SPARQL with Property Paths

Egor V. Kostylev[1], Juan L. Reutter[2], Miguel Romero[3],
and Domagoj Vrgoč[2](✉)

[1] University of Oxford, Oxford, UK
egor.kostylev@cs.ox.ac.uk
[2] PUC Chile and Center for Semantic Web Research, Santiago, Chile
{jreutter,dvrgoc}@ing.puc.cl
[3] University of Chile and Center for Semantic Web Research, Santiago, Chile
mromero@dcc.uchile.cl

Abstract. The original SPARQL proposal was often criticized for its inability to navigate through the structure of RDF documents. For this reason property paths were introduced in SPARQL 1.1, but up to date there are no theoretical studies examining how their addition to the language affects main computational tasks such as query evaluation, query containment, and query subsumption. In this paper we tackle all of these problems and show that although the addition of property paths has no impact on query evaluation, they do make the containment and subsumption problems substantially more difficult.

1 Introduction

Following the initial proposal for the SPARQL 1.0 query language [22] a lot of work has been done by the theory community to study its basic properties. A seminal paper by Pérez et al. [16] gave us a clean theoretical foundation for the study of the language, and by now we understand very well the complexity of query evaluation [12,17], as well as the issues related to basic static analysis tasks such as containment and equivalence [12,20,21].

However, with the growth of RDF data available on the Web, also came the need for features not present in the original proposal. One such feature should allow to navigate though RDF documents and discover how different resourses are connected. This becomes apparent when considering applications such as linked data where the local topology of the document often does not provide sufficient information, and long chains have to be followed to obtain the desired answer. For this reason the W3C included *property paths* in the specification of SPARQL 1.1 [10], an extension of the original language with several important features.

Intuitively, a property path searches through the RDF graph for a sequence of IRIs that form a path conforming to an regular expression. For example, to infer that one property is a subclass of another we could ask a query $(?x, \texttt{subclass}^*, ?y)$ and check if our pair is in the answer. Here the property path is given by the regular expression $\texttt{subclass}^*$, which specifies that we can traverse an arbitrary number of subclass property links in order to reach $?y$ from $?x$.

M. Arenas et al. (Eds.): ISWC 2015, Part I, LNCS 9366, pp. 3–18, 2015.
DOI: 10.1007/978-3-319-25007-6_1

Although some work has been done on SPARQL with different forms of navigation [1–3,8,9,14,18,24], little is known about the language that has property paths as specified in the latest standard [10]. Therefore, our goal is to study theoretical aspects of SPARQL with this functionality. In particular, in this paper we focus on the fundamental problems of query evaluation, containment, and subsumption. The first one is key for understanding the properties of any query language, while the other two are of fundamental importance in query optimization, ontological reasoning, and managing incomplete information.

So far, these problems have been studied for fragments of SPARQL that allow only basic operators such as AND, UNION, SELECT, and OPTIONAL (abbreviated as OPT in this paper) [9,12,20]. It is therefore interesting to see how property paths mix with the previous results on core SPARQL. A natural approach here would be to use techniques from the field of graph databases. After all, RDF triples closely resemble edges in a labelled graph, and property paths are similar to *regular path queries* [5]. However, we will show that this cannot be done directly, as not only RDF data model is richer than usual graphs [13], but also the SPARQL 1.1 standard allows for negation in property paths, which is known to make things more difficult [11,15]. Another challenge is the presence of the OPT operator (which is not usually included in graph database query languages) and the way it interacts with property paths. We will show that techniques for SPARQL without property paths [12,20] cannot be straightforwardly adapted to deal with the general language. To this end, we develop new techniques that merge the approaches of [5,20] and use them to obtain matching complexity bounds for the considered problems.

We begin in Section 3 with a formalisation of property paths according to the latest specification [10]. We also pinpoint the differences between the resulting language and known formalisms, and discuss the difficulties they impose on possible adaptations of known techniques for solving the considered problems. Then, in Section 4, we study evaluation, containment, and subsumption for SPARQL with property paths that do not allow for optional matching. In particular, using techniques from automata theory we show that in this case property paths do not increase the complexity of evaluation, but have a significant effect on containment and subsumption. Finally, in Section 5 we study the full language, with both property paths and optional matching. Blending standard SPARQL and graph databases techniques we can show that adding OPT usually makes evaluation more difficult, but almost always leaves the complexity of the optimisation problems intact.

2 Preliminaries

RDF Graphs. Let $\mathbf{I}$, $\mathbf{L}$, and $\mathbf{B}$ be countably infinite disjoint sets of *IRIs*, *literals*, and *blank nodes*, respectively. The set of *RDF terms* $\mathbf{T}$ is $\mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$. An *RDF triple* is a triple (s, p, o) from $\mathbf{T} \times \mathbf{I} \times \mathbf{T}$, where s is called *subject*, p *predicate*, and o *object*. An *(RDF) graph* is a finite set of RDF triples.

SPARQL Syntax. SPARQL is the standard pattern-matching language for querying RDF graphs. In what follows we build on the formalisation of the language proposed in [17]; in particular, we consider two-placed OPT and adopt set semantics of queries, leaving three-placed optional and the multiplicities of the answers as defined in the standard for future work. For now we also concentrate on the core fragment and introduce property paths in a separate section.

Formally, let $\mathbf{V}$ be an infinite set $\{?x, ?y, \ldots\}$ of *variables*, disjoint from $\mathbf{T}$. SPARQL *(graph) patterns* are defined recursively as follows:

1. a triple in $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ is a pattern, called *triple pattern*;
2. if P_1 and P_2 are patterns, then $P_1 \mathbin{\mathsf{AND}} P_2$, $P_1 \mathbin{\mathsf{OPT}} P_2$, and $P_1 \mathbin{\mathsf{UNION}} P_2$ are patterns, called AND-, OPT-, and UNION-*patterns*, respectively.

The set of all variables appearing in a pattern P is denoted by $\mathsf{var}(P)$.

