINDINGS query, and for every dataset DS, every RDF graph G in DS and every $\mu \in [\![P]\!]_G$: $?X \in \text{dom}(\mu)$ and $\mu(?X) \in (\text{dom}(DS) \cup \text{names}(DS) \cup \text{dom}(P))$.*
- *P is a SELECT query* $(\text{SELECT } W\ P_1)$ *and $?X$ is bound in P_1.*

In the evaluation of a graph pattern $(\text{GRAPH } ?X\ P)$ over a dataset DS, variable $?X$ necessarily takes a value from $\text{names}(DS)$. Thus, the GRAPH operator makes such a variable $?X$ to be bound. Given that the values in $\text{names}(DS)$ are not necessarily mentioned in the dataset DS, the previous definition first imposes the condition that $?X \in \text{dom}(\mu)$, and then not only considers the case $\mu(?X) \in \text{dom}(DS)$ but also the case $\mu(?X) \in \text{names}(DS)$. In the same way, the BINDINGS operator can make a variable $?X$ in a query P to be bound by assigning to it a fixed set of values. Given that these values are not necessarily mentioned in the dataset DS where P is being evaluated, the previous definition also considers the case $\mu(?X) \in \text{dom}(P)$. As an example of the above definition, we note that variable $?Y$ is bound in the graph pattern

$$P_1 = ((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y\ (?N, \text{email}, ?E))),$$

as for every dataset DS, every RDF graph G in DS and every mapping $\mu \in [\![P_1]\!]_G$, we know that $?Y \in \text{dom}(\mu)$ and $\mu(?Y) \in \text{dom}(DS)$. Moreover, we also have that variable $?Y$ is bound in $(\text{SELECT } \{?X, ?N, ?E\}\ P_1)$ as $?Y$ is bound in graph pattern P_1.

A natural way to ensure that a SPARQL query P can be evaluated in practice is by imposing the restriction that for every sub-pattern $(\text{SERVICE } ?X\ P_1)$ of P, it holds that $?X$ is bound in P. However, the following theorem shows that such a condition is undecidable and, thus, a SPARQL query engine would not be able to check it in order to ensure that a query can be evaluated.

Theorem 1 ([7,8]). *The problem of verifying, given a* SPARQL *query P and a variable $?X \in \mathrm{var}(P)$, whether $?X$ is bound in P is undecidable.*

The fact that the notion of boundedness is undecidable prevents one from using it as a restriction over the variables in SPARQL queries. To overcome this limitation, it was introduced in [7,8] a syntactic condition that ensures that a variable is bound in a pattern and that can be efficiently verified.

Definition 2 (Strong boundedness [7,8]). *Let P be a* SPARQL *query. Then the set of strongly bound variables in P, denoted by* $\mathrm{SB}(P)$*, is recursively defined as follows:*

- *if $P = t$, where t is a triple pattern, then* $\mathrm{SB}(P) = \mathrm{var}(t)$*;*
- *if $P = (P_1 \text{ AND } P_2)$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$*;*
- *if $P = (P_1 \text{ UNION } P_2)$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$*;*
- *if $P = (P_1 \text{ OPT } P_2)$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1)$*;*
- *if $P = (P_1 \text{ FILTER } R)$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1)$*;*
- *if $P = (\text{GRAPH } c\ P_1)$, with $c \in I \cup V$, then*

$$\mathrm{SB}(P) = \begin{cases} \emptyset & c \in I, \\ \mathrm{SB}(P_1) \cup \{c\} & c \in V; \end{cases}$$

- *if $P = (\text{SERVICE } c\ P_1)$, with $c \in I \cup V$, then* $\mathrm{SB}(P) = \emptyset$*;*
- *if $P = (P_1 \text{ BINDINGS } \boldsymbol{W}\ \{\boldsymbol{A_1}, \ldots, \boldsymbol{A_n}\})$, then*

$$\begin{aligned} \mathrm{SB}(P) \;=\; & \mathrm{SB}(P_1) \cup \\ & \{?X \mid ?X \textit{ is included in } \boldsymbol{W} \textit{ and for} \\ & \ \textit{every } i \in \{1, \ldots, n\} : ?X \in \mathrm{dom}(\mu_{\boldsymbol{W} \mapsto \boldsymbol{A_i}})\}; \end{aligned}$$

- *if $P = (\text{SELECT } W\ P_1)$, then* $\mathrm{SB}(P) = (W \cap \mathrm{SB}(P_1))$.

The previous definition recursively collects from a SPARQL query P a set of variables that are guaranteed to be bound in P. For example, if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$ as one knows that for every variable $?X \in \mathrm{var}(t)$, every dataset DS and every RDF graph G in DS, if $\mu \in [\![t]\!]_G$, then $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) \in \mathrm{dom}(G)$ (which is a subset of $\mathrm{dom}(DS)$). In the same way, if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$ as one knows that if $?X$ is bound in P_1 or in P_2, then $?X$ is bound in P. As a final example, notice that if $P = (P_1 \text{ BINDINGS } \boldsymbol{W}\ \{\boldsymbol{A_1}, \ldots, \boldsymbol{A_n}\})$ and $?X$ is a variable mentioned in $\boldsymbol{W}$ such that $?X \in \mathrm{dom}(\mu_{\boldsymbol{W} \mapsto \boldsymbol{A_i}})$ for every $i \in \{1, \ldots, n\}$, then $?X \in \mathrm{SB}(P)$. In this case, one knows that $?X$ is bound in P since $[\![P]\!]_G = [\![P_1]\!]_G \bowtie \{\mu_{\boldsymbol{W} \mapsto \boldsymbol{A_1}}, \ldots, \mu_{\boldsymbol{W} \mapsto \boldsymbol{A_n}}\}$ and $?X$ is in the domain of each one of the mappings $\mu_{\boldsymbol{W} \mapsto \boldsymbol{A_i}}$, which implies that $\mu(?X) \in \mathrm{dom}(P)$ for every $\mu \in [\![P]\!]_G$. In the following proposition, it is formally shown that our intuition about $\mathrm{SB}(P)$ is correct, in the sense that every variable in this set is bound in P.

Proposition 1 ([7,8]). *For every* SPARQL *query* P *and variable* $?X \in \mathrm{var}(P)$*, if* $?X \in \mathrm{SB}(P)$*, then* $?X$ *is bound in* P*.*

Given a SPARQL query P and a variable $?X \in \mathrm{var}(P)$, it can be efficiently verified whether $?X$ is strongly bound in P. Thus, a natural and efficiently verifiable way to ensure that a SPARQL query P can be evaluated in practice is by imposing the restriction that for every sub-pattern $(\mathrm{SERVICE}\ ?X\ P_1)$ of P, it holds that $?X$ is strongly bound in P. However, this notion still needs to be modified in order to be useful in practice, as shown by the following examples.

Example 1. Assume first that P_1 is the following graph pattern:

$$P_1 = \Bigg[(?X, \text{service\_description}, ?Z) \text{ UNION} \\ \Bigg((?X, \text{service\_address}, ?Y) \text{ AND} \\ (\text{SERVICE } ?Y\ (?N, \text{email}, ?E))\Bigg)\Bigg].$$

That is, either $?X$ and $?Z$ store the name of a SPARQL endpoint and a description of its functionalities, or $?X$ and $?Y$ store the name of a SPARQL endpoint and the IRI where it is located (together with a list of names and email addresses retrieved from that location). Variable $?Y$ is neither bound nor strongly bound in P_1. However, there is a simple strategy that ensures that P_1 can be evaluated over a dataset DS and an RDF graph G in DS: first compute $[\![(?X, \text{service\_description}, ?Z)]\!]_G$, then compute $[\![(?X, \text{service\_address}, ?Y)]\!]_G$, and finally for every μ in the set of mappings $[\![(?X, \text{service\_address}, ?Y)]\!]_G$, compute $[\![(\text{SERVICE } a\ (?N, \text{email}, ?E))]\!]_G$ with $a = \mu(?Y)$. In fact, the reason why P_1 can be evaluated in this case is that $?Y$ is bound (and strongly bound) in the following sub-pattern of P_1:

$$((?X, \text{service\_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y\ (?N, \text{email}, ?E))).$$

As a second example, assume that DS is a dataset and G is an RDF graph in DS that uses triples of the form $(a_1, \text{related\_with}, a_2)$ to indicate that the SPARQL endpoints located at the IRIs a_1 and a_2 store related data. Moreover, assume that P_2 is the following graph pattern:

$$P_2 = \Bigg[(?U_1, \text{related\_with}, ?U_2) \text{ AND} \\ \Bigg(\text{SERVICE } ?U_1 \Bigg((?N, \text{email}, ?E) \text{ OPT} \\ (\text{SERVICE } ?U_2\ (?N, \text{phone}, ?F))\Bigg)\Bigg)\Bigg].$$

When this query is evaluated over the dataset DS and the RDF graph G in DS, it returns for every tuple $(a_1, \text{related\_with}, a_2)$ in G, the list of names and email addresses that that can be retrieved from the SPARQL endpoint located at a_1, together with the phone number for each person in this list for which this data can be retrieved from the SPARQL endpoint located at a_2 (recall that pattern (SERVICE $?U_2$ $(?N, \text{phone}, ?F)$) is nested inside the first SERVICE operator in P_2). To evaluate this query over an RDF graph, first it is necessary to determine the possible values for variable $?U_1$, and then to submit the query $((?N, \text{email}, ?E)$ OPT (SERVICE $?U_2$ $(?N, \text{phone}, ?F)$)) to each one of the endpoints located at the IRIs stored in $?U_1$. In this case, variable $?U_2$ is bound (and also strongly bound) in P_2. However, this variable is not bound in the graph pattern $((?N, \text{email}, ?E)$ OPT (SERVICE $?U_2$ $(?N, \text{phone}, ?F)$)), which has to be evaluated in some of the SPARQL endpoints stored in the RDF graph where P_2 is being evaluated, something that is infeasible in practice. It is important to notice that the difficulties in evaluating P_2 are caused by the nesting of SERVICE operators (more precisely, by the fact that P_2 has a sub-pattern of the form (SERVICE $?X_1$ Q_1), where Q_1 has in turn a sub-pattern of the form (SERVICE $?X_2$ Q_2) such that $?X_2$ is bound in P_2 but not in Q_1). □

In the following section, the concept of strongly boundedness is used to define a notion that ensures that a SPARQL query containing the SERVICE operator can be evaluated in practice, and which takes into consideration the ideas presented in the above examples.

