Journal of Artificial Intelligence Research 56 (2016) 403–428 Submitted 11/15; published 07/16

On the Satisfiability Problem for SPARQL Patterns

Xiaowang Zhang XIAOWANGZHANG@TJU.EDU.CN
School of Computer Science and Technology,
Tianjin University, China
Tianjin Key Laboratory of
Cognitive Computing and Application,
Tianjin, China

Jan Van den Bussche JAN.VANDENBUSSCHE@UHASSELT.BE
Hasselt University, Belgium

François Picalausa FPICALAUSA@GMAIL.COM

Abstract

The satisfiability problem for SPARQL 1.0 patterns is undecidable in general, since the relational algebra can be emulated using such patterns. The goal of this paper is to delineate the boundary of decidability of satisfiability in terms of the constraints allowed in filter conditions. The classes of constraints considered are bound-constraints, negated bound-constraints, equalities, nonequalities, constant-equalities, and constant-nonequalities. The main result of the paper can be summarized by saying that, as soon as inconsistent filter conditions can be formed, satisfiability is undecidable. The key insight in each case is to find a way to emulate the set difference operation. Undecidability can then be obtained from a known undecidability result for the algebra of binary relations with union, composition, and set difference. When no inconsistent filter conditions can be formed, satisfiability is decidable by syntactic checks on bound variables and on the use of literals. Although the problem is shown to be NP-complete, it is experimentally shown that the checks can be implemented efficiently in practice. The paper also points out that satisfiability for the so-called 'well-designed' patterns can be decided by a check on bound variables and a check for inconsistent filter conditions.

1. Introduction

The Resource Description Framework is a popular data model for information on the Web. RDF represents information in the form of directed, labeled graphs. The standard query language for RDF data is SPARQL (Harris & Seaborne, 2013). The current version 1.1 of SPARQL extends SPARQL 1.0 (Prud'hommeaux & Seaborne, 2008) with important features such as aggregation and regular path expressions (Arenas, Conca, & Pérez, 2012). Other features, such as negation and subqueries, have also been added, but mainly for efficiency reasons, as they were already expressible, in a more involved manner, in version 1.0. Hence, it is still relevant to study the fundamental properties of SPARQL 1.0. In this paper, we follow the elegant formalization of SPARQL 1.0 by Arenas, Gutierrez, & Pérez (2009) which is eminently suited for theoretical investigations.

The fundamental problem that we investigate is that of *satisfiability* of SPARQL patterns. A pattern is called satisfiable if there exists an RDF graph under which the pattern evaluates to a nonempty set of mappings. For any query language, satisfiability is clearly one of the essential properties one needs to understand if one wants to do automated reasoning. Since SPARQL patterns can emulate relational algebra expressions (Angles & Gutierrez, 2008; Polleres, 2007; Arenas & Pérez, 2011), and satisfiability for relational algebra is undecidable (Abiteboul, Hull, & Vianu, 1995), the general satisfiability problem for SPARQL is undecidable as well.

Whether or not a pattern is satisfiable depends mainly on the filter operations appearing in the pattern; without filter operations, a pattern is always satisfiable except for trivial cases where a literal occurs in the wrong place. The goal of this paper is to precisely delineate the decidability of SPARQL fragments that are defined in terms of the constraints that can be used as filter conditions. The six basic classes of constraints we consider are bound-constraints; equalities; constant-equalities; and their negations. In this way, fragments of SPARQL can be constructed by specifying which kinds of constraints are allowed as filter conditions. For example, in the fragment SPARQL$(\text{bound}, \neq, \neq_c)$, filter conditions can only be bound constraints, nonequalities, and constant-nonequalities.

Our main result states that the only fragments for which satisfiability is decidable are the two fragments SPARQL$(\text{bound}, =, \neq_c)$ and SPARQL$(\text{bound}, \neq, \neq_c)$ and their subfragments. Consequently, as soon as either negated bound-constraints, or constant-equalities, or combinations of equalities and nonequalities are allowed, the satisfiability problem becomes undecidable. Each undecidable case is established by showing how the set difference operation can be emulated. This was already known using negated bound-constraints (Angles & Gutierrez, 2008; Arenas & Pérez, 2011); so we show it is also possible using constant-equalities, and using combinations of equalities and nonequalities, but in no other way. Undecidability can then be obtained from a known undecidability result for the algebra of binary relations with union, composition, and set difference (Tan, Van den Bussche, & Zhang, 2014).