In this paper we do not consider FILTER operator, leaving it for future work. It is also known that arbitrary graph patterns (even without FILTER) may have counter-intuitive behaviour and bad computational properties [17]. That is why we concentrate on a restricted class of graph patterns, which is widely used, has expected behaviour and better computational properties [17, 20]—namely, well designed patterns [17, 19]. Formally, a graph pattern P is *well designed* if it is UNION-free and each of its OPT-subpatterns $P_1 \mathbin{\mathsf{OPT}} P_2$ is such that all the variables in $\mathsf{var}(P_2)$ appearing in P outside this subpattern are also in $\mathsf{var}(P_1)$.

The class of well designed patterns, denoted $\mathcal{AO}$-SPARQL, is the main class for this paper. However, we also consider its restrictions and extensions. In particular, the subclass of $\mathcal{AO}$-SPARQL that allows only for AND-subpatterns is denoted $\mathcal{A}$-SPARQL. It corresponds to conjunctive queries without non-distinguished (existential) variables. These classes extend with UNION operator on the top level to $\mathcal{AOU}$-SPARQL and $\mathcal{AU}$-SPARQL: for example, the patterns in the former have the form $P_1 \mathbin{\mathsf{UNION}} \ldots \mathbin{\mathsf{UNION}} P_\ell$ where all the P_i are in $\mathcal{AO}$-SPARQL.

Finally, we also consider the SELECT operator which acts as a result modifier of a graph pattern. In particular, SELECT *queries* are expressions of the form

$$\mathsf{SELECT}\, X \;\mathsf{WHERE}\; P,$$

with P a graph pattern and *distinguished (projection) variables* X a subset of $\mathsf{var}(P)$. A class of SELECT queries with patterns from a class introduced above is denoted by adding $\mathcal{S}$ to the prefix; for example, $\mathcal{AOS}$-SPARQL stands for SELECT queries with well designed patterns. Note that patterns can be seen as queries with all the variables distinguished, so we use "query" as a general term.

SPARQL Semantics. The semantics of graph patterns is defined in terms of *mappings*, that is, partial functions from variables $\mathbf{V}$ to RDF terms $\mathbf{T}$. The *domain* $\mathsf{dom}(\mu)$ of a mapping μ is the set of variables on which μ is defined. Two mappings μ_1 and μ_2 are *compatible* (written as $\mu_1 \sim \mu_2$) if $\mu_1(?x) = \mu_2(?x)$ for all variables $?x$ that are in both $\mathsf{dom}(\mu_1)$ and $\mathsf{dom}(\mu_2)$. If $\mu_1 \sim \mu_2$, then $\mu_1 \cup \mu_2$ denotes the mapping obtained by extending μ_1 according to μ_2 on all the variables in $\mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1)$.

Given two sets of mappings M_1 and M_2, the *join*, *union*, and *difference* of M_1 and M_2 are defined respectively as follows:

$$\begin{aligned} M_1 \bowtie M_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2, \text{ and } \mu_1 \sim \mu_2\}, \\ M_1 \cup M_2 &= \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\}, \\ M_1 \setminus M_2 &= \{\mu_1 \mid \mu_1 \in M_1 \text{ and there is no } \mu_2 \in M_2 \text{ such that } \mu_1 \sim \mu_2\}. \end{aligned}$$

Based on this, the *left outer join* of M_1 and M_2 is defined as

$$M_1 ⟕ M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2).$$

For a triple pattern P and a mapping μ we write $\mu(P)$ for the triple obtained from P by replacing each variable $?x \in \mathsf{dom}(\mu)$ by $\mu(?x)$. The *evaluation* $\llbracket P \rrbracket_G$ of a graph pattern P over a graph G is defined as follows:

1. if P is a triple pattern, then $\llbracket P \rrbracket_G = \{\mu : \mathsf{var}(P) \to \mathbf{T} \mid \mu(P) \in G\}$,
2. if $P = P_1$ AND P_2, then $\llbracket P \rrbracket_G = \llbracket P_1 \rrbracket_G \bowtie \llbracket P_2 \rrbracket_G$,
3. if $P = P_1$ OPT P_2, then $\llbracket P \rrbracket_G = \llbracket P_1 \rrbracket_G ⟕ \llbracket P_2 \rrbracket_G$,
4. if $P = P_1$ UNION P_2, then $\llbracket P \rrbracket_G = \llbracket P_1 \rrbracket_G \cup \llbracket P_2 \rrbracket_G$.

Finally, the *evaluation* $\llbracket Q \rrbracket_G$ of a query Q of the form SELECT X WHERE P is the set of all projections $\mu|_X$ of mappings μ from $\llbracket P \rrbracket_G$ to X, where the *projection* of μ to X is the mapping that coincides with μ on X and undefined elsewhere.

3 Property Paths in SPARQL

Property paths are a new feature introduced in SPARQL 1.1 [10] to allow for navigational querying over RDF graphs. Intuitively, a property path views an RDF document as a labelled graph where the predicate IRI in each triple acts as an edge label. It then extracts each pair of nodes connected by a path such that the word formed by the edge labels along this path belongs to the language of the expression specifying the property path. Property paths resemble regular path queries studied in graph databases [4], but these formalisms have important differences both in syntax and semantics. In this section we define the new SPARQL operator according to the specification and compare the resulting extension with known query languages.

3.1 Property Path Expressions

We start with the definition of property path expressions, following the SPARQL 1.1 specification [10]. We use adopted syntax in spirit of graph database languages, but note that the standard sometimes uses different symbols for operators; for example, inverse paths e^- and alternative paths $e_1 + e_2$ from our definition are denoted there by $\hat{\ }e$ and $e_1 \mid e_2$, respectively.

Definition 1. Property path expressions *are defined by the grammar*

$$e \ := \ a \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^+ \mid e^* \mid e? \mid !\{a_1, \ldots, a_k\} \mid !\{a_1^-, \ldots, a_k^-\},$$

where $a, a_1, \ldots, a_k$ are IRIs in $\mathbf{I}$. Expressions of the last two forms (i.e., starting with !) are called negated property sets.