3.2 The Notion of Service-Safeness: Considering Sub-Patterns and Nested SERVICE Operators

The goal of this section is to provide a condition that ensures that a SPARQL query containing the SERVICE operator can be safely evaluated in practice. To this end, we first need to introduce some terminology. Given a SPARQL query P, define $\mathcal{T}(P)$ as the *parse* tree of P. In this tree, every node corresponds to a sub-pattern of P. An example of a parse tree of a pattern Q is shown in Figure 1. In this figure, $u_1, u_2, u_3, u_4, u_5, u_6$ are the identifiers of the nodes of the tree, which are labeled with the sub-patterns of Q. It is important to notice that in this tree we do not make any distinction between the

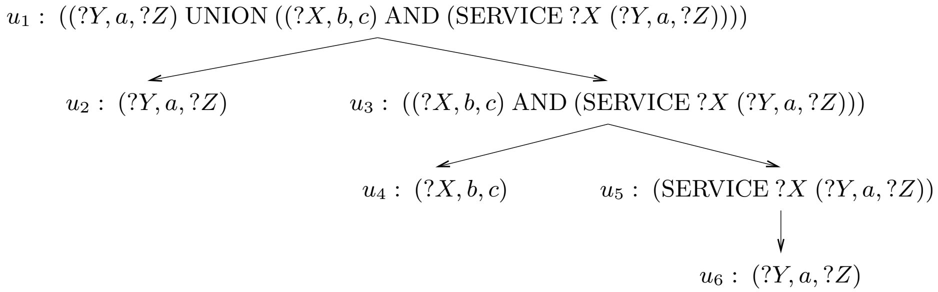
u_1 : $((?Y, a, ?Z)$ UNION $((?X, b, c)$ AND (SERVICE $?X$ $(?Y, a, ?Z)$)))
u_2 : $(?Y, a, ?Z)$
u_3 : $((?X, b, c)$ AND (SERVICE $?X$ $(?Y, a, ?Z)$))
u_4 : $(?X, b, c)$
u_5 : (SERVICE $?X$ $(?Y, a, ?Z)$)
u_6 : $(?Y, a, ?Z)$

Fig. 1. Parse tree $\mathcal{T}(Q)$ for the graph pattern $Q = ((?Y, a, ?Z)$ UNION $((?X, b, c)$ AND (SERVICE $?X$ $(?Y, a, ?Z)$)))

different operators in SPARQL, we just use the child relation to store the structure of the sub-patterns of a SPARQL query.

Tree $\mathcal{T}(P)$ is used to define the notion of service-boundedness, which extends the concept of boundedness, introduced in the previous section, to consider variables that are bound inside sub-patterns and nested SERVICE operators. It should be noticed that these two features were identified in the previous section as important for the definition of a notion of boundedness (see Example 1).

Definition 3 (Service-boundedness [7,8]). *A* SPARQL *query P is service-bound if for every node u of $\mathcal{T}(P)$ with label* (SERVICE $?X$ P_1)*, it holds that:*

(1) there exists a node v of $\mathcal{T}(P)$ with label P_2 such that v is an ancestor of u in $\mathcal{T}(P)$ and $?X$ is bound in P_2;
(2) P_1 is service-bound.

For example, query Q in Figure 1 is service-bound. In fact, condition (1) of Definition 3 is satisfied as u_5 is the only node in $\mathcal{T}(Q)$ having as label a SERVICE graph pattern, in this case (SERVICE $?X$ $(?Y, a, ?Z)$), and for the node u_3, it holds that: u_3 is an ancestor of u_5 in $\mathcal{T}(P)$, the label of u_3 is $P = ((?X, b, c)$ AND (SERVICE $?X$ $(?Y, a, ?Z)))$ and $?X$ is bound in P. Moreover, condition (2) of Definition 3 is satisfied as the sub-pattern $(?Y, a, ?Z)$ of the label of u_5 is also service-bound.

The notion of service-boundedness captures our intuition about the condition that a SPARQL query containing the SERVICE operator should satisfy. Unfortunately, the following theorem shows that such a condition is undecidable and, thus, a SPARQL query engine would not be able to check it in order to ensure that a query can be evaluated.

Theorem 2 ([7,8]). *The problem of verifying, given a* SPARQL *query P, whether P is service-bound is undecidable.*

As for the case of the notion of boundedness, the fact that the notion of service-boundedness is undecidable prevents one from using it as a restriction over the variables used in SERVICE calls. To overcome this limitation, in the definition of service-boundedness, the restriction that the variables used in SERVICE calls are bound is replaced by the decidable restriction that they are strongly bound. In this way, one obtains a syntactic condition over SPARQL queries that ensures that they are service-bound, and which can be efficiently verified.

Definition 4 (Service-safeness [7,8]). *A* SPARQL *query P is service-safe if for every node u of $\mathcal{T}(P)$ with label* (SERVICE $?X$ P_1)*, it holds that:*

(1) there exists a node v of $\mathcal{T}(P)$ with label P_2 such that v is an ancestor of u in $\mathcal{T}(P)$ and $?X \in \mathrm{SB}(P_2)$;
(2) P_1 is service-safe.

As a corollary of Proposition 1, one obtains the following proposition.

Proposition 2 ([7,8]). *If a* SPARQL *query P is service-safe, then P is service-bound.*

The notion of service-safeness is used in the system presented in [7,8] to verify that a SPARQL query can be evaluated in practice. More precisely, that system uses a bottom-up algorithm over the parse tree $\mathcal{T}(Q)$ of a SPARQL query Q for validating the service-safeness condition. This procedure traverses the parse tree $\mathcal{T}(Q)$ twice for ensuring that Q can be correctly evaluated. In the first traversal, for each node identifier u of $\mathcal{T}(Q)$, the algorithm computes the set of strongly bound variables for the label P of u. For example, in the parse tree shown in Figure 1, the variable $?X$ is identified as the only strongly bound variable for the label of the node with identifier u_3. In the second traversal, the bottom-up algorithm uses these sets of strongly bound variables to check two conditions for every node identifier u of $\mathcal{T}(Q)$ with label of the form (SERVICE $?X$ P): whether there exists a node v of $\mathcal{T}(Q)$ with label P' such that v is an ancestor of u in $\mathcal{T}(Q)$ and $?X$ is strongly bound in P', and whether P is itself service-safe. If these two conditions are fulfilled, then the algorithm returns *true* to indicate that Q is service-safe. Otherwise, the procedure returns *no*.

4 Navigation

Navigational features have been largely recognized as fundamental for graph database query languages. This fact has motivated several authors to propose RDF query languages with navigational capabilities [22,1,18,4,2], and, in fact, it was the motivation to include the *property-path* feature in the upcoming version of the SPARQL standard, SPARQL 1.1 [16]. Property paths are essentially regular expressions, that are used to retrieve pairs of nodes from an RDF graph if they are connected by paths conforming to those expressions. In the following two sections, we introduce the syntax of property paths, and some of the main results on the semantics and complexity of these expressions.

We focus on the semantics for property paths presented in the last published version of the specification [16] (January 2012), and that has been considered as the semantics for property paths since early stages of the standardization process (first introduced in October 2010). As a disclaimer, it should be noticed that recently (beginning of 2012), the normative semantics of property paths is being heavily discussed in the W3C mailing lists [38], and, thus, this semantics will probably change in the future. This discussion was initiated by two recently published articles on this subject [5,19], which show some efficiency problems in the original design of property paths. Thus, although this design may change, from a pedagogical point of view, as well as from a historical point of view, it is important and interesting to present the semantics that have lasted for more than a year as the official semantics of property paths, the rationale behind its definition and its main features, and the issues that may lead to its replacement in the near future.

The normative semantics of property paths in SPARQL 1.1 poses several interesting research issues. Although property paths are syntactically nothing else but classical regular expressions, SPARQL 1.1 defines a *bag* (or *multiset*) semantics for these expressions. That is, when evaluating property-path expressions, one can obtain several duplicates for the same solution, essentially one duplicate for every different path in the graph satisfying the expression. Since RDF graphs containing cycles may lead to an infinite number of paths satisfying a particular expression, the official specification defines the semantics of property paths by means of a particular counting procedure, which handles cycles in a way that ensures that the final count is finite. In this section, we consider the formalization of this procedure that was presented in [5].

In order to formally introduce the semantics of property paths, we first formalize the bag semantics of SPARQL operators in Section 4.1. Then based on this formalization, we introduce in Section 4.2, the semantics of property paths in SPARQL 1.1.

4.1 Bag Semantics for SPARQL 1.1

In this section, we introduce a bag (or multiset) semantics for SPARQL, that is, the evaluation of a query is defined as a set of mappings in which every element μ is annotated with a positive integer that represents the cardinality of μ in the bag. As we will see, cardinality of solutions is a crucial part of the normative semantics of property paths in SPARQL 1.1 [16].

Formally, we represent a bag of mappings as a pair $(\Omega, \mathrm{card}_\Omega)$, where Ω is a set of mappings and card_Ω is a function such that $\mathrm{card}_\Omega(\mu)$ is the cardinality of μ in Ω (we assume that $\mathrm{card}_\Omega(\mu) > 0$ for every $\mu \in \Omega$, and $\mathrm{card}_\Omega(\mu') = 0$ for every $\mu' \notin \Omega$). With this notion, we have the necessary ingredients to define the semantics of SPARQL 1.1 queries. For the sake of readability, we repeat here the definitions presented in Section 2.2 but now including the computation of cardinalities. Since our main focus in this section are the navigational features in SPARQL 1.1, we do not consider GRAPH and SERVICE operators. Thus, we focus on points (1), (2), (3), (4), and (7) presented in Section 2.2. More precisely, let G be an RDF graph and P a graph pattern:

(1) If P is a triple pattern t, then $[\![P]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{var}(t) \text{ and } \mu(t) \in G\}$. Moreover, for every $\mu \in [\![P]\!]_G$, it holds that $\mathrm{card}_{[\![P]\!]_G}(\mu) = 1$.