In the decidable cases, satisfiability can be decided by syntactic checks on bound variables and the use of literals. Although the problem is shown to be NP-complete, it is experimentally shown that the checks can be implemented efficiently in practice.

At the end of the paper we look at a well-behaved class of patterns known as the 'well-designed' patterns (Pérez et al., 2009). We observe that satisfiability of well-designed patterns can be decided by combining the check on bound variables with a check for inconsistent filter conditions.

This paper is further organized as follows. In the next section, we introduce syntax and semantics of SPARQL patterns and introduce the different fragments under consideration. Section 3 introduces the satisfiability problem and shows satisfiability checking for the fragments SPARQL$(\text{bound}, =, \neq_c)$ and SPARQL$(\text{bound}, \neq, \neq_c)$. Section 4 shows undecidability for the fragments SPARQL$(\neg\text{bound})$, SPARQL$(=_c)$, and SPARQL$(=, \neq)$. Section 5 considers well-designed patterns.

Section 6 reports on experiments that test our decision methods in practice. In Section 7 we briefly discuss how our results extend to the new operators that have been added to SPARQL 1.1. We conclude in Section 8.

2. SPARQL and Fragments

In this section we recall the syntax and semantics of SPARQL patterns, closely following the core SPARQL formalization given by Arenas, Gutierrez, & Pérez (2009).[1] The semantics we use is set-based, whereas the semantics of real SPARQL is bag-based. However, for satisfiability (the main topic of this paper), it makes no difference whether we use a set or bag semantics (Schmidt, Meier, & Lausen, 2010, Lemma 1).

In this section we will also define the language fragments defined in terms of allowed filter conditions, which will form the object of this paper.

2.1 RDF Graphs

Let I, B, and L be infinite sets of *IRIs*, *blank nodes* and *literals*, respectively. These three sets are pairwise disjoint. We denote the union $I \cup B \cup L$ by U, and elements of $I \cup L$ will be referred to as *constants*. Note that blank nodes are not constants.

A triple $(s, p, o) \in (I \cup B) \times I \times U$ is called an *RDF triple*. An *RDF graph* is a finite set of RDF triples.

2.2 Syntax of SPARQL Patterns

Assume furthermore an infinite set V of *variables*, disjoint from U. The convention in SPARQL is that variables are written beginning with a question mark, to distinguish them from constants. We will follow this convention in this paper.

SPARQL *patterns* are inductively defined as follows.

- Any triple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a pattern (called a *triple pattern*).
- If P_1 and P_2 are patterns, then so are the following:
  - P_1 UNION P_2;
  - P_1 AND P_2;
  - P_1 OPT P_2.
- If P is a pattern and C is a constraint (defined next), then P FILTER C is a pattern; we call C the *filter condition*.

  Here, a *constraint* can have one of the six following forms:

  1. *bound-constraint:* bound($?x$)
  2. *negated bound-constraint:* $\neg$bound($?x$)
  3. *equality:* $?x = ?y$
  4. *nonequality:* $?x \neq ?y$ with $?x$ and $?y$ distinct variables
  5. *constant-equality:* $?x = c$ with c a constant
  6. *constant-nonequality:* $?x \neq c$

[1] Arenas, Pérez, and Guttierez (2009) discuss minor deviations between the formalization and real SPARQL, and why these differences are inessential for the purpose of formal investigation.

We do not need to consider conjunctions and disjunctions in filter conditions, since conjunctions can be expressed by repeated application of filter, and disjunctions can be expressed using UNION. Hence, by going to disjunctive normal form, any predicate built using negation, conjunction, and disjunction is indirectly supported by our language.

Moreover, real SPARQL also allows blank nodes in triple patterns. This feature has been omitted from the formalization because blank nodes in triple patterns can be equivalently replaced by variables.

2.3 Semantics of SPARQL Patterns

The semantics of patterns is defined in terms of sets of so-called *solution mappings*, hereinafter simply called *mappings*. A solution mapping is a total function $\mu : S \to U$ on some finite set S of variables. We denote the domain S of μ by $\mathrm{dom}(\mu)$.