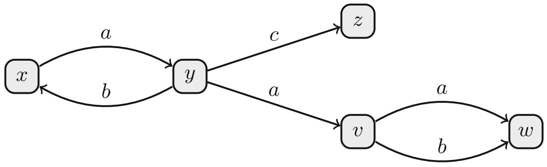


Fig. 1. Example RDF graph G

When dealing with singleton negated property sets brackets may be omitted: for example, $!a$ is a shortcut for $!\{a\}$. Besides the forms in Definition 1 the SPARQL 1.1 specification includes a third version of the negated property sets $!\{a_1, \ldots, a_k, b_1^-, \ldots, b_\ell^-\}$, which allows for negating both normal and inverted IRIs at the same time. We however do not include this extra form in our formalisation, since it is equivalent to the expression $!\{a_1, \ldots, a_k\} + !\{b_1^-, \ldots, b_\ell^-\}$.

The set of all property path expressions is denoted by **PP**. Their normative semantics is given in the following definition.

Definition 2. *The* evaluation $[\![e]\!]_G$ *of a property path expression e over an RDF graph G is a set of pairs of RDF terms from* $\mathbf{T}$ *defined as follows:*

$$\begin{aligned}
[\![a]\!]_G &= \{(s,o) \mid (s,a,o) \in G\},\\
[\![e^-]\!]_G &= \{(s,o) \mid (o,s) \in [\![e]\!]_G\},\\
[\![e_1 \cdot e_2]\!]_G &= [\![e_1]\!]_G \circ [\![e_2]\!]_G,\\
[\![e_1 + e_2]\!]_G &= [\![e_1]\!]_G \cup [\![e_2]\!]_G,\\
[\![e^+]\!]_G &= \textstyle\bigcup_{i\geq 1} [\![e^i]\!]_G,\\
[\![e^*]\!]_G &= [\![e^+]\!]_G \cup \{(a,a) \mid a \textit{ is a term in } G\},\\
[\![e?]\!]_G &= [\![e]\!]_G \cup \{(a,a) \mid a \textit{ is a term in } G\},\\
[\![!\{a_1,\ldots,a_k\}]\!]_G &= \{(s,o) \mid \exists a \textit{ with } (s,a,o) \in G \textit{ and } a \notin \{a_1,\ldots,a_k\}\},\\
[\![!\{a_1^-,\ldots,a_k^-\}]\!]_G &= \{(s,o) \mid (o,s) \in [\![!\{a_1,\ldots,a_k\}]\!]_G\},
\end{aligned}$$

where $\circ$ is the usual composition of binary relations, and e^i is the concatenation $e \cdot \ldots \cdot e$ of i copies of e.

Intuitively, two IRIs are connected by a negated property set if they are subject and object of a triple in the graph whose predicate is not mentioned in the set under negation. Note that, according to Definition 2, the expression $!\{a_1^-, \ldots, a_k^-\}$ retrieves the inverse of $!\{a_1, \ldots, a_k\}$, and thus it respects the direction: a negated inverted IRI returns all pairs of nodes connected by some other inverted IRI. To exemplify, consider the RDF graph G from Figure 1. We have that $[\![!a]\!]_G = \{(y,x),(y,z),(v,w)\}$ as we can find a forward looking predicate different from a for any of these pairs. Note that there is an a-labelled edge between v and w, but since there is also a b-labelled one, the pair (v,w) is in the answer. On the other hand, $[\![!a^-]\!]_G = \{(x,y),(z,y),(w,v)\}$, because we can traverse a backward looking predicate (either b^- or c^-) between these pairs.

Note that $!\{a_1, \ldots, a_k\}$ is not equivalent to $!a_1 + \ldots + !a_k$. To see this consider again the graph G from Figure 1. We have $[\![!a]\!]_G = \{(y,x),(y,z),(v,w)\}$ and $[\![!b]\!]_G = \{(x,y),(y,z),(y,v),(v,w)\}$, while $[\![!\{a,b\}]\!]_G = \{(y,z)\}$.

Property path expressions resemble navigational query languages for graph databases. Indeed, syntactically, property paths without negated property sets are nothing more than the well studied *2-way regular path queries* (*2RPQs*) [5], the default core navigational language for graph databases, with the only minor exception that the empty 2RPQ ε is not expressible as a property path expression (see [4] for a good survey on graph database query languages). However, negated property sets are a unique feature which has not been properly studied before in the SPARQL literature, as far as we are aware (safe [24], where nSPARQL$^{\neg}$ language is introduced, which provides much more expressive navigational facilities than property paths, but no evaluation or optimisation bounds are given, and [1,2], where PSPARQL is studied, whose navigational operator is incomparable with property paths). Note that if we were working with graph databases, where predicates come from a finite alphabet Σ, then one could easily replace $!a$ with a disjunction of all other symbols in Σ. But since we are dealing with RDF graphs, which have predicates from the infinite set of IRIs $\mathbf{I}$, we cannot treat this feature in such a naive way. Nevertheless, we can still show that deciding whether a pair of IRIs belongs to the evaluation of a property path expression e over an RDF graph G is as easy as computing the answers of 2RPQs—the problem is in low polynomial time. The idea of the algorithm is in the same spirit as the ideas of standard algorithms for evaluation of 2RPQs [4,7] and their extensions [1,2,18]: we construct from G and e two nondeterministic finite automata A_e and A_G of special type that can account for negated property sets, and then check that the cross product of these two automata is nonempty.

Proposition 1. *For every property path e and RDF graph G the problem of deciding whether a pair (a, b) of terms belongs to $[\![e]\!]_G$ can be solved in time $O(|G| \cdot |e|)$.*

3.2 Queries with Property Paths

SPARQL 1.1 incorporates property path expressions on the atomic level by means of triples with RDF terms or variables on the subject and object positions, but property path expressions on the predicate position. Formally, we have the following definition.

Definition 3. *A* property path pattern *is a triple in* $(\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times \mathbf{PP} \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$.

Note, however, that property path patterns are incomparable with triple patterns, because they allow for property path expressions in predicate positions, but forbid variables in these positions. We use the notion of *atomic patterns* as a general term for triple and property path patterns.