(2) If P is $(P_1 \text{ AND } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$. Moreover, for every $\mu \in [\![P]\!]_G$ we have that $\mathrm{card}_{[\![P]\!]_G}(\mu)$ is given by the expression:

$$\sum_{\mu_1 \in [\![P_1]\!]_G} \left[\sum_{\mu_2 \in [\![P_2]\!]_G \,:\, \mu = \mu_1 \cup \mu_2} \Big(\mathrm{card}_{[\![P_1]\!]_G}(\mu_1) \cdot \mathrm{card}_{[\![P_2]\!]_G}(\mu_2) \Big) \right].$$

(3) If P is $(P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G ⟕ [\![P_2]\!]_G$. Moreover, for every $\mu \in [\![P]\!]_G$, if $\mu \in [\![(P_1 \text{ AND } P_2)]\!]_G$, then $\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![(P_1 \text{ AND } P_2)]\!]_G}(\mu)$, and if $\mu \notin [\![(P_1 \text{ AND } P_2)]\!]_G$, then $\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![P_1]\!]_G}(\mu)$.

(4) If P is $(P_1 \text{ UNION } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$. Moreover, for every $\mu \in [\![P]\!]_G$, it holds that $\mathrm{card}_{[\![P]\!]_G}(\mu) = \mathrm{card}_{[\![P_1]\!]_G}(\mu) + \mathrm{card}_{[\![P_2]\!]_G}(\mu)$.

The evaluation of a SPARQL 1.1 query Q over an RDF graph G, denoted by $[\![Q]\!]_G$, is defined as follows. If Q is a SPARQL 1.1 query (SELECT W P), then $[\![Q]\!]_G = \{\mu_{|W} \mid \mu \in [\![P]\!]_G\}$ and for every $\mu \in [\![Q]\!]_G$:

$$\mathrm{card}_{[\![Q]\!]_G}(\mu) = \sum_{\mu' \in [\![P]\!]_G \,:\, \mu = \mu'_{|W}} \mathrm{card}_{[\![P]\!]_G}(\mu').$$

If Q is a SPARQL 1.1 query (SELECT * P), then $[\![Q]\!]_G = [\![P]\!]_G$ and $\text{card}_{[\![Q]\!]_G}(\mu) = \text{card}_{[\![P]\!]_G}(\mu)$ for every $\mu \in [\![Q]\!]_G$. If Q is a SPARQL 1.1 query (SELECT DISTINCT W P), then $[\![Q]\!]_G = \{\mu_{|W} \mid \mu \in [\![P]\!]_G\}$ and for every $\mu \in [\![Q]\!]_G$, we have that $\text{card}_{[\![Q]\!]_G}(\mu) = 1$. Finally, if Q is a SPARQL 1.1 query (SELECT DISTINCT * P), then $[\![Q]\!]_G = [\![P]\!]_G$ and for every $\mu \in [\![Q]\!]_G$, we have that $\text{card}_{[\![Q]\!]_G}(\mu) = 1$.

To conclude the definition of the semantics of SPARQL 1.1, we need to define the semantics of filter expressions. Given an RDF graph G and a graph pattern expression $P = (P_1 \text{ FILTER } R)$, we have that $[\![P]\!]_G = \{\mu \in [\![P_1]\!]_G \mid \mu \models R\}$, and for every $\mu \in [\![P]\!]_G$, we have that $\text{card}_{[\![P]\!]_G}(\mu) = \text{card}_{[\![P_1]\!]_G}(\mu)$.

4.2 Syntax and Semantics of SPARQL 1.1 Property Paths

In this section, we use the framework presented in the previous section to formalize the semantics of property paths. According to [16], a property paths is recursively defined as follows: (1) if $a \in I$, then a is a property path, and (2) if p_1 and p_2 are property paths, then $p_1|p_2$, p_1/p_2 and p_1^* are property paths. Thus, from a syntactical point of view, property paths are regular expressions over the vocabulary I, being | disjunction, / concatenation and $(\ )^*$ the Kleene star. It should be noticed that the definition of property paths in [16] includes some additional features that are common in regular expressions, such as $p?$ (zero or one occurrences of p) and p^+ (one or more occurrences of p). In this paper, we focus on the core operators |, / and $(\ )^*$, as they suffice to prove the infeasibility of the evaluation of property paths in SPARQL 1.1.

A property-path triple is a tuple t of the form (u, p, v), where $u, v \in (I \cup V)$ and p is a property path. SPARQL 1.1 includes as atomic formulas triple patterns and property-path triples. Thus, to complete the definition of the semantics of SPARQL 1.1, we need to specify how property-path triples are evaluated over RDF graphs, that is, we need to extend the definition of the function $[\![\cdot]\!]_G$ to include property-path triples.

To define the semantics of property-path triples, we follow closely the standard specification [16]. Assume that $u, v \in (I \cup V)$, $W = (\{u, v\} \cap V)$ and p is a property path. Notice that if $u, v \in I$, then $W = \emptyset$. Then the evaluation of property-path triple $t = (u, p, v)$ over an RDF graph G, denoted by $[\![t]\!]_G$, is defined recursively as follows. If $p = a$, where $a \in I$, then (u, p, v) is a triple pattern and $[\![t]\!]_G$ and $\text{card}_{[\![t]\!]_G}(\cdot)$ are defined as in Section 4.1. Otherwise, we have that either $p = p_1|p_2$ or $p = p_1/p_2$ or $p = p_1^*$, where p_1, p_2 are property paths, and $[\![t]\!]_G$ is defined as follows. First, if $p = p_1|p_2$, then $[\![t]\!]_G$ is defined in [16] as the result of evaluating the pattern $((u, p_1, v) \text{ UNION } (u, p_2, v))$ over G. Thus, we have that:

$$[\![t]\!]_G = \{\mu \mid \mu \in [\![(u, p_1, v)]\!]_G \text{ or } \mu \in [\![(u, p_2, v)]\!]_G\},$$

and for every $\mu \in [\![t]\!]_G$, we have that:

$$\text{card}_{[\![t]\!]_G}(\mu) = \text{card}_{[\![(u,p_1,v)]\!]_G}(\mu) + \text{card}_{[\![(u,p_2,v)]\!]_G}(\mu).$$

Second, if $p = p_1/p_2$, then assuming that $?X$ is a variable such that $?X \notin W$, we have that $[\![t]\!]_G$ is defined in [16] as the result of first evaluating the pattern $((u, p_1, ?X)$ AND

$(?X, p_2, v))$ over G, and then projecting over the variables of property-path triple t (and, thus, projecting out the variable $?X$). Thus, we have that:

$$[\![t]\!]_G \;=\; \{(\mu_1 \cup \mu_2)_{|W} \mid \mu_1 \in [\![(u, p_1, ?X)]\!]_G, \mu_2 \in [\![(?X, p_2, v)]\!]_G \text{ and } \mu_1 \sim \mu_2\},$$

and for every $\mu \in [\![t]\!]_G$, we have that:

$$\mathrm{card}_{[\![t]\!]_G}(\mu) \;=\; \sum_{\mu_1 \in [\![(u,p_1,?X)]\!]_G} \left[\sum_{\substack{\mu_2 \in [\![(?X,p_2,v)]\!]_G \\ \mu = (\mu_1 \cup \mu_2)_{|W}}} \left(\mathrm{card}_{[\![(u,p_1,?X)]\!]_G}(\mu_1) \cdot \mathrm{card}_{[\![(?X,p_2,v)]\!]_G}(\mu_2) \right) \right].$$

Finally, if $p = p_1^*$, then $[\![t]\!]_G$ is defined in [16] in terms of the procedures COUNT and ALP shown in Figure 2. More precisely,

$$[\![t]\!]_G \;=\; \{\mu \mid \mathrm{dom}(\mu) = W \text{ and } \textsc{Count}(\mu(u), p_1, \mu(v), G) > 0\}.$$

Moreover, for every $\mu \in [\![t]\!]_G$, it holds that

$$\mathrm{card}_{[\![t]\!]_G}(\mu) = \textsc{Count}(\mu(u), p_1, \mu(v), G).$$

Procedure ALP in Figure 2 is taken from [16]. It is important to notice that lines 5 and 6 in ALP formalize, in our terminology, the use of a procedure call `eval` in the

```
Function COUNT($a$, path, $b$, $G$)
Input: $a, b \in I$, path is a property path and $G$ is an RDF graph.
 1: Result := empty list
 2: Visited := empty set
 3: ALP($a$, path, Result, Visited, $G$)
 4: $n$ := number of occurrences of $b$ in Result
 5: return $n$
```

```
Procedure ALP($a$, path, Result, Visited, $G$)
Input: $a \in I$, path is a property path, Result is a list of elements from $I$, Visited is a set of
elements from $I$ and $G$ is an RDF graph.
 1: if $a \in$ Visited then
 2:     return
 3: end if
 4: add $a$ to Visited, and add $a$ to Result
 5: $\Omega := [\![(a, path, ?X)]\!]_G$
 6: let Next be the list of elements $b = \mu(?X)$ for $\mu \in \Omega$, such that the number of occurrences
    of $b$ in Next is $\mathrm{card}_\Omega(\mu)$
 7: for each $c \in$ Next do
 8:     ALP($c$, path, Result, Visited, $G$)
 9: end for
10: remove $a$ from Visited
```

Fig. 2. Procedures used in the evaluation of property-path triples of the form $(u, path^*, v)$

definition of ALP in [16]. According to [16], procedure ALP has to be used as follows to compute $\mathrm{card}_{[\![t]\!]_G}(\mu)$, where $t = (u, p_1^*, v)$. Assuming that *Result* is the empty list and *Visited* is the empty set, first one has to invoke ALP$(\mu(u)$, p, *Result*, *Visited*, $G)$, then one has to check whether $\mu(v)$ appears in the resulting list *Result*, and if this is the case then $\mathrm{card}_{[\![t]\!]_G}(\mu)$ is set as the number of occurrences of $\mu(v)$ in the list *Result*. For the sake of readability, we have encapsulated in the auxiliary procedure COUNT these steps to compute $\mathrm{card}_{[\![t]\!]_G}(\mu)$ from procedure ALP, and we have defined $[\![t]\!]_G$ by using COUNT, thus formalizing the semantics proposed by the W3C in [16].