We make use of the following convention.

Convention. *For any mapping μ and any constant $c \in I \cup L$, we agree that $\mu(c)$ equals c itself.*

In other words, mappings are by default extended to constants according to the identity mapping.

Now given a graph G and a pattern P, we define the semantics of P on G, denoted by $[\![P]\!]_G$, as a set of mappings, in the following manner.

- If P is a triple pattern (u, v, w), then

$$[\![P]\!]_G := \{\mu : \{u, v, w\} \cap V \to U \mid (\mu(u), \mu(v), \mu(w)) \in G\}.$$

  This definition relies on Convention 2.3 formulated above.

- If P is of the form P_1 UNION P_2, then

$$[\![P]\!]_G := [\![P_1]\!]_G \cup [\![P_2]\!]_G.$$

- If P is of the form P_1 AND P_2, then

$$[\![P]\!]_G := [\![P_1]\!]_G \bowtie [\![P_2]\!]_G,$$

  where, for any two sets of mappings Ω_1 and Ω_2, we define

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}.$$

  Here, two mappings μ_1 and μ_2 are called *compatible*, denoted by $\mu_1 \sim \mu_2$, if they agree on the intersection of their domains, i.e., if for every variable $?x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$, we have $\mu_1(?x) = \mu_2(?x)$. Note that when μ_1 and μ_2 are compatible, their union $\mu_1 \cup \mu_2$ is a well-defined mapping; this property is used in the formal definition above.

- If P is of the form P_1 OPT P_2, then

$$[\![P]\!]_G := ([\![P_1]\!]_G \bowtie [\![P_2]\!]_G) \cup ([\![P_1]\!]_G \smallsetminus [\![P_2]\!]_G),$$

  where, for any two sets of mappings Ω_1 and Ω_2, we define

$$\Omega_1 \smallsetminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \neg\exists \mu_2 \in \Omega_2 : \mu_1 \sim \mu_2\}.$$

- Finally, if P is of the form P_1 FILTER C, then

  $$[\![P]\!]_G := \{\mu \in [\![P_1]\!]_G \mid \mu \models C\}$$

  where the satisfaction of a constraint C by a mapping μ, denoted by $\mu \models C$, is defined as follows:

  1. $\mu \models \text{bound}(?x)$ if $?x \in \text{dom}(\mu)$;
  2. $\mu \models \neg\text{bound}(?x)$ if $?x \notin \text{dom}(\mu)$;
  3. $\mu \models ?x = ?y$ if $?x, ?y \in \text{dom}(\mu)$ and $\mu(?x) = \mu(?y)$;
  4. $\mu \models ?x \neq ?y$ if $?x, ?y \in \text{dom}(\mu)$ and $\mu(?x) \neq \mu(?y)$;
  5. $\mu \models ?x = c$ if $?x \in \text{dom}(\mu)$ and $\mu(?x) = c$;
  6. $\mu \models ?x \neq c$ if $?x \in \text{dom}(\mu)$ and $\mu(?x) \neq c$.

Note that $\mu \models ?x \neq ?y$ is not the same as $\mu \not\models ?x = ?y$, and similarly for $\mu \models ?x \neq c$. This is in line with the three-valued logic semantics for filter conditions used in the official semantics (Arenas et al., 2009). For example, if $?x \notin \text{dom}(\mu)$, then in three-valued logic $?x = c$ evaluates to *error* under μ; consequently, also $\neg ?x = c$ evaluates to *error* under μ. Accordingly, in the semantics above, we have both $\mu \not\models ?x = c$ and $\mu \not\models ?x \neq c$.

2.4 SPARQL Fragments

We can form fragments of SPARQL by specifying which of the six classes of constraints are allowed as filter conditions. We denote the class of bound-constraints by 'bound', negated bound-constraints by '¬bound', equalities by '=', nonequalities by '≠', constant-equalities by '$=_c$', and constant-nonequalities by '$\neq_c$'. Then for any subset F of $\{\text{bound}, \neg\text{bound}, =, \neq, =_c, \neq_c\}$ we can form the fragment SPARQL(F). For example, in SPARQL(bound, $=$, $\neq_c$), filter conditions can only be bound constraints, equalities, and constant-nonequalities.