The classes of queries introduced in Section 2 incorporate navigational functionality by allowing arbitrary atomic patterns as graph patterns, along with complex operator patterns. In our notation this is reflected by letter $\mathcal{P}$ in names of the classes. For example $\mathcal{AOUSP}$-SPARQL is the maximal language considered in this paper, which allows for AND, OPT, UNION, SELECT operators and arbitrary atomic patterns. Remember, however, that all the patterns we consider are (unions of) well designed patterns, assuming that for fragments with property paths this notion stays exactly the same as in Section 2.

To complete the formalization of SPARQL with property paths we need to define the semantics.

Definition 4. *For a property path pattern $P = (u, e, v)$ and an RDF graph G the evaluation $[\![P]\!]_G$ of P over G is the set of mappings*

$$\{\mu : \mathsf{var}(P) \to \mathbf{T} \mid (\mu(u), \mu(v)) \in [\![e]\!]_G\},$$

assuming that mappings μ extends to terms t from $\mathbf{T}$ as identity, that is, $\mu(t) = t$.

Having this definition at hand, the semantics of graph patterns and queries with property paths is exactly the same as in Section 2.

Since property paths resemble 2RPQs, SPARQL with property paths has a lot in common with other graph database languages, such as *conjunctive 2RPQs* (*C2RPQs*), which extend 2RPQs with conjunction and existential quantification, and *unions of C2RPQs* (*UC2RPQs*), further extending 2RPQs with union on the top level (see again [4]). However, there are some important differences.

First, SPARQL with property paths allows for both property path patterns and triple patterns, which may have a variable in the middle position. This is not possible in (U)C2RPQs.

Second, the UNION operator in SPARQL behaves differently from union in classical databases and UC2RPQs. In particular, it is *not null-rejecting*, that is, the patterns constituting a union may have different sets of variables, and, hence, the mappings in the evaluation may have different domains, even if the query is OPT-free.

The third and most important difference is the presence of optional matching in SPARQL. This unique SPARQL feature requires complete rethinking of many standard results in database theory, and, as we will see, results on property paths are not an exception.

In the rest of the paper we study properties of the SPARQL classes with property paths. It is convenient to start in the next section with classes without OPT and then continue in Section 5 with the ones incorporating this operator.

4 Properties of Classes without Optional Matching

The fundamental properties of query languages considered in this paper are complexity of query answering and optimisation problems, such as containment and subsumption. We begin the study of these properties with OPT-free classes of SPARQL with property paths.

4.1 Query Evaluation

We start with the most important fundamental problem for query languages—query evaluation. According to [23], this problem is formalised for any class $\mathcal{X}$-SPARQL defined in the previous sections as follows.

Evaluation($\mathcal{X}$-SPARQL)
Input: An RDF graph G, $\mathcal{X}$-SPARQL query Q, and mapping μ.
Question: Does μ belong to $[\![Q]\!]_G$?

As discussed above, the class $\mathcal{AUS}$-SPARQL without optional matching and property paths is just the class of unions of conjunctive queries, for which the evaluation problem is well known to be NP-complete. Without selection, that is, without non-distinguished variables, it is in PTIME. Based upon Proposition 1, we can show that adding property paths to OPT-free SPARQL does not affect the complexity of query evaluation, same as adding 2RPQs to conjunctive queries.

Proposition 2. *The following holds:*

- Evaluation($\mathcal{X}$-SPARQL) *is* NP*-complete for* $\mathcal{X} \in \{\mathcal{ASP}, \mathcal{AUSP}\}$*;*
- Evaluation($\mathcal{AUP}$-SPARQL) *is in* PTIME*.*

4.2 Query Containment

In this section we consider query containment for OPT-free SPARQL with property paths. This is one of the fundamental problems for static analysis of query languages [23], which asks whether all the answers of one query are among answers of another for any input RDF graph.

Formally, a query Q_1 is *contained* in a query Q_2, denoted by $Q_1 \subseteq Q_2$, if for every RDF graph G we have $[\![Q_1]\!]_G \subseteq [\![Q_2]\!]_G$. Then, the corresponding decision problem is defined as follows for classes of queries $\mathcal{X}_1$-SPARQL and $\mathcal{X}_2$-SPARQL.

Containment($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL)
Input: Queries Q_1 from $\mathcal{X}_1$-SPARQL and Q_2 from $\mathcal{X}_2$-SPARQL.
Question: Is $Q_1 \subseteq Q_2$?

It is known that containment of 2RPQs and C2RPQs without projection is PSPACE-complete [6], and EXPSPACE-complete if projection is allowed. Given the resemblance of 2RPQs and property paths, it is natural to ask whether the techniques of [6] and [5] can be reused in the context of SPARQL with property paths. It turns out that, to some extend, this is indeed the case, but the nature of triples in RDF graphs and the presence of negated property sets oblige us to rework most of their definitions, including the key one—“canonical database”, in order to adapt them to the SPARQL scenario. The following examples illustrate the main challenges that arise and ideas how to overcome them.

Example 1. Consider $\mathcal{ASP}$-SPARQL queries

$$\begin{aligned} Q_1 &= \mathsf{SELECT}\ ?x, ?y, ?z\ \mathsf{WHERE}\ (?x, !a, ?y)\ \mathsf{AND}\ (?x, !a, ?z), \\ Q_2 &= \mathsf{SELECT}\ ?x, ?y, ?z\ \mathsf{WHERE}\ (?x, ?v, ?y)\ \mathsf{AND}\ (?x, ?v, ?z). \end{aligned}$$

One can easily check that Q_1 is not contained in Q_2. However, a counterexample for this fact requires a graph in which images of $?x$ and $?y$ are connected by a different property than those of $?x$ and $?z$. It means that we cannot treat $!a$ just as a usual RDF term, but we need to allow each occurrence of a negated property set to be witnessed by a fresh term.

Example 2. Consider now $\mathcal{ASP}$-SPARQL queries

$$Q_3 = \mathsf{SELECT}\ ?x, ?y\ \mathsf{WHERE}\ (?x, ?v, ?y),$$
$$Q_4 = \mathsf{SELECT}\ ?x, ?y\ \mathsf{WHERE}\ (?x, !a, ?y).$$

Again, Q_3 is not contained in Q_4. This time, however, counterexamples are formed by triples of the form (b, a, c), for IRIs b and c. Thus, we cannot just construct a canonical graph by *freezing* every variable in the query on the left of the possible containment, because counterexamples may need to be formed by mapping some of these variables to negated IRIs from the query on the right.