The idea behind algorithm ALP is to incrementally construct paths that conform to a property path of the form p_1^*, that is, to construct sequences of nodes a_1, a_2, ..., a_n from an RDF graph G such that each node a_{i+1} is reachable from a_i in G by following the path p_1, but with the important feature (implemented through the use of the set *Visited*) that each node a_i is distinct from all the previous nodes a_j selected in the sequence (thus avoiding cycles in the sequence a_1, a_2, ..., a_n).

5 The Complexity of Evaluating Property-Path SPARQL Queries

In this section, we show some of the results presented in [5,19] on the complexity of evaluating property paths according to the semantics proposed by the W3C, as well as several other alternative semantics. We first present in Section 5.1 an experimental study on the impact of counting property paths. As observed in [5,19], current implementations of SPARQL 1.1 show a strikingly poor performance when evaluating even the most simple queries. Then in Section 5.2, the computational complexity of the evaluation problem for property paths is studied, providing a formal explanation on the poor performance of the implementations. Later in Section 5.3, we present some alternative semantics for property paths based on more classical ways of navigating graph databases. Finally, in Section 5.4, we show some results stating that when repeated solutions are not considered, then one can obtain efficient evaluation methods for property paths.

5.1 Experimental Evaluation

This section is based on the experimental study performed in [5]. The idea is to provide the reader with a sense of the practical impact in query evaluation of using property paths, by comparing the performance of several important implementations of SPARQL.

The SPARQL 1.1 engines considered in the evaluation are the following [5]:

- **ARQ** – version 2.8.8, 21 April 2011 [33]: ARQ is a java implementation of SPARQL for the Jena Semantic Web Framework [10].
- **RDF::Query** – version 2.907, 1 June 2011 [35]: RDF::Query is a perl module that implements SPARQL 1.1.
- **KGRAM** – version 3.0, September 2011 [34]: KGRAM is an implementation of an abstract machine that unifies graph match and SPARQL 1.1 [11]. The engine is accessed via the Corese (COnceptual REsource Search Engine) libraries implemented in java.

```
@prefix : <http://example.org/> .
:a0 :p :a1, :a2, :a3, :a4, :a5, :a6, :a7 .
:a1 :p :a0, :a2, :a3, :a4, :a5, :a6, :a7 .
:a2 :p :a0, :a1, :a3, :a4, :a5, :a6, :a7 .
:a3 :p :a0, :a1, :a2, :a4, :a5, :a6, :a7 .
:a4 :p :a0, :a1, :a2, :a3, :a5, :a6, :a7 .
:a5 :p :a0, :a1, :a2, :a3, :a4, :a6, :a7 .
:a6 :p :a0, :a1, :a2, :a3, :a4, :a5, :a7 .
:a7 :p :a0, :a1, :a2, :a3, :a4, :a5, :a6 .
```

Fig. 3. RDF graph representing a clique with 8 nodes

- **Sesame** – version 2.5.1, 23 September 2011 [36]: Sesame is a framework for processing RDF data implemented in java, and provides a set of libraries to access data and execute SPARQL 1.1 queries.

The tests were run in a dedicated machine with the following configuration: Debian 6.0.2 Operating System, Kernel 2.6.32, CPU Intel Xeon X3220 Quadcore with 2.40GHz, and 4GB PC2-5300 RAM. All tests were run considering main memory storage. This should not be considered as a problem since the maximum size of the input RDF graphs that we used was only 25.8 KB. A timeout of 60 minutes was considered. For each test, the number reported is the average of the results obtained by executing the test (at least) 4 times. No experiment showed a significant standard deviation [5].

The Clique Experiment. The first experiment reported in [5] considered cliques (complete graphs) of different sizes, from a clique with 2 nodes (containing 2 triples) to a clique with 13 nodes (156 triples). For example, a clique with 8 nodes in N3 notation is shown in Figure 3. The first query to be tested, is the following query

Cliq-1: `SELECT * WHERE { :a0 (:p)* :a1 }`.

This query essentially tests if the nodes `:a0` and `:a1` are connected. Since this query has no variables, the solution is an *empty tuple*, which, for example, in ARQ is represented by the string `|   |`, and in Sesame by the string `[]` (when the query solution is printed to the standard output). RDF::Query does not support queries without variables, thus for this implementation the following query was tested:

CliqF-1: `SELECT * WHERE { :a0 (:p)* ?x  FILTER (?x = :a1) }`.

Table 1 shows the result obtained for this experiment in terms of the time (in seconds) and the number of solutions produced as output, when the input is a clique with n nodes. The symbol "–" in the table means *timeout* of one hour. Figure 4 shows a plot of the same data.

The impact of using *nested stars* was also tested [5]. In particular, the following queries were tested:

Cliq-2: `SELECT * WHERE { :a0 ((:p)*)* :a1 }`
Cliq-3: `SELECT * WHERE { :a0 (((:p)*)*)* :a1 }`

For these expressions containing nested stars, Sesame produces a run-time error (we have reported this bug in the Sesame's mailing list), and KGRAM does not produce

Table 1. Time in seconds and number of solutions for query Cliq-1 (CliqF-1 for RDF::Query)

n	ARQ	RDFQ	Kgram	Sesame	Solutions
5	1.18	0.90	0.57	0.76	16
6	1.19	1.44	0.60	1.24	65
7	1.37	5.09	0.95	2.36	326
8	1.73	34.01	1.38	9.09	1,957
9	2.31	295.88	5.38	165.28	13,700
10	4.15	2899.41	228.68	–	109,601
11	31.21	–	–	–	986,410
12	1422.30	–	–	–	9,864,101
13	–	–	–	–	–

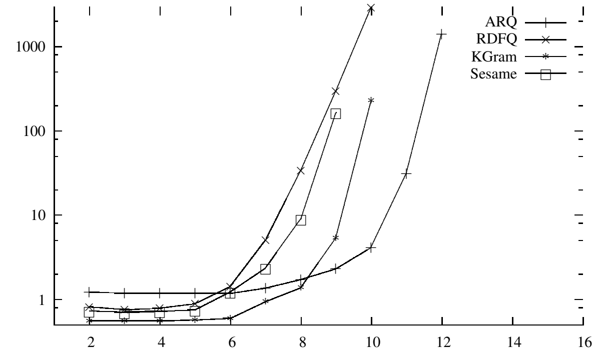

Fig. 4. Time in seconds for processing Cliq-1 w.r.t. the clique size n (time axis in log-scale)

the expected output according to the official SPARQL 1.1 specification [16]. Thus, for these cases it is only meaningful to test ARQ and RDF::Query (we use `FILTER` for RDF::Query, as we did for the case of query CliqF-1). The results are shown in Table 2.

As described in [5], the experimental results show the infeasibility of evaluating property paths including the star operator in the the four tested implementations. We emphasize only here the unexpected impact of nesting stars: for query Cliq-3 both implementations tested fail for an RDF graph representing a clique with only 4 nodes, which contains only 12 triples and has a size of 126 bytes in N3 notation. Although in this example the nesting of the star operator does not seem to be natural, it is well known that nesting is indeed necessary to represent some regular languages [13]. It is also notable how the number of solutions increase w.r.t. the input size. For instance, for query Cliq-1, ARQ returns more than 9 million solutions for a clique with 12 nodes (ARQ's output in this case has more than 9 million lines containing the string | |). We show in Section 5 that the duplication of solutions is indeed the main source of complexity when evaluating property paths.

The Foaf Experiment. The second experiment presented in [5] use real data crawled from the Web. It considered the `foaf:knows` property, as it has been used as

Table 2. Time in seconds and number of solutions for queries Cliq-2 (left) and Cliq-3 (right)

n	ARQ	RDFQ	Solutions
2	1.40	0.76	1
3	1.19	0.84	6
4	1.65	19.38	305
5	97.06	–	418,576
6	–	–	–

n	ARQ	RDFQ	Solutions
2	1.20	0.77	1
3	1.42	6.85	42
4	–	–	–

a paradigmatic property for examples regarding path queries (notice that it is in several of the examples used to describe property paths in the official SPARQL 1.1 specification [16]).

The dataset for this experiment is constructed using the SemWeb Client Library [40], which is part of the Named Graph API for Jena. This library provides a command-line tool `semwebquery` that can be used to query the Web of Linked Data. The tool receives as input a SPARQL query Q, an integer value k and a URI u. When executed, it first retrieves the data from u, evaluates Q over this data, and follows the URIs mentioned in it to obtain more data. This process is repeated k times (see [17] for a description of this query approach). The data is constructed by using a `CONSTRUCT` query to retrieve URIs linked by `foaf:knows` properties with Axel Polleres' foaf document as the starting URI[5]. The parameter k was set as 3, which already produce a file of 1.5MB containing more than 33,000 triples. To obtain a file of reasonable size, the data was filtered by removing all triples that mention URIs from large Social Networks sites (in particular, URIs from MyOpera.com and SemanticTweet.com were removed), and then the *strongly connected component* to which Axel Polleres' URI belongs was extracted, obtaining a file of 25.8 KB. From this file, the authors constructed several test cases by deleting subsets of nodes and then recomputing the strongly connected component. With this process 8 different test cases from 9.2 KB to 25.8 KB were constructed. The descriptions of these files are shown in Table 3. Just as an example of the construction process, file D is constructed from file E by deleting the node corresponding to Richard Cyganiack's URI, and then computing the strongly connected component to which Axel's URI belong.