3. Satisfiability: Decidable Fragments

A pattern P is called *satisfiable* if there exists a graph G such that $[\![P]\!]_G$ is nonempty. In general, checking satisfiability is a very complicated, indeed undecidable, problem. But for the two fragments SPARQL(bound, $=$, $\neq_c$) and SPARQL(bound, $\neq$, $\neq_c$), it will turn out that there are essentially only two possible reasons for unsatisfiability.

The first possible reason is that the pattern specifies a literal value in the first position of some RDF triple, whereas RDF triples can only have literals in the third position. For example, using the literal 42, the triple pattern $(42, ?x, ?y)$ is unsatisfiable. Note that literals in the middle position of a triple pattern are already disallowed by the definition of triple pattern, so we only need to worry about the first position.

This discrepancy between triple patterns and RDF triples is easy to sidestep, however. In the Appendix we show how, without loss of generality, we may assume from now on that *patterns do not contain any triple pattern* (u, v, w) *where* u *is a literal.*

The second and main possible reason for unsatisfiability is that filter conditions require variables to be bound together in a way that cannot be satisfied by the subpattern to which the filter applies. For example, the pattern

$$((?x, a, ?y) \text{ UNION } (?x, b, ?z)) \text{ FILTER } (\text{bound}(?y) \wedge \text{bound}(?z))$$

is unsatisfiable. Note that bound constraints are not strictly necessary to illustrate this phenomenon: if in the above example we replace the filter condition by $?y = ?z$ the resulting pattern is still unsatisfiable.

We next prove formally that satisfiability for patterns in SPARQL(bound, $=, \neq_c$) and SPARQL(bound, $\neq, \neq_c$) is effectively decidable, by catching the reason for unsatisfiability described above. Note also that the two fragments can not be combined, since satisfiability for SPARQL($=, \neq$) is undecidable as we will see in the next Section.

3.1 Checking Bound Variables

To perform bound checks on variables, we associate to every pattern P a set $\Gamma(P)$ of schemes, where a *scheme* is simply a set of variables, in the following way.[2]

- If P is a triple pattern (u, v, w), then $\Gamma(P) := \{\{u, v, w\} \cap V\}$.
- $\Gamma(P_1 \text{ UNION } P_2) := \Gamma(P_1) \cup \Gamma(P_2)$.
- $\Gamma(P_1 \text{ AND } P_2) := \{S_1 \cup S_2 \mid S_1 \in \Gamma(P_1) \text{ and } S_2 \in \Gamma(P_2)\}$.
- $\Gamma(P_1 \text{ OPT } P_2) := \Gamma(P_1 \text{ AND } P_2) \cup \Gamma(P_1)$.
- $\Gamma(P_1 \text{ FILTER } C) := \{S \in \Gamma(P_1) \mid S \vdash C\}$, where $S \vdash C$ is defined as follows:
  - If C is of the form bound($?x$) or $?x = c$ or $?x \neq c$, then $S \vdash C$ if $?x \in S$;
  - If C is of the form $?x = ?y$ or $?x \neq ?y$, then $S \vdash C$ if $?x, ?y \in S$;
  - $S \vdash \neg\text{bound}(?x)$ if $?x \notin S$.

Example 1. Consider the pattern

$$P = (?x, p, ?y) \text{ OPT } ((?x, q, ?z) \text{ UNION } (?x, r, ?u)).$$

For the subpattern $P_1 = (?x, q, ?z) \text{ UNION } (?x, r, ?u)$ we have $\Gamma(P_1) = \{\{?x, ?z\}, \{?x, ?u\}\}$. Hence, $\Gamma((?x, p, ?y) \text{ AND } P_1) = \{\{?x, ?y, ?z\}, \{?x, ?y, ?u\}\}$. We conclude that $\Gamma(P) = \{\{?x, ?y\}, \{?x, ?y, ?z\}, \{?x, ?y, ?u\}\}$.