Taking into account these ideas, we can rework the machinery in [5] so that the notion of canonical graphs is adapted to SPARQL queries with full property paths, including the limited negation. Then, using automata techniques, we can prove results similar to [5] for containment of OPT-free SPARQL with property paths—it is EXPSPACE-complete in general and PSPACE-complete if the right-hand side query is a pattern without projection.

Theorem 1. *The following holds:*

- Containment($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL) *is* EXPSPACE-*complete for* $\mathcal{X}_1 \in \{\mathcal{AP}, \dots, \mathcal{AUSP}\}$ *and* $\mathcal{X}_2 \in \{\mathcal{ASP}, \mathcal{AUSP}\}$*;*
- Containment($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL) *is* PSPACE-*complete for* $\mathcal{X}_1 \in \{\mathcal{AP}, \dots, \mathcal{AUSP}\}$ *and* $\mathcal{X}_2 \in \{\mathcal{AP}, \mathcal{AUP}\}$*.*

To conclude, we note that in the first case the space used depends exponentially only on the size of each of the union-free subpatterns and not on the number of these subpatterns. This property is crucial for the results of Section 5.3 (in particular, Theorem 3).

4.3 Query Subsumption

Query containment is a way of specifying that one query is more general than another, which is common across different query formalisms. However, the unique SPARQL feature is the ability to return partial answers, and Pérez et al. argued in [17] that it is more natural to compare SPARQL queries for subsumption, that is, to check whether for any answer to one query there is a more elaborate answer to the other one on any input RDF graph.

Formally, a mapping μ is *subsumed* by a mapping μ', denoted by $\mu \sqsubseteq \mu'$, if $\mathsf{dom}(\mu)$ is contained in $\mathsf{dom}(\mu')$ and $\mu \sim \mu'$. A query Q_1 is *subsumed* by a query Q_2 (written as $Q_1 \sqsubseteq Q_2$) if for every RDF graph G it holds that for each $\mu_1 \in [\![Q_1]\!]_G$ there exists $\mu_2 \in [\![Q_2]\!]_G$ such that $\mu_1 \sqsubseteq \mu_2$. The corresponding decision problem for classes of queries $\mathcal{X}_1$-SPARQL and $\mathcal{X}_2$-SPARQL is defined as follows.

Subsumption($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL)
Input: Queries Q_1 from $\mathcal{X}_1$-SPARQL and Q_2 from $\mathcal{X}_2$-SPARQL.
Question: Is $Q_1 \sqsubseteq Q_2$?

Although the notion of subsumption becomes more natural when dealing with the OPT operator, for completion we still study this problem for the case of OPT-free SPARQL queries with property paths. We also find it interesting that the complexity of subsumption ends up being higher than the complexity of containment for some of the classes.

Before stating the results on subsumption, we give some intuition behind them and compare subsumption with containment. For a query Q_1 from $\mathcal{ASP}$-SPARQL to be subsumed by a query Q_2 from the same class it is necessary that the set of distinguished variables of Q_1 is a subset of the distinguished variables of Q_2. Moreover, $Q_1 \sqsubseteq Q_2$ if and only if for every RDF graph G and mapping μ_1 in $[\![Q_1]\!]_G$ one can obtain μ_1 from some μ_2 in $[\![Q_2]\!]_G$ by projecting out the distinguished variables of Q_2 that are not distinguished in Q_1. The first obvious consequence of this observation is that in this case the subsumption problem for $\mathcal{ASP}$-SPARQL is not more difficult than the containment problem, because $Q_1 \sqsubseteq Q_2$ if and only if Q_1 is contained in $\mathsf{SELECT}\ X\ \mathsf{WHERE}\ P_2$, with X the set of output variables of Q_1 and P_2 the pattern of Q_2. However, rather surprisingly, the limited projection inherent to the subsumption problem is enough to make the problem $\mathsf{EXPSPACE}$-hard even for patterns from $\mathcal{AP}$-SPARQL, which do not have non distinguished variables.

Proposition 3. *The problem* Subsumption($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL) *is* $\mathsf{EXPSPACE}$*-complete for* $\mathcal{X}_1, \mathcal{X}_2 \in \{\mathcal{AP}, \ldots, \mathcal{AUSP}\}$*.*

5 Properties of Classes with Optional Matching

In this section we consider query evaluation, containment, and subsumption for SPARQL classes that allow for both the OPT operator and property paths. In addition to the difficulties from the previous section, such as negated property sets and non-null-rejecting union, we have to deal with mixture of optional matching and property paths. To overcome these difficulties we develop non-trivial compositions of the usual SPARQL and graph databases techniques, as well as invent new ones.

5.1 Query Evaluation

Complexity bounds for query evaluation of SPARQL classes with (well designed) optional matching that do not use property paths are by now well understood [12,17]. In particular, the problem is coNP-complete for graph patterns, that is, queries with all the variables distinguished, and jumps to Σ_2^p if arbitrary SELECT clauses are allowed. In this section we show that adding property paths to the set of allowed operators preserves these bounds. To this end, we develop a characterisation similar to the one in [12], by adapting the notions of OPT normal form and pattern trees to work with property path patterns.

A graph pattern P is in OPT *normal form* if no OPT operators appear in AND-subpatterns of P. It was shown in [17, Proposition4.11] that every well designed graph pattern without property path patterns can be transformed to an equivalent pattern in OPT normal form in polynomial time by means of a set of rewriting rules that “push” AND inside OPT (recall that well designed patterns have neither UNION nor SELECT clauses). It is straightforward to check that these rules are correctly applicable to graph patterns that allow for property paths, so in what follows we assume that all patterns are in OPT normal form (in particular, AND-patterns are just AND combinations of atomic patterns).