The following query is used in this experiment:

Foaf-1: `SELECT * WHERE { axel:me (foaf:knows)* ?x }.`

which asks for the network of friends of Axel Polleres. Since the graphs in the test cases are strongly connected, this query retrieves all the nodes in the graph (possibly with duplicates). The time to process the query, the number of solutions produced, and the size of the output produced by ARQ are shown in Table 4 (file E is the last file shown in the table, as all implementations exceed the timeout limit for the larger files). As for the case of the clique experiment, one of the most notable phenomenon is the large increase in the output size.

[5] `http://www.polleres.net/foaf.rdf`

Table 3. Description of the files (name, number of nodes, number of RDF triples, and size in disk) used in the foaf experiment

File	#nodes	#triples	size (N3 format)
A	38	119	9.2KB
B	43	143	10.9KB
C	47	150	11.4KB
D	52	176	13.2KB
E	54	201	14.8KB
F	57	237	17.2KB
G	68	281	20.5KB
H	76	360	25.8KB

Table 4. Time in seconds, number of solutions, and output size for query Foaf-1

File	ARQ	RDFQ	Kgram	Sesame	Solutions	Size (ARQ)
A	5.13	75.70	313.37	–	29,817	2MB
B	8.20	325.83	–	–	122,631	8.4MB
C	65.87	–	–	–	1,739,331	120MB
D	292.43	–	–	–	8,511,943	587MB
E	–	–	–	–	–	–

5.2 Intractability of SPARQL 1.1 in the Presence of Property Paths

In this section, we study the computational complexity of the problem of evaluating SPARQL 1.1 queries containing property paths. Specifically, we study the complexity of computing the function $\text{card}_{\llbracket t \rrbracket_G}(\cdot)$, as this computation embodies the main task needed to evaluate a property-path triple. For the sake of readability, we focus here on computing such functions for property-path triples of the form (a, p, b) where $a, b \in I$. Notice that this is not a restriction, as for every property path triple t and every mapping μ whose domain is equal to the set of variables mentioned in t, it holds that $\text{card}_{\llbracket t \rrbracket_G}(\mu) = \text{card}_{\llbracket \mu(t) \rrbracket_G}(\mu_\emptyset)$ (recall that $\mu_\emptyset$ is the mapping with empty domain). Thus, we study the following *counting* problem:

PROBLEM : COUNTW3C
INPUT : an RDF graph G, elements $a, b \in I$ and a property path p
OUTPUT : $\text{card}_{\llbracket (a,p,b) \rrbracket_G}(\mu_\emptyset)$.

It is important to notice that property paths are part of the input of the previous problem and, thus, we are formalizing the *combined complexity* of the evaluation problem [31]. As it has been observed in many scenarios, and, in particular, in the context of evaluating SPARQL [24], when computing a function like $\text{card}_{\llbracket (a,p,b) \rrbracket_G}(\cdot)$, 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 [31], and it is defined in our context as the complexity of computing $\text{card}_{\llbracket(a,p,b)\rrbracket_G}(\cdot)$ for a fixed property-path p. More precisely, assume given a fixed property path p, and consider the following counting problem:

PROBLEM : CountW3C(p)
INPUT : an RDF graph G, elements $a, b \in I$
OUTPUT : $\text{card}_{\llbracket(a,p,b)\rrbracket_G}(\mu_\emptyset)$.

To pinpoint the complexity of CountW3C and CountW3C(p), where p is a property path, we need to consider the complexity class #P (we refer the reader to [30] for its formal definition). A function f is said to be in #P if there exists a non-deterministic Turing Machine M that works in polynomial time such that for every string w, the value of f on w is equal to the number of accepting runs of M with input w. A prototypical #P-complete problem is the problem of computing, given a propositional formula φ, the number of truth assignments satisfying φ. Clearly #P is a class of intractable computation problems [30].

In [5], the authors prove the following complexity result stating the intractability of property path evaluation.

Theorem 3 ([5]). *The problem* CountW3C(p) *is in* #P *for every property path p. Besides,* CountW3C(c^*) *is* #P*-complete, where $c \in I$.*

Theorem 3 shows that the problem of evaluating property paths under the semantics proposed by the W3C is intractable in data complexity. In fact, it shows that one will not be able to find efficient algorithms to evaluate even simple property paths such as c^*, where c is an arbitrary element of I.

The proof of Theorem 3 reveals that the complexity of the problem CountW3C(p) depends essentially on the way the star symbol is used in p. More precisely, the star height of a property path p, denoted by $\text{sh}(p)$, is the maximum depth of nesting of the star symbols appearing in p [13], that is: (1) $\text{sh}(p) = 0$ if $p \in I$, (2) $\text{sh}(p) = \max\{\text{sh}(p_1), \text{sh}(p_2)\}$ if $p = p_1|p_2$ or $p = p_1/p_2$, and (3) $\text{sh}(p) = \text{sh}(p_1) + 1$ if $p = p_1^*$. Then for every positive integer k, define $\mathcal{SH}_k$ as the class of property paths p such that $\text{sh}(p) \leq k$, and define CountW3C$(\mathcal{SH}_k)$ as the problem of computing, given an RDF graph G, elements $a, b \in I$ and a property path $p \in \mathcal{SH}_k$, the value $\text{card}_{\llbracket(a,p,b)\rrbracket_G}(\mu_\emptyset)$. Then Theorem 3 can be generalized as follows:

Theorem 4 ([5]). CountW3C$(\mathcal{SH}_k)$ *is* #P*-complete for each $k \geq 1$.*

We now move to the study of the combined complexity of the problem CountW3C. In [5], the authors formalized the clique experiment presented in Section 5.1, and then provided lower bounds in this scenario for the number of occurrences of a mapping in the result of the procedure (ALP) used by the W3C to define the semantics of property paths [16]. Interestingly, these lower bounds show that the poor behavior detected in the experiments is not a problem with the tested implementations, but instead a characteristic of the semantics of property paths proposed in [16]. These lower bounds provide strong evidence that evaluating property paths under the semantics proposed by the W3C is completely infeasible, as they show that CountW3C is not even in #P.

Fix an element $c \in I$ and an infinite sequence $\{a_i\}_{i \geq 1}$ of pairwise distinct elements from I, which are all different from c. Then for every $n \geq 2$, let $\mathrm{clique}(n)$ be an RDF graph forming a clique with nodes $a_1, \ldots, a_n$ and edge label c, that is, $\mathrm{clique}(n) = \{(a_i, c, a_j) \mid i, j \in \{1, \ldots, n\} \text{ and } i \neq j\}$. Moreover, for every property path p, define $\textsc{CountClique}(p, n)$ as $\mathrm{card}_{[\![(a_1,p,a_n)]\!]_{\mathrm{clique}(n)}}(\mu_\emptyset)$.

Lemma 1 ([5]). *For every property path* p *and* $n \geq 2$*:*

$$\textsc{CountClique}(p^*, n) = \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \textsc{CountClique}(p, n)^k}{(n-k-1)!}$$

Let $p_0 = c$ and $p_{s+1} = p_s^*$, for every $s \geq 0$. For example, $p_1 = c^*$ and $p_3 = ((c^*)^*)^*$. From Lemma 1, we obtain that:

$$\textsc{CountClique}(p_{s+1}, n) \;=\; \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \textsc{CountClique}(p_s, n)^k}{(n-k-1)!}, \qquad (1)$$

for every $s \geq 0$. This formula can be used to obtain the number of occurrences of the mapping with empty domain in the answer to the property-path triple (a_1, p_s, a_n) over the RDF graph $\mathrm{clique}(n)$. For instance, the formula states that if a system implements the semantics proposed by the W3C [16], then with input $\mathrm{clique}(8)$ and $(a_1, (c^*)^*, a_8)$, the empty mapping would have to appear more than $79 \cdot 10^{24}$ times in the output. Thus, even if a single byte is used to store the empty mapping[6], then the output would be of more than 79 Yottabytes in size! Table 5 shows more lower bounds obtained with formula (1). Notice that these numbers coincide with the results obtained in the reported experiments (Tables 1 and 2). Also notice that, for example, for $n = 6$ and $s = 2$ the lower bound is of more than 28 billions, and for $n = 4$ and $s = 3$ is of more than 56 millions, which explains why the tested implementations exceeded the timeout for queries Cliq-2 and Cliq-3 (Table 2).

Most notably, Table 5 allows one to provide a *cosmological lower bound* for evaluating property paths: if one proton is used to store the mapping with empty domain, with input $\mathrm{clique}(6)$ (which contains only 30 triples) and the property-path triple

$$(a_1, (((c^*)^*)^*)^*, a_6),$$

every system implementing the semantics proposed by the W3C [16] would have to return a file that would not fit in the observable universe!

From Lemma 1, the following double-exponential lower bound can be provided for the complexity of $\textsc{CountClique}(p_s, n)$.

Lemma 2 ([5]). *For every* $n \geq 2$ *and* $s \geq 1$*:*

$$\textsc{CountClique}(p_s, n) \geq (n-2)!^{(n-1)^{s-1}}$$

From this bound, we obtain that CountW3C is not in #P. Besides, from the proof of Theorem 3, it can be shown that CountW3C is in the complexity class #EXP, which is defined as #P but considering non-deterministic Turing Machines that work in exponential time.

[6] Recall that the empty mapping $\mu_\emptyset$ is represented as the four-bytes string `|  |` in ARQ, and as the two-bytes string `[]` in Sesame.

Table 5. Number of occurrences of the mapping with empty domain in the answer to property-path triple (a_1, p_s, a_n) over the RDF graph clique(n), according to the semantics for property paths proposed by the W3C in [16]

s	n	CountClique(p_s, n)	s	n	CountClique(p_s, n)
1	3	2	1	5	16
2	3	6	2	5	418576
3	3	42	3	5	$> 10^{23}$
4	3	1806	4	5	$> 10^{93}$
1	4	5	1	6	65
2	4	305	2	6	28278702465
3	4	56931605	3	6	$> 10^{53}$
4	4	$> 10^{23}$	4	6	$> 10^{269}$

Theorem 5 ([5]). CountW3C *is in* #EXP *and not in* #P.