Example 2. For another example, consider the pattern

$$P = ((?x, p, ?y) \text{ OPT } ((?x, q, ?z) \text{ FILTER } ?y = ?z)) \text{ FILTER } ?x \neq c.$$

We have $\Gamma(?x, q, ?z) = \{\{?x, ?z\}\}$. Note that $\{?x, ?z\} \not\vdash ?y = ?z$, because $?y \notin \{?x, ?z\}$. Hence, for the subpattern $P_1 = (?x, q, ?z) \text{ FILTER } ?y = ?z$ we have $\Gamma(P_1) = \emptyset$. For the subpattern $P_2 = (?x, p, ?y) \text{ OPT } P_1$ we then have $\Gamma(P_2) = \Gamma(?x, p, ?y) = \{\{?x, ?y\}\}$. Since $\{?x, ?y\} \vdash ?x \neq c$, we conclude that $\Gamma(p) = \{\{?x, ?y\}\}$. □

[2] We define $\Gamma(P)$ for general patterns, not only for those belonging to the fragments considered in this Section, because we will make another use of $\Gamma(P)$ in Section 5.

We now establish the main result of this Section.

Theorem 3. *Let P be a SPARQL*$(\text{bound}, =, \neq_c)$ *or SPARQL*$(\text{bound}, \neq, \neq_c)$ *pattern. Then P is satisfiable if and only if $\Gamma(P)$ is nonempty.*

The only-if direction of Theorem 3 is the easy direction and is given by the following Lemma 4. Note that this lemma holds for general patterns; it can be straightforwardly proven by induction on the structure of P.

Lemma 4. *Let P be a pattern and G a graph. If $\mu \in [\![P]\!]_G$ then there exists $S \in \Gamma(P)$ such that* $\text{dom}(\mu) = S$.

The if direction of Theorem 3 for SPARQL$(\text{bound}, =, \neq_c)$ is given by the following Lemma 5.

In the following we use $\text{var}(P)$ to denote the set of all variables occurring in a pattern P.[3]

Lemma 5. *Let P be a pattern in SPARQL*$(\text{bound}, =, \neq_c)$. *Let $c \in I$ be a constant that does not appear in any constant-nonequality filter condition in P. With the constant mapping $\mu : \text{var}(P) \to \{c\}$, let G be the RDF graph consisting of all possible triples $(\mu(u), \mu(v), \mu(w))$ where (u, v, w) is a triple pattern in P.*

Then for every $S \in \Gamma(P)$ there exists $S' \supseteq S$ such that $\mu|_{S'}$ belongs to $[\![P]\!]_G$.

Proof. By induction on the structure of P. If P is a triple pattern (u, v, w) then $S = \{u, v, w\} \cap V$. Since $(\mu|_S(u), \mu|_S(v), \mu|_S(w)) = (\mu(u), \mu(v), \mu(w)) \in G$, we have $\mu|_S \in [\![P]\!]_G$ and we can take $S' = S$.

If P is of the form P_1 UNION P_2, then the claim follows readily by induction.

If P is of the form P_1 AND P_2, then we have $S = S_1 \cup S_2$ with $S_i \in \Gamma(P_i)$ for $i = 1, 2$. By induction, there exists $S'_i \supseteq S_i$ such that $\mu|_{S'_i} \in [\![P_i]\!]_G$. Clearly $\mu|_{S'_1} \sim \mu|_{S'_2}$ since they are restrictions of the same mapping. Hence $\mu|_{S'_1} \cup \mu|_{S'_2} = \mu_{S'_1 \cup S'_2} \in [\![P]\!]_G$ and we can take $S' = S'_1 \cup S'_2$.

If P is of the form P_1 OPT P_2, then there are two possibilities.

- If $S \in \Gamma(P_1 \text{ AND } P_2)$ then we can reason as in the previous case.
- If $S \in \Gamma(P_1)$ then by induction there exists $S'_1 \supseteq S$ so that $\mu|_{S'_1} \in [\![P_1]\!]_G$. Now there are two further possibilities:
  - If $\Gamma(P_2)$ is nonempty then by induction there exists some S'_2 so that $\mu|_{S'_2} \in [\![P_2]\!]_G$. We can now reason again as in the case P_1 AND P_2.
  - Otherwise, by Lemma 4 we know that $[\![P_2]\!]_G$ is empty. But then $[\![P]\!]_G = [\![P_1]\!]_G$ and we can take $S' = S'_1$.