Each graph pattern P in OPT normal form can be intuitively represented as a *pattern tree* $\mathit{Tree}(P)$, that is, a rooted tree with nodes labelled by sets of atomic (i.e., triple and property path) patterns which is recursively constructed as follows:

- if P is an AND-pattern then $\mathit{Tree}(P)$ consists of a single node labelled with the set of all atomic patterns in this AND-pattern;
- if $P = P_1 \;\mathsf{OPT}\; P_2$ then $\mathit{Tree}(P)$ is obtained from $\mathit{Tree}(P_1)$ and $\mathit{Tree}(P_2)$ by adding an edge form the root of the former to the root of the latter.

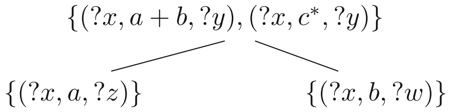
In other words, the labels of nodes in pattern trees correspond to conjunctions of atomic patterns, while edges represent the structure of optional matching. For a node n in a pattern tree, $\mathsf{and}(n)$ denotes the AND pattern consisting of the atomic patterns in its label, and $\mathsf{var}(n)$ denotes the set of all variables in these patterns; these notations propagate to sets of nodes and subtrees of pattern trees. In fact, we are interested only in subtrees containing the root of the original tree, so in what follows we assume this restriction without mentioning it explicitly. A node in a pattern tree is a *child* of a subtree T if it is not in T but its parent is.

It is important to note that pattern trees are unordered, so different patterns may have the same representation. However, we disregard this syntactical mismatch, because such patterns are always equivalent. This follows from the fact that for well designed patterns $((P_1 \;\mathsf{OPT}\; P_2) \;\mathsf{OPT}\; P_3)$ is equivalent to $((P_1 \;\mathsf{OPT}\; P_3) \;\mathsf{OPT}\; P_2)$ (this was stated in [16] and proved in [12] for patterns without property paths, but a generalisation to our case is straightforward).

Example 3. Consider the pattern

$$((((?x, a+b, ?y) \;\mathsf{AND}\; (?x, c^*, ?y)) \;\mathsf{OPT}\; (?x, a, ?z)) \;\mathsf{OPT}\; (?x, b, ?w)).$$

The tree representing this pattern is as follows.



Another interesting property of pattern trees is that for each variable the set of all nodes with this variable in the labels is always connected. This is a vivid illustration of the well-designedness property of patterns. Moreover, every pattern tree $\mathit{Tree}(P)$ (and hence every well-designed pattern) can be normalised to an equivalent tree T' (called *NR normal form* [12]) such that $\mathsf{var}(n') \not\subseteq \mathsf{var}(n)$

for every edge (n, n') in T', that is, such that every node introduces a new variable in comparison to the parent of this node. Transformation to NR normal form can be done in polynomial time by adding the label of every node without new variables to the labels of its children and then removing all such nodes from the tree. In what follows we assume all well designed patterns and corresponding pattern trees to be in NR normal form.

An intuitive coNP algorithm for evaluation of well designed patterns with property paths works in the same way as the one described in [12] for the case without property paths. It consists in the following two steps. Since the input pattern is in NR normal form, the input mapping μ uniquely defines a subtree T'_μ such that $\mathsf{dom}(\mu) = \mathsf{var}(T'_\mu)$. So, on the first step we need to check for the input graph that $\mu(\mathsf{and}(T'_\mu)) \subseteq G$, that is, all the patterns in the subtree under μ indeed materialise in the input graph G. This check can be done in polynomial time, because by Proposition 1 property paths have tractable evaluation. On the second and more difficult step we need to guarantee that μ cannot be consistently extended to the variables of any child of T'_μ in T'. This can be done in coNP by guessing a counterexample (i.e., an extension) for one of these children.

Same as for patterns without property paths, this algorithm can be extended to union and selection in a straightforward way. In the latter case the complexity jumps one level of the polynomial hierarchy, because we have to guess the values of non-distinguished variables. Combining these results with matching lower bounds for the classes without property paths [12,17] we obtain the following proposition.

Proposition 4. *The following holds:*
- Evaluation($\mathcal{X}$-SPARQL) *is* Σ_2^p*-complete for* $\mathcal{X} \in \{\mathcal{AOSP}, \mathcal{AUOSP}\}$*;*
- Evaluation($\mathcal{X}$-SPARQL) *is* coNP*-complete for* $\mathcal{X} \in \{\mathcal{AOP}, \mathcal{AUOP}\}$*.*

The focus of this paper is SPARQL with well designed optional matching, and we leave a comprehensive study of SPARQL with property paths and arbitrary nesting of other operators considered in this paper for future work. However, as a final remark in this section, we note that it is not difficult to show PSPACE-completeness of evaluation for this class, that is, the same complexity as for any subclass of this class that allows for arbitrary optional matching [21].

5.2 Query Containment

Now we move to the containment problem of SPARQL with property paths. As shown in [20], without them the problem Containment($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL) is NP-complete for any $\mathcal{X}_1$-SPARQL that allows for optional matching and for $\mathcal{X}_2$-SPARQL = $\mathcal{AO}$-SPARQL, that is for the class of well designed patterns. If $\mathcal{X}_2$-SPARQL also allows for union, then the complexity becomes Π_2^p-complete (again, for the full range of $\mathcal{X}_1$-SPARQL), and the problem is undecidable if $\mathcal{X}_2$-SPARQL allows for arbitrary selection. Thus we focus on the most general case where we can hope for decidability: checking whether a query in $\mathcal{AOUSP}$-SPARQL is contained in a query in $\mathcal{AOUP}$-SPARQL. Our main result is that this problem is also decidable, specifically, $\mathsf{EXPSPACE}$-complete.

As we saw in the previous subsection, the techniques developed in [12,17] for checking evaluation can be extended to work with property paths with relatively little effort. Later we will see that similar strategy works for subsumption, because it can be reduced to checking containment of OPT-free queries, which is extensible to classes with property paths. However, the situation is different for containment. It is not clear how to apply the known techniques (e.g., the one in [20, Theorem3.7]) to state the problem in terms of containment of OPT-free queries. To overcome this, we develop a new characterization of containment that reduces the problem to a weaker form of containment between OPT-free queries. Then we take advantage of the automata techniques developed in Section 4.

In what follows we first present our new characterisation for containment for queries without property paths (which we believe is of independent interest) and then adapt it to the general case. We start with a definition.