It is open whether CountW3C is #EXP-complete.

The Complexity of the Entire Language. We consider now the data complexity of the evaluation problem for the entire language. More precisely, we use the results presented in the previous section to show the major impact of using property paths on the complexity of evaluating SPARQL 1.1 queries. The evaluation problem is formalized as follows. Given a fixed SPARQL 1.1 query Q, define EvalW3C(Q) as the problem of computing, given an RDF graph G and a mapping μ, the value $\text{card}_{[\![Q]\!]_G}(\mu)$.

It is easy to see that the data complexity of SPARQL 1.1 without property paths is polynomial. However, from Theorem 3, we obtain the following corollary that shows that the data complexity is considerably higher if property paths are included, for the case of the semantics proposed by the W3C [16]. The following corollary states that EvalW3C(Q) is in the complexity class $\text{FP}^{\#\text{P}}$, which is the class of functions that can be computed in polynomial time if one has access to an efficient subroutine for a #P-complete problem (or, more formally, one has an *oracle* for a #P-complete problem).

Corollary 1 ([5]). EvalW3C(Q) *is in* $\text{FP}^{\#\text{P}}$*, for every SPARQL 1.1 query Q. Moreover, there exists a SPARQL 1.1 query* Q_0 *such that* EvalW3C(Q_0) *is* #P*-hard.*

5.3 Intractability for Alternative Semantics that Count Paths

In [5,19], the authors consider some alternative semantics for property paths that take into account the cardinality of solutions. In this section, we focus on the two alternative semantics proposed in [5], showing that both leads to intractability.

The usual graph theoretical notion of path has been extensively and successfully used when defining the semantics of queries including regular expressions [21,9,2,25,6]. Nevertheless, given that the W3C SPARQL 1.1 Working Group is interested in counting paths, the classical notion of path in a graph cannot be naively used to define a semantics for property-path queries, given that cycles in an RDF graph may lead to an infinite number of different paths. In this section, we consider two alternatives to deal with this problem that were introduced in [5]. We consider a semantics for property paths based on classical paths that is only defined for *acyclic RDF graphs*, and we consider a general semantics that is based on *simple paths* (which are paths in a graph with no repeated nodes). In both cases, the query evaluation based on counting is intractable [5]. Next we formalize these two alternative semantics and present their complexity.

A path π in an RDF graph G is a sequence $a_1, c_1, a_2, c_2, \ldots, a_n, c_n, a_{n+1}$ such that $n \geq 0$ and $(a_i, c_i, a_{i+1}) \in G$ for every $i \in \{1, \ldots, n\}$. Path π is said to be from a to b in G if $a_1 = a$ and $a_{n+1} = b$, it is said to be nonempty if $n \geq 1$, and it is said to be a *simple path*, or just s-path, if $a_i \neq a_j$ for every distinct pair i, j of elements from $\{1, \ldots, n+1\}$. Finally, given a property path p, path π is said to *conform* to p if $c_1 c_2 \cdots c_n$ is a string in the regular language defined by p.

Classical Paths over Acyclic RDF Graphs. We first define the semantics of a property-path triple considering classical paths, that we denote by $[\![\cdot]\!]_G^{\text{path}}$. Notice that we have to take into consideration the fact that the number of paths in an RDF graph may be infinite, and thus we define this semantics only for acyclic graphs. More precisely, an RDF graph G is said to be cyclic if there exists an element a mentioned in G and a nonempty path π in G from a to a, and otherwise it is said to be acyclic. Then assuming that G is acyclic, the evaluation of a property-path triple t over G in terms of classical paths, denoted by $[\![t]\!]_G^{\text{path}}$, is defined as follows. Let $t = (u, p, v)$ and $W = (\{u, v\} \cap V)$, then

$$\begin{aligned}[\![t]\!]_G^{\text{path}} \;=\; \{\mu \mid \text{dom}(\mu) = W \text{ and there exists a} \\ \text{path from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p\},\end{aligned}$$

and for every $\mu \in [\![t]\!]_G^{\text{path}}$, the value $\text{card}_{[\![t]\!]_G^{\text{path}}}(\mu)$ is defined as the number of paths from $\mu(u)$ to $\mu(v)$ in G that conform to p.

Similarly as we defined the problem COUNTW3C in Section 5.2, we define the problem COUNTPATH as the following counting problem.

PROBLEM : COUNTPATH
INPUT : an acyclic RDF graph G, elements $a, b \in I$ and a property path p
OUTPUT : $\text{card}_{[\![(a,p,b)]\!]_G^{\text{path}}}(\mu_\emptyset)$.

We also define, given a fixed property path p, the problem COUNTPATH(p) as the the problem of computing, given an acyclic RDF graph G and elements $a, b \in I$, the value $\text{card}_{[\![(a,p,b)]\!]_G^{\text{path}}}(\mu_\emptyset)$.

To pinpoint the exact complexity of the problems COUNTPATH and COUNTPATH(p), we need to consider two counting complexity classes: #L and SPANL. We introduce these classes here, and we refer the reader to [3] for their formal definitions. #L is the counting class associated with the problems that can be solved in logarithmic space in a non-deterministic Turing Machine (NTM). In fact, a function f is said to be in this class if there exists an NTM M that works in logarithmic space such that for every string w, the value of f on w is equal to the number of accepting runs of M with input w.

A prototypical #L-complete problem is the problem of computing, given a deterministic finite automaton A and a string w, the number of strings that are accepted by A and whose length is smaller than the length of w [3]. SPANL is defined in a similar way to #L, but considering logarithmic-space NTMs with output. More precisely, a function f is said to be in this class if there exists such TM M such that for every string w, the value of f on w is equal to the number of different outputs of M with input w. A prototypical SPANL-complete problem is the problem of computing, given a non-deterministic finite automaton A and a string w, the number of strings that are accepted by A and whose length is smaller than the length of w [3]. Although classes #L and SPANL look alike, they are quite different in terms of complexity: #L is known to be included in FP, the class of functions that can be computed in polynomial time, while it is known that SPANL is a class of intractable computation problems, if SPANL $\subseteq$ FP, then P $=$ NP.

It was proved in [5] that even for the simple case considered in this section, the problem of evaluating property paths is intractable.

Theorem 6 ([5]). COUNTPATH *is* SPANL*-complete.*

Interestingly, one can show that at least in terms of data complexity, the problem of evaluating property paths is tractable if their semantics is based on the usual notion of path.

Theorem 7 ([5]). COUNTPATH(p) *is in* #L *for every property path* p*. Moreover, there exists a property path* p_0 *such that* COUNTPATH(p_0) *is* #L*-complete.*

Although COUNTPATH(p) is tractable, it only considers acyclic RDF graphs, and thus leaves numerous practical cases uncovered.

Simple Paths. We continue our investigation by considering the alternative semantics for property paths that is defined in terms of simple paths. Notice that even for cyclic RDF graphs, the number of simple paths is finite, and thus, this semantics is properly defined for every RDF graph. Formally, assume that G is an RDF graph, $t = (u, p, v)$ is a property-path triple and $W = (\{u, v\} \cap V)$. The evaluation of t over G in terms of s-paths, denoted by $[\![t]\!]_G^{\text{s-path}}$, is defined as:

$$[\![t]\!]_G^{\text{s-path}} = \{\mu \mid \text{dom}(\mu) = W \text{ and there exists an s-path} \\ \text{from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p\},$$

and for every $\mu \in [\![t]\!]_G^{\text{s-path}}$, the value $\text{card}_{[\![t]\!]_G^{\text{s-path}}}(\mu)$ is defined as the number of s-paths from $\mu(u)$ to $\mu(v)$ in G that conform to p. For the case of s-paths, we define the problem COUNTSIMPLEPATH as follows.

PROBLEM :	COUNTSIMPLEPATH
INPUT :	an RDF graph G, elements $a, b \in I$ and a property path p
OUTPUT :	$\text{card}_{[\![(a,p,b)]\!]_G^{\text{s-path}}}(\mu_\emptyset)$.

As for the previous cases, we define the counting problem COUNTSIMPLEPATH(p) as COUNTSIMPLEPATH for a fixed property path p. The following result shows that these problems are also intractable.

Theorem 8 **([5,19]).** *The problem* COUNTSIMPLEPATH *is in* #P*. Moreover, if* $c \in I$*, then the problem* COUNTSIMPLEPATH(c^*) *is* #P*-complete.*

Notice that the data complexity of evaluating property paths according to the s-path semantics is the same as evaluating them according to the W3C semantics. The difference is in the combined complexity that is radically higher for the W3C semantics: for the case of the semantics based on s-paths the combined complexity is in #P, while for the W3C semantics it is not in #P (Theorem 5).

5.4 An Existential Semantics for Evaluating Property Paths

We have shown in the previous section that evaluating property-path triples according to the semantics proposed in [16] is essentially infeasible, being the core of this problem the necessity of counting different paths. We have also shown that the version in which one counts simple-paths is infeasible too. As described in [5], a possible solution to this problem is to not use a semantics that requires counting paths, but instead a more traditional existential semantics for property-path triples. That is, one just checks if two nodes are connected (or not) by a path that conforms to a property-path expression, without considering whether there are several paths connecting two nodes. This existential semantics has been used for years in graph databases [21,9,6], in XML [20,15], and even on RDF [2,25] previous to SPARQL 1.1. In this section, we introduce this semantics and study the complexity of evaluating property paths, and also SPARQL 1.1 queries, under it based on the results presented in [5]. We also compare this proposal with the current official semantics for property paths, and present some experimental results.