Finally, if P is of the form P_1 FILTER C, then we know that $S \in \Gamma(P_1)$ and $S \vdash C$. By induction, there exists $S' \supseteq S$ such that $\mu|_{S'} \in [\![P_1]\!]_G$. We show that $\mu|_{S'} \in [\![P]\!]_G$ by showing that $\mu|_{S'} \models C$. There are three possibilities for C.

[3] We also use the following standard notion of restriction of a mapping. If $f : X \to Y$ is a total function and $Z \subseteq X$, then the restriction $f|_Z$ of f to Z is the total function from Z to Y defined by $f|_Z(z) = f(z)$ for every $z \in Z$. That is, $f|_Z$ is the same as f but is only defined on the subdomain Z.

- If C is of the form bound$(?x)$, then we know by $S \vdash C$ that $?x \in S'$. Hence $\mu|_{S'} \models C$.
- If C is of the form $?x = ?y$, then we again know $?x, ?y \in S'$, and certainly $\mu|_{S'} \models C$ since μ maps everything to c.
- If C is of the form $?x \neq d$, then we have $d \neq c$ by the choice of c, so $\mu|_{S'} \models C$ since $\mu(?x) = c$. □

Example 6. To illustrate the above Lemma, consider the pattern

$$P = ((?x, p, ?y)\ \mathrm{FILTER}\ ?x \neq a)\ \mathrm{OPT}\ ((?x, q, ?z)\ \mathrm{UNION}\ (?x, r, ?u))$$

which is a variant of the pattern from Example 1. As in that example, we have $\Gamma(P) = \{\{?x, ?y\}, \{?x, ?y, ?z\}, \{?x, ?y, ?u\}\}$. In this case, the mapping μ from the Lemma maps $?x$, $?y$, $?z$ and $?u$ to c. The graph G from the Lemma equals $\{(c, p, c), (c, q, c), (c, r, c)\}$, and $[\![P]\!]_G = \{\mu_1, \mu_2\}$ where $\mu_1 = \mu|_{\{?x,?y,?z\}}$ and $\mu_2 = \mu|_{\{?x,?y,?u\}}$. Now consider $S = \{?x, ?y\} \in \Gamma(P)$. Then for $S' = \{?x, ?y, ?z\}$ we indeed have $S' \supseteq S$ and $\mu|_{S'} = \mu_1 \in [\![P]\!]_G$. Note that in this example we could also have chosen $\{?x, ?y, ?u\}$ for S'. □

The counterpart to Lemma 5 for the fragment SPARQL(bound, $\neq$, $\neq_c$) is given by the following Lemma, thus settling Theorem 3 for that fragment.

Lemma 7. *Let P be a pattern in SPARQL*(bound, $\neq$, $\neq_c$)*. Let W be the set of all constants appearing in a constant-nonequality filter condition in P. Let $Z \subseteq I$ be a finite set of constants of the same cardinality as* var(P)*, and disjoint from W. With μ :* var$(P) \to Z$ *an arbitrary but fixed injective mapping, let G be the RDF graph consisting of all possible triples $(\mu(u), \mu(v), \mu(w))$ where (u, v, w) is a triple pattern in P.*

Then for every $S \in \Gamma(P)$ there exists $S' \supseteq S$ such that $\mu|_{S'}$ belongs to $[\![P]\!]_G$.

Proof. We prove for every subpattern Q of P that for every $S \in \Gamma(Q)$ there exists $S' \supseteq S$ such that $\mu|_{S'} \in [\![Q]\!]_G$. The proof is by induction on the height of Q. The reasoning is largely the same as in the proof of Lemma 5. The only difference is in the case where Q is of the form Q_1 FILTER C. In showing that $\mu_{S'} \models C$, we now argue as follows for the last two cases:

- If C is of the form $?x \neq ?y$, then $\mu|_{S'} \models C$ since μ is injective.
- If C is of the form $?x \neq c$, then $\mu|_{S'} \models C$ since Z and W are disjoint. □

3.2 Computational Complexity

In this section we show that satisfiability for the decidable fragments is NP-complete. Note that this does not immediately follow from the NP-completeness of SAT, since boolean formulas are not part of the syntax of the decidable fragments.