Definition 5. *Let*

$$Q_1 = \mathsf{SELECT}\, X\ \mathsf{WHERE}\ P \quad \textit{and} \quad Q_2 = P^1\ \mathsf{UNION}\ \ldots\ \mathsf{UNION}\ P^k$$

be queries from $\mathcal{AOSP}$-SPARQL and $\mathcal{AOUP}$-SPARQL respectively, with P, P^i well designed patterns. A good extension *E of Q_1 over Q_2 is an* AND *pattern*

$$\mathsf{and}(\mathit{Tree}(P))\ \ \mathsf{AND}\ \ \mathsf{and}(n_1)\ \mathsf{AND} \ldots \mathsf{AND}\ \mathsf{and}(n_m),$$

where $m \leq k$ and every n_j is obtained from a child of a subtree T_j of one of $\mathit{Tree}(P^1), \ldots, \mathit{Tree}(P^k)$ with $\mathsf{var}(T_j) = X$ by renaming all variables not in X to fresh ones. The support *$\mathsf{sup}(E)$ of E is the set of all the subtrees T_j.*

Our new characterisation of containment for the case without property paths is based on the following lemma.

Lemma 1. *Let*

$$Q_1 = \mathsf{SELECT}\, X\ \mathsf{WHERE}\ P \quad \textit{and} \quad Q_2 = P^1\ \ \mathsf{UNION} \ldots \mathsf{UNION}\ \ P^k$$

be a $\mathcal{AOUS}$-SPARQL and $\mathcal{AOU}$-SPARQL queries respectively. Then $Q_1 \not\subseteq Q_2$ if and only if there is a good extension E over Q_2 of some $\mathcal{AOS}$-SPARQL query with a pattern P^* such that $\mathit{Tree}(P^*)$ is a subtree of one of the trees representing components of P and distinguished variables $X^* = X \cap \mathsf{var}(P^*)$ that satisfies the following conditions:

(C1) *for each child n of $\mathit{Tree}(P^*)$, there is no homomorphism h from $\mathsf{and}(n)$ to E such that $h(?x) = ?x$ for all variables $?x$ in $\mathsf{var}(n) \cap \mathsf{var}(E)$, and*

(C2) *for each subtree T of one of $\mathit{Tree}(P^1), \ldots, \mathit{Tree}(P^k)$ with $\mathsf{var}(T) = X^*$ that is not in $\mathsf{sup}(E)$ there is no homomorphism h from $\mathsf{and}(T)$ to E such that $h(?x) = ?x$, for all variables $?x$ in $\mathsf{var}(T) \cap \mathsf{var}(E)$.*

The intuition behind Lemma 1 is as follows. A good extension E satisfying conditions (C1) and (C2) gives us a witness for non-containment: it suffices to consider the "frozen RDF graph" G of E obtained by replacing each variable

$?x$ by a fresh IRI a_x and the mapping μ with $\mu(?x) = a_x$, for all $?x \in X^*$ and undefined for other $?x$. Then conditions (C1) and (C2) are a convenient way of stating that $\mu \in [\![Q_1]\!]_G$ and $\mu \notin [\![Q_2]\!]_G$.

Observe that the size of a good extension is polynomial in the size of Q_1 and Q_2. Thus, Lemma 1 gives us an alternative proof for Π_2^p-membership of containment of a query in a pattern if both of them do not use property paths. Indeed to find a counterexample for containment we need to guess a good extension and then call for a coNP oracle to check conditions (C1) and (C2).

To extend the characterisation of Lemma 1 to queries with property paths we need the following auxiliary notation. We write $P_1 \preceq P_2$ for patterns P_1 and P_2 if for each RDF graph G and mapping $\mu_1 \in [\![P_1]\!]_G$ there is a mapping $\mu_2 \in [\![P_2]\!]_G$ such that $\mu_1 \sim \mu_2$.

We analyse the complexity of containment in the presence of property paths by means of the following generalised statement.

Lemma 2. *Let*

$$Q_1 = \mathsf{SELECT}\, X \ \mathsf{WHERE}\ P \quad \textit{and} \quad Q_2 = P^1\ \mathsf{UNION} \ldots \mathsf{UNION}\ \ P^k$$

be a $\mathcal{AOUSP}$-SPARQL and $\mathcal{AOUP}$-SPARQL queries respectively. Then $Q_1 \not\sqsubseteq Q_2$ if and only if there is a good extension E over Q_2 of some $\mathcal{AOSP}$-SPARQL query with a pattern P^* such that Tree(P^*) is a subtree of one of the trees representing components of P and distinguished variables $X^* = X \cap \mathsf{var}(P^*)$ that satisfies $E \not\preceq (N\ \mathsf{UNION}\ S)$, where

(C1′) *N is a union of all $\mathsf{and}(n)$ for children n of Tree(P^*), and*

(C2′) *S is a union of all $\mathsf{and}(T)$ for subtrees T of trees Tree$(P^1), \ldots,$ Tree(P^k) with $\mathsf{var}(T) = X^*$ that are not in $\mathsf{sup}(E)$.*

Using techniques developed in Section 4.2, the condition $E \not\preceq (N\ \mathsf{UNION}\ S)$ can be checked in $\mathsf{EXPSPACE}$. This gives us an $\mathsf{EXPSPACE}$ upper bound for containment of $\mathcal{AOUSP}$-SPARQL and $\mathcal{AOUP}$-SPARQL queries. Moreover, the matching lower bound can be derived from Proposition 3.

Theorem 2. *The problem* Containment($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL) *is* $\mathsf{EXPSPACE}$*-complete for $\mathcal{X}_1 \in \{\mathcal{AOP}, \ldots, \mathcal{AOUSP}\}$ and $\mathcal{X}_2 \in \{\mathcal{AOP}, \mathcal{AOUP}\}$.*

5.3 Query Subsumption

The last problem we study in this paper is subsumption of SPARQL queries with property paths. Letelier et al. [12,20] proved Π_2^p-completeness of this problem for all the classes with optional matching but without property paths, even if arbitrary selection is allowed. Moreover, they provide the following very simple and useful characterisation for the subsumption of $\mathcal{AO}$-SPARQL patterns: a pattern P_1 is subsumed by a pattern P_2 if and only if for every subtree T_1' of *Tree*(P_1) there is a subtree T_2' of *Tree*(P_2) such that $\mathsf{var}(T_1') \subseteq \mathsf{var}(T_2')$ and there is a homomorphism from $\mathsf{and}(T_2')$ to $\mathsf{and}(T_1')$ that is the identity on $\mathsf{var}(T_1')$. This idea extends to patterns with union in the usual way—the subsumption holds if and only if for every component of the first pattern there is a subsuming one in the second.