The most natural way to define an existential semantics for property paths is to consider the existential version of the semantics $[\![\cdot]\!]_G^{\mathrm{path}}$. More precisely, assume that $u, v \in (I \cup V)$, $W = (\{u, v\} \cap V)$, $t = (u, p, v)$ is a property-path triple, and G is an RDF graph. Then define $[\![t]\!]_G^{\exists(\mathrm{path})}$ as:

$$[\![t]\!]_G^{\exists(\mathrm{path})} = \{\mu \mid \mathrm{dom}(\mu) = W \text{ and there exists a path} \\ \text{from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p\}.$$

Moreover, define the cardinality of every mapping μ in $[\![t]\!]_G^{\exists(\mathrm{path})}$ just as 1. Notice that with the semantics $[\![t]\!]_G^{\exists(\mathrm{path})}$, we are essentially discarding all the duplicates from $[\![t]\!]_G^{\mathrm{path}}$. This allows us to consider general graphs (not necessarily acyclic graph as in Section 5). To study the complexity of evaluating property paths under this semantics, we define the decision problem EXISTSPATH.

PROBLEM :	EXISTSPATH
INPUT :	an RDF graph G, elements $a, b \in I$ and a property path p
OUTPUT :	is $\mathrm{card}_{[\![(a,p,b)]\!]_G^{\exists(\mathrm{path})}}(\mu_\emptyset) = 1$?

That is, the problem EXISTSPATH is equivalent to checking if $\mu_\emptyset$ is in $[\![(a, p, b)]\!]_G^{\exists(\mathrm{path})}$.

Notice that with ExistsPath, we are measuring the combined complexity of evaluating paths under the existential semantics. The following result shows that ExistsPath is tractable. This is a corollary of some well-known results on graph databases (e.g. see Section 3.1 in [25]). In the result, we use $|G|$ to denote the size of an RDF graph G and $|p|$ to denote the size of a property-path p.

Proposition 3. ExistsPath *can be solved in time* $O(|G| \cdot |p|)$.

Discarding Duplicates from the Standard and Simple-Paths Semantics. A natural question at this point is whether there exists a relationship between the existential semantics defined in the previous section and the semantics that can be obtained by discarding duplicates from $[\![t]\!]_G$ and $[\![t]\!]_G^{\text{s-path}}$ for a property-path triple t. As shown in [5], a relationship does exists, which can be considered as an important step towards the optimization of property-paths evaluation.

Assume that G is an RDF graph and t is a property-path triple. Then we define $[\![t]\!]_G^{\exists}$ as having exactly the same mappings as in $[\![t]\!]_G$, but with the cardinality of every mapping in $[\![t]\!]_G^{\exists}$ defined just as 1. Similarly, we define $[\![t]\!]_G^{\exists(\text{s-path})}$ as having exactly the same mappings as in $[\![t]\!]_G^{\text{s-path}}$, but with the cardinality of every mapping in $[\![t]\!]_G^{\exists(\text{s-path})}$ defined as 1. Thus, in this section, we study the decision problems ExistsW3C and ExistsSimplePath, which are defined as follows:

Problem : ExistsW3C
Input : an RDF graph G, elements $a, b \in I$ and a property path p
Question : is $\text{card}_{[\![(a,p,b)]\!]_G^{\exists}}(\mu_\emptyset) = 1$?

Problem : ExistsSimplePath
Input : an RDF graph G, elements $a, b \in I$ and a property path p
Question : is $\text{card}_{[\![(a,p,b)]\!]_G^{\exists(\text{s-path})}}(\mu_\emptyset) = 1$?

The following result shows that, somehow surprisingly, the semantics $[\![\cdot]\!]_G^{\exists}$ coincides with $[\![\cdot]\!]_G^{\exists(\text{path})}$. Thus, even though the official semantics of property paths is given in terms of a particular procedure [16], when one does not count paths, it coincides with the classical existential semantics based on the usual notion of path.

Theorem 9 ([5]). *For every RDF graph G, mapping μ and property-path triple t it holds that $\mu \in [\![t]\!]_G^{\exists}$ if and only if $\mu \in [\![t]\!]_G^{\exists(\text{path})}$.*

As a corollary of Propositions 3 and Theorem 9, one obtains that:

Theorem 10 ([5]). ExistsW3C *can be solved in time* $O(|G| \cdot |p|)$.

The situation is radically different for the case of simple paths. From some well-known results on graph databases [21], one can prove that ExistsSimplePath is an intractable problem, even for a fixed property-path. More precisely, for a fixed property-path p, the decision problem ExistsSimplePath(p) has as input an RDF graph G and elements $a, b \in I$, and the question is whether $\text{card}_{[\![(a,p,b)]\!]_G^{\exists(\text{s-path})}}(\mu_\emptyset) = 1$.

Proposition 4 ([21,5,19]). *The problem* ExistsSimplePath *is in* NP. *Moreover, the problem* ExistsSimplePath$((c/c)^*)$ *is* NP*-complete, where $c \in I$.*

Existential Semantics and SPARQL 1.1. We have shown that when bags are considered for the semantics of property paths, the evaluation becomes intractable, even in data complexity. However, the previous version of SPARQL, that did not include path queries, considered a bag semantics for the mapping operators (AND, OPT, UNION, FILTER and SELECT), which has proved to be very useful in practice. Thus, a natural question is whether one can construct a language with functionalities to express interesting queries about paths in RDF graphs, with bag semantics for the mappings operators, and that, at the same time, can be efficiently evaluated. In this section, we give a positive answer to this question. It was shown in [5] that if one combines existential semantics for property paths and bag semantics for the SPARQL 1.1 operators, one obtains the best of both worlds and still has tractable data complexity. Moreover, it was also shown in [5] that considering simple paths is not an option as the evaluation problem is still intractable in data complexity.

We start by formalizing this alternative way of evaluating SPARQL 1.1 queries that considers existential semantics for property-path triples. Given a SPARQL 1.1 query Q, define $[\![Q]\!]^{\exists}_{G}$ exactly as $[\![Q]\!]_{G}$ is defined in Sections 4.1 and 4.2, but evaluating property-paths triples according to the semantics $[\![\cdot]\!]^{\exists}_{G}$ defined in Section 5.4 (that is, $[\![t]\!]_{G}$ is replaced by $[\![t]\!]^{\exists}_{G}$ if t is a property-path triple), and likewise for $[\![Q]\!]^{\exists(\text{s-path})}_{G}$ and $[\![Q]\!]^{\exists(\text{path})}_{G}$. Notice that for the three semantics $[\![Q]\!]^{\exists}_{G}$, $[\![Q]\!]^{\exists(\text{path})}_{G}$ and $[\![Q]\!]^{\exists(\text{s-path})}_{G}$, we are not discarding all duplicates but only the duplicates that are generated when evaluating property paths. Thus, these semantics are still bag semantics and, therefore, we consider the computation problems associated to them to study their complexity.

Let Q be a SPARQL 1.1 query. We define first the following computation problem.

PROBLEM : EVALEXISTSW3C(Q)
INPUT : an RDF graph G, and a mapping μ
OUTPUT : $\mathrm{card}_{[\![Q]\!]^{\exists}_{G}}(\mu)$

Moreover, we also consider the computation problems EVALEXISTSSIMPLEPATH(Q) and EVALEXISTSPATH(Q), defined as EVALEXISTSW3C(Q) but consider as output the values $\mathrm{card}_{[\![Q]\!]^{\exists(\text{s-path})}_{G}}(\mu)$ and $\mathrm{card}_{[\![Q]\!]^{\exists(\text{path})}_{G}}(\mu)$, respectively.

PROBLEM : EVALEXISTSSIMPLEPATH(Q)
INPUT : an RDF graph G, and a mapping μ
OUTPUT : $\mathrm{card}_{[\![Q]\!]^{\exists(\text{s-path})}_{G}}(\mu)$

PROBLEM : EVALEXISTSPATH(Q)
INPUT : an RDF graph G, and a mapping μ
OUTPUT : $\mathrm{card}_{[\![Q]\!]^{\exists(\text{path})}_{G}}(\mu)$

Notice that in these three problems, we are considering the data complexity of SPARQL 1.1 under the respective semantics.

Notably, the next result shows that the just defined semantics $[\![\cdot]\!]^{\exists}_{G}$ and $[\![\cdot]\!]^{\exists(\text{path})}_{G}$ are tractable, in terms of data complexity. This result is a consequence of Theorem 10 and

Proposition 3. In the formulation of this result we use the class FP, which is defined as the class of all functions that can be computed in polynomial time (and thus, it is a class of tractable functions).

Theorem 11 ([5]). EvalExistsW3C(Q) *and* EvalExistsPath(Q) *are in* FP *for every SPARQL 1.1 query Q.*

We conclude this section by showing that for the case of the semantics $[\![\cdot]\!]_G^{\exists(\text{s-path})}$, the data complexity is unfortunately still high. To study this problem we need the complexity classes $\text{FP}^{\|\text{NP}}$ and $\text{FP}^{\text{NP}[O(\log n)]}$, which are defined in terms of oracles as for the case of the complexity class $\text{FP}^{\#\text{P}}$ used in Corollary 1. More precisely, the class $\text{FP}^{\|\text{NP}}$ contains all the functions that can be computed in polynomial time by a procedure that is equipped with an efficient subroutine (oracle) for an NP-complete problem, with the restriction that all the calls to the subroutine should be made *in parallel*, that is, no call to the subroutine can depend on the result of a previous call to this subroutine [32]. The class $\text{FP}^{\text{NP}[O(\log n)]}$ is defined in the same way, but with the restriction that the subroutine for an NP-complete problem can be called only a logarithmic number of times. Both classes $\text{FP}^{\text{NP}[O(\log n)]}$ and $\text{FP}^{\|\text{NP}}$ are considered to be intractable. Moreover, it is known that $\text{FP}^{\text{NP}[O(\log n)]} \subseteq \text{FP}^{\|\text{NP}}$, but it is open whether this containment is strict [29].

Theorem 12 ([5]). EvalExistsSimplePath(Q) *is in* $\text{FP}^{\|\text{NP}}$ *for every SPARQL 1.1 query Q. Moreover, there exists a query* Q_0 *such that* Q_0 *mentions only operators* AND *and* SELECT*, and* EvalExistsSimplePath(Q_0) *is* $\text{FP}^{\text{NP}[O(\log n)]}$*-hard.*

Theorem 12 shows that simple paths are not a good option even if duplicates are not considered. The exact complexity of EvalExistsSimplePath(Q) is an open problem [5].