Theorem 3 implies the following complexity upper bound:

Corollary 8. *The satisfiability problem for SPARQL*(bound, $=$, $\neq_c$) *patterns, as well as for SPARQL*(bound, $\neq$, $\neq_c$) *patterns, belongs to the complexity class NP.*

Proof. By Theorem 3, a SPARQL(bound, $=, \neq_c$) or SPARQL(bound, $\neq, \neq_c$) pattern P is satisfiable if and only if there exists a scheme in $\Gamma(P)$. Following the definition of $\Gamma(P)$, it is clear that there is a polynomial-time nondeterministic algorithm such that, on input P, each accepting possible run computes a scheme in $\Gamma(P)$, and such that every scheme in $\Gamma(P)$ is computed by some accepting possible run.

Specifically, the algorithm works bottom-up on the syntax tree of P and computes a scheme for every subpattern. At every leaf Q, corresponding to a triple pattern in P, we compute the unique scheme in $\Gamma(Q)$. At every UNION operator we nondeterministically choose between continuing with the scheme from the left or from right child. At every AND operator we continue with the union of the left and right child schemes. At every OPT operator, we nondeterministically choose between treating it as an AND, or simply continuing with the scheme from the left. At every FILTER operation with constraint C we check for the child scheme S whether $S \vdash C$. If the check succeeds, we continue with S; if the check fails, the run is rejected. When the computation has reached the root of the syntax tree and we can compute a scheme for the root, the run is accepting and the computed scheme is the output. □

Remark 9. In our presentation of the syntax of SPARQL, we do not consider conjunction and disjunction in filter conditions. Extending the syntax to allow this would not ruin the NP upper bound. Allowing conjunctions and disjunctions, we would need to extend the definition of $\Gamma(P)$ in the obvious manner, defining $S \vdash C_1 \vee C_2$ if $S \vdash C_1$ or $S \vdash C_2$, and similarly for the definition of $S \vdash C_1 \wedge C_2$. The results would then carry through. □

We next show that satisfiability is actually NP-hard, even for patterns not using any OPT operators and using only bound constraints in filter conditions.

Proposition 10. *The satisfiability problem for OPT-free patterns in SPARQL*(bound) *is NP-hard.*

Proof. We define the problem Nested Set Cover as follows:

Input: A finite set T and a finite set E of sets of subsets of T. (So, every element of E is a set of subsets of T.)

Decide: Whether for each element e of E we can choose a subset S_e in e, so that $\bigcup_{e \in E} S_e = T$.

We will show later that the above problem is NP-hard; let us first describe how it can be reduced in polynomial time to the satisfiability problem at hand. Consider an input (T, E) for Nested Set Cover. Without loss of generality we may assume that T is a set of variables $\{?x_1, ?x_2, \dots, ?x_n\}$. Fix some constant c. For any subset S of T, we can make a pattern P_S by taking the AND of all (x, c, c) for $x \in S$. Now for a set e of subsets of T, we can form the pattern P_e by taking the UNION of all P_S for $S \in e$. Finally, we form the pattern P_E by taking the AND of all P_e for $e \in E$.

Now consider the following pattern which we denote by $P_{(T,E)}$:

$$P_E \, \text{FILTER} \, \text{bound}(?x_1) \, \text{FILTER} \, \text{bound}(?x_2) \dots \text{FILTER} \, \text{bound}(?x_n)$$

We claim that $P_{(T,E)}$ is satisfiable if and only if (T,E) is a yes-instance for Nested Set Cover. To see the only-if direction, let G be a graph such that $[\![P_{(T,E)}]\!]_G$ is nonempty, i.e., has as an element some solution mapping μ. Then in particular $\mu \in [\![P_E]\!]_G$. Hence, for every $e \in E$ there exists $\mu_e \in [\![P_e]\!]_G$ such that $\mu = \bigcup_{e\in E} \mu_e$. Since P_e is the UNION of all P_S for $S \in e$, for each $e \in E$ there exists $S_e \in e$ such that $\mu_e \in [\![P_{S_e}]\!]_G$. Since P_{S_e} is the AND of all (x,c,c) for $x \in S_e$, it follows that $\mathrm{dom}(\mu_e) = S_e$. Hence, since $\mathrm{dom}(\mu) = \bigcup_{e\in E} \mathrm{dom}(\mu_e)$, we have $\mathrm{dom}(\mu) = \bigcup_{e\in E} S_e$. However, by the bound constraints in the filters applied in $P_{(T,E)}$, we also have $\mathrm{dom}(\mu) = \{?x_1, \dots, ?x_n\} = T$. We conclude that $T = \bigcup_{e\in E} S_e$ as desired.

For the if-direction, assume that for each $e \in E$ there exists $S_e \in e$ such that $T = \bigcup_{e\in E} S_e$. Consider the singleton graph $G = \{(c,c,c)\}$. For any subset S of T, let $\mu_S : S \to \{c\}$ be the constant solution mapping with domain S. Clearly, $\mu_S \in [\![P_S]\!]_G$, so $\mu_{S_e} \in [\![P_e]\!]_G$ for every $e \in E$. All the μ_S map to the same constant, so they are all compatible. Hence, for $\mu = \bigcup_{e\in E} \mu_{S_e}$, we have $\mu \in [\![P_E]\!]_G$. Since $\mathrm{dom}(\mu) = \bigcup_{e\in E} \mathrm{dom}(\mu_{S_e}) = \bigcup_{e\in E} S_e = T = \{?x_1, \dots, ?x_n\}$, the mapping μ satisfies every constraint $\mathrm{bound}(?x_i)$ for $i = 1, \dots, n$. We conclude that $\mu \in [\![P_{(E,T)}]\!]_G$ as desired.

It remains to show that Nested Set Cover is NP-hard. Thereto we reduce the classical CNF-SAT problem. Assume given a boolean formula ϕ in CNF, so ϕ is a conjunction of clauses, where each clauses is a disjunction of literals (variables or negated variables). We construct an input (T,E) for Nested Set Cover as follows. Denote the set of variables used in ϕ by W.

For T we take the set of clauses of ϕ. For any variable $x \in W$, consider the set Pos_x consisting of all clauses that contain a positive occurrence of x, and the set Neg_x consisting of all clauses that contain a negative occurrence of x. Then we define e_x as the pair $\{\mathrm{Pos}_x, \mathrm{Neg}_x\}$.

Now E is defined as the set $\{e_x \mid x \in W\}$. It is clear that ϕ is satisfiable if and only if the constructed input is a yes-instance for Nested Set Cover. Indeed, truth assignments to the variables correspond to selecting either Pos_x or Neg_x from e_x for each $x \in W$. □

4. Undecidable Fragments

In this Section we show that the two decidable fragments SPARQL(bound, $=$, $\neq_c$) and SPARQL(bound, $\neq$, $\neq_c$) are, in a sense, maximal. Specifically, the three minimal fragments not subsumed by one of these two fragments are SPARQL($\neg$bound), SPARQL($=$, $\neq$), and SPARQL($=_c$). The main result of this Section is:

Theorem 11. *Satisfiability is undecidable for SPARQL(*$\neg$bound*) patterns, for SPARQL(*$=$, $\neq$*) patterns, and for SPARQL(*$=_c$*) patterns.*

We will prove this theorem by reducing from the satisfiability problem for the algebra of finite binary relations with union, composition, and difference (Tan et al., 2014). This algebra is also called the Downward Algebra and denoted by DA. The expressions of DA are defined as follows. Let R be an arbitrary fixed binary relation symbol.

- The symbol R is a DA-expression.
- If e_1 and e_2 are DA-expressions, then so are $e_1 \cup e_2$, $e_1 - e_2$, and $e_1 \circ e_2$.

Semantically, DA-expressions represent binary queries on binary relations, i.e., mappings from binary relations to binary relations. Let J be a binary relation. For DA-expression e, we define the binary relation $e(J)$ inductively as follows:

- $R(J) = J$;
- $(e_1 \cup e_2)(J) = e_1(J) \cup e_2(J)$;
- $(e_1 - e_2)(J) = e_1(J) - e_2(J)$ (set difference);
- $(e_1 \circ e_2)(J) = \{(x, z) \mid \exists y : (x, y) \in e_1(J) \text{ and } (y, z) \in e_2(J)\}$.

A DA-expression is called *satisfiable* if there exists a finite binary relation J such that $e(J)$ is nonempty.

Example 12. An example of a DA-expression is $e = (R \circ R) - R$. If J is the binary relation $\{(a, b), (b, c), (a, c), (c, d)\}$ then $e(J) = \