How can this characterisation be extended to deal with property paths? The immediate idea is just to replace homomorphism with containment of corresponding OPT-free queries. However, in the presence of union this simple strategy does not always work. Indeed, the pattern $(?x, (a+b), ?y)$ is subsumed by the pattern $(?x, a, ?y)$ UNION $(?x, b, ?y)$ (in fact, they are equivalent), but not in any of its components.

As we see, the problem is the disjunction introduced by property paths, and our characterisation needs to account for this. By doing so we arrive at the following characterisation. A pattern P_1 is subsumed by a pattern P_2 if and only if for every subtree T_1' of $\mathit{Tree}(P_1)$ the AND-pattern $\mathsf{and}(T_1')$ is subsumed in the union of all AND-patterns $\mathsf{and}(T_2')$, where T_2' ranges over subtrees of $\mathit{Tree}(P_2)$ with $\mathsf{var}(T_1') \subseteq \mathsf{var}(T_2')$. With this characterisation we avoid dealing with optional matching, and can thus solve subsumption by the techniques introduced in the previous section. As an illustration, we can use this characterisation in the example above to show that $Q_1 \sqsubseteq Q_2$, by choosing the same query $(?x, a, ?y)$ UNION $(?x, b, ?y)$. By extending this characterisation for all queries with arbitrary selection we obtain our last theorem.

Theorem 3. *The problem* SUBSUMPTION($\mathcal{X}_1$-SPARQL,$\mathcal{X}_2$-SPARQL) *is* EXPSPACE-*complete for* $\mathcal{X}_1, \mathcal{X}_2 \in \{\mathcal{AOP}, \ldots, \mathcal{AOUSP}\}$.

6 Conclusions

At a first glance it was not clear whether one could combine techniques from graph databases and the Semantic Web to study SPARQL with property paths. Indeed, on the one hand, graph database techniques failed short for such study, because RDF data allows for predicates from an infinite alphabet and property paths may have negation. On the other hand, even if the machinery developed to study SPARQL without property paths proved to be inspirational for this work, the characterisations provided in the literature were too specific to be used in the general case. In this paper we have shown how these two classes of techniques can be generalised and combined to reason about SPARQL queries that allow for property path patterns. In particular, we developed algorithms for evaluating such queries and deciding their containment and subsumption. Finally we would like to note that many of the upped bounds obtained here (e.g., all EXPSPACE and Π_2^p bounds) match the lower bounds for more restricted classes of queries.

As for future work, the main direction we would like to tackle is the addition of the FILTER operator to the language, since so far this feature of SPARQL has not been comprehensively considered in the literature. We have some preliminary results showing that the techniques from Section 5.2 can be extended to work in this setting. Another interesting direction is to study the fragments with property paths and full power of optional matching, that is, that allow for three-placed and not well designed OPTIONAL.

Acknowledgments. Reutter, Romero and Vrgoč were supported by the Millennium Nucleus Center for Semantic Web Research Grant NC120004.

References

1. Alkhateeb, F.: Querying RDF(S) with regular expressions. Ph.D. thesis, Université Joseph Fourier, Grenoble (2008)
2. Alkhateeb, F., Baget, J.F., Euzenat, J.: Extending SPARQL with regular expression patterns (for querying RDF). J. Web Sem. **7**(2), 57–73 (2009)
3. 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 2012, pp. 629–638 (2012)
4. Barceló Baeza, P.: Querying graph databases. In: PODS 2013, pp. 175–188 (2013)
5. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Containment of conjunctive regular path queries with inverse. In: KR 2000, pp. 176–185 (2000)
6. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Reasoning on regular path queries. ACM SIGMOD Record **32**(4), 83–92 (2003)
7. Consens, M., Mendelzon, A.: GraphLog: a visual formalism for real life recursion. In: PODS 1990, pp. 404–416 (1990)
8. Chekol, M.W.: Static Analysis of Semantic Web Queries. Ph.D. thesis, Université de Grenoble (2012)
9. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: SPARQL Query Containment under RDFS Entailment Regime. In: IJCAR 2012 (2012)
10. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query
11. Kostylev, E.V., Reutter, J.L., Vrgoč, D.: Containment of Data Graph Queries. In: ICDT 2014, pp. 131–142 (2014)
12. Letelier, A., Pérez, J., Pichler, R., Skritek, S.: Static analysis and optimization of semantic web queries. ACM TODS 38(4) (2013)
13. Libkin, L., Reutter, J.L., Vrgoč, D.: Trial for RDF: adapting graph query languages for RDF data. In: PODS 2013, pp. 201–212 (2013)
14. Losemann, K., Martens, W.: The Complexity of Regular Expressions and Property Paths in SPARQL. ACM TODS 38(4) (2013)
15. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM TOCL **5**(3), 403–435 (2004)
16. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: ISWC 2006, pp. 30–43 (2006)
17. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM TODS 34(3) (2009)
18. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. J. Web Sem. **8**(4), 255–270 (2010)
19. Picalausa, F., Vansummeren, S.: What are real SPARQL queries like? In: SWIM 2011 (2011)
20. Pichler, R., Skritek, S.: Containment and equivalence of well-designed SPARQL. In: PODS 2014, pp. 39–50 (2014)
21. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: ICDT 2010, pp. 4–33 (2010)
22. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (January 15, 2008). http://www.w3.org/TR/rdf-sparql-query/
23. Vardi, M.Y.: The Complexity of Relational Query Languages. In: STOC (1982)
24. Zhang, X., Van den Bussche, J.: On the Power of SPARQL in Expressing Navigational Queries. The Computer Journal (2014)