5.5 Optimizing Property Path Queries with DISTINCT

As discussed in the previous section, SPARQL 1.1 is tractable in terms of data complexity if one considers the existential semantics $[\![\cdot]\!]_G^{\exists}$ and $[\![\cdot]\!]_G^{\exists(\text{path})}$ for property paths. The goal of this section is to show the impact of using these semantics in practice, by reporting on some experimental results obtained with two implementations that

Table 6. Time in seconds for processing Foaf-1D

Input	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen
A	2.24	47.31	2.37	–	0.29	1.39
B	2.60	204.95	6.43	–	0.30	1.32
C	6.88	3222.47	80.73	–	0.30	1.34
D	24.42	–	394.61	–	0.31	1.38
E	–	–	–	–	0.33	1.38
F	–	–	–	–	0.35	1.42
G	–	–	–	–	0.44	1.50
H	–	–	–	–	0.45	1.52

Table 7. Time in seconds for queries Cliq-1D, Cliq-2D, and Cliq-3D

n	ARQ	RDFQ	Kgram	Sesame	Psparql	Gleen
8	1.68	32.61	1.39	9.08	0.18	1.24
9	2.00	213.99	5.34	166.82	0.20	1.23
10	3.65	2123.90	227.66	–	0.20	1.25
11	29.71	–	–	–	0.23	1.25
12	1394.06	–	–	–	0.24	1.24
13	–	–	–	–	0.27	1.24

Cliq-1D

n	ARQ	RDFQ	Psparql	Gleen
2	1.40	0.76	0.14	1.23
3	1.19	0.84	0.14	1.23
4	1.65	19.38	0.14	1.23
5	97.06	–	0.15	1.22
6	–	–	0.16	1.23
7	–	–	0.16	1.23

Cliq-2D

n	ARQ	RDFQ	Psparql	Gleen
2	1.18	0.77	0.14	1.24
3	1.41	6.78	0.14	1.23
4	–	–	0.15	1.24
5	–	–	0.15	1.24
6	–	–	0.16	1.24
7	–	–	0.16	1.24

Cliq-3D

extends SPARQL 1.0 with existential path semantics: Psparql (version 3.3) [37], and Gleen (version 0.6.1) [39]. These two implementations evaluate SPARQL queries according to $[\![\cdot]\!]_G^{\exists(\text{path})}$, although they use a slightly different syntax for path queries. We also show that SPARQL 1.1 queries can be highly optimized by making use of the `SELECT DISTINCT` feature. This result is of highly practical interest since, as shown by the experiments, no current implementation of SPARQL 1.1 is taking advantage of queries that use `SELECT DISTINCT`.

We begin by stating the relationship between the existential semantics and the standard semantics in the presence of DISTINCT. It is important to notice that in the following result we consider the fragment of SPARQL 1.1 introduced in Section 4. In particular, we do not consider some features of SPARQL 1.1 introduced in [16], such as subqueries and aggregates.

Theorem 13 ([5]). *Let P be a SPARQL 1.1 graph pattern, G an RDF graph and W a set of variables. Then we have that:*

$$[\![(\text{SELECT DISTINCT } W\ P)]\!]_G = [\![(\text{SELECT DISTINCT } W\ P)]\!]_G^{\exists(\text{path})}$$
$$[\![(\text{SELECT DISTINCT } *\ P)]\!]_G = [\![(\text{SELECT DISTINCT } *\ P)]\!]_G^{\exists(\text{path})}$$

In view of this theorem, in [5] the authors present an experimental evaluation considering all the queries presented in Section 5.1, but using the `SELECT DISTINCT` feature, that is:

Cliq-1D: `SELECT DISTINCT * WHERE { :a0 (:p)* :a1 }`
Cliq-2D: `SELECT DISTINCT * WHERE { :a0 ((:p)*)* :a1 }`
Cliq-3D: `SELECT DISTINCT * WHERE { :a0 (((:p)*)*)* :a1 }`
Foaf-1D: `SELECT DISTINCT * WHERE { axel:me (foaf:knows)* ?x }`

Tables 6 and 7, taken from [5], show the results of this experiment. Although all the tested systems return the same results for these queries, the differences in efficiency between the SPARQL 1.1 implementations and the implementations that use an existential semantics are dramatic.

6 Concluding Remarks

SPARQL is now widely used as the standard RDF query language. The new version of the language, SPARQL 1.1, has been in the process of standardization since 2009, and it is going to include several features not presented in the previous version. Even before SPARQL 1.1 becomes a standard, the research community has been investigating some properties of the language, providing valuable feedback to the groups in charge of the standardization process.

In this paper, we have focused on two of the most interesting and useful features introduced in SPARQL 1.1: federation and navigation. These features not only add expressiveness to the language but also pose several interesting and challenging research questions. We presented a formalization of these new features and studied their expressiveness and complexity, showing some theoretical and empirical results. We also presented some recently discovered issues that can have a significant impact on the adoption of the new standard, and we discussed on possible alternatives and their implications. As of the beginning of year 2012, these issues are being carefully discussed in the groups in charge of the standard.

Acknowledgments. Several of the results surveyed in this paper were presented in some articles of the authors and their co-authors Carlos Buil-Aranda, Sebastián Conca, Oscar Corcho, Claudio Gutierrez and Axel Polleres. Marcelo Arenas was supported by Fondecyt grant 1110287, and Jorge Pérez by Fondecyt grant 11110404 and by VID grant U-Inicia 11/04 Universidad de Chile.

References

1. Alkhateeb, F., Baget, J.-F., Euzenat, J.: Constrained regular expressions in SPARQL. In: SWWS, pp. 91–99 (2008)
2. Alkhateeb, F., Baget, J.-F., Euzenat, J.: Extending SPARQL with regular expression patterns (for querying RDF). JWS 7(2), 57–73 (2009)
3. Àlvarez, C., Jenner, B.: A very hard log-space counting class. Theor. Comput. Sci. 107(1), 3–30 (1993)
4. Anyanwu, K., Maduko, A., Sheth, A.P.: Sparq2l: towards support for subgraph extraction queries in rdf databases. In: WWW, pp. 797–806 (2007)
5. Arenas, M., Conca, S., Pérez, J.: Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In: WWW, pp. 629–638 (2012)
6. Barceló, P., Hurtado, C.A., Libkin, L., Wood, P.T.: Expressive languages for path queries over graph-structured data. In: PODS, pp. 3–14 (2010)
7. Buil-Aranda, C., Arenas, M., Corcho, O.: Semantics and Optimization of the SPARQL 1.1 Federation Extension. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part II. LNCS, vol. 6644, pp. 1–15. Springer, Heidelberg (2011)

8. Buil-Aranda, C., Arenas, M., Corcho, Ó., Polleres, A.: Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. Submitted for Journal Publication
9. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries. In: PODS, pp. 194–204 (1999)
10. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW (Alternate Track Papers & Posters), pp. 74–83 (2004)
11. Corby, O., Faron-Zucker, C.: The kgram abstract machine for knowledge graph querying. In: Web Intelligence, pp. 338–341 (2010)
12. Durst, M., Suignard, M.: Rfc 3987, internationalized resource identifiers (iris) (2005)
13. Eggan, L.C.: Transition graphs and the star-height of regular events. The Michigan Mathematical Journal 10(4), 385–397 (1963)
14. Glimm, B., Ogbuji, C.: Sparql 1.1 entailment regimes (2010)
15. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. TODS 30(2), 444–491 (2005)
16. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C Working Draft, (January 5, 2012), http://www.w3.org/TR/2012/WD-sparql11-query-20120105/
17. Hartig, O., Bizer, C., Freytag, J.-C.: Executing SPARQL Queries over the Web of Linked Data. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 293–309. Springer, Heidelberg (2009)
18. 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)
19. Losemann, K., Martens, W.: The complexity of evaluating path expressions in sparql. In: PODS, pp. 101–112 (2012)
20. Marx, M.: Conditional XPath. TODS 30(4), 929–959 (2005)
21. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM J. Comput. 24(6), 1235–1258 (1995)
22. Olson, M., Ogbuji, U.: The Versa specification, http://uche.ogbuji.net/tech/rdf/versa/etc/versa-1.0.xml
23. 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)
24. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. TODS 34(3) (2009)
25. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. JWS 8(4), 255–270 (2010)
26. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF (January 2008)
27. Prud'hommeaux, E., Buil-Aranda, C.: SPARQL 1.1 federated query. W3C Working Draft (November 17, 2011), http://www.w3.org/TR/2011/+WD-sparql11-federated-query-20111117/
28. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (January 15 (2008), http://www.w3.org/TR/rdf-sparql-query/
29. Selman, A.L.: A taxonomy of complexity classes of functions. J. Comput. Syst. Sci. 48(2), 357–381 (1994)
30. Valiant, L.G.: The complexity of computing the permanent. Theor. Comput. Sci. 8, 189–201 (1979)
31. Vardi, M.Y.: The complexity of relational query languages (extended abstract). In: STOC, pp. 137–146 (1982)

32. Wagner, K.W.: More complicated questions about maxima and minima, and some closures of NP. Theor. Comput. Sci. 51, 53–80 (1987)
33. ARQ, `http://sourceforge.net/projects/jena/files/ARQ/`
34. KGRAM, `http://www-sop.inria.fr/edelweiss/software/corese/`
35. RDF: Query, `http://search.cpan.org/~gwilliams/RDF-Query`
36. Sesame, `http://sourceforge.net/projects/sesame/`
37. Psparql, `http://exmo.inrialpes.fr/software/psparql/`
38. RDF data access working group mailing list, `http://lists.w3.org/Archives/Public/public-rdf-dawg-comments/`
39. Gleen, `http://sig.biostr.washington.edu/projects/ontviews/gleen/`
40. Semantic Web Client Library, `http://www4.wiwiss.fu-berlin.de/bizer/ng4j/semwebclient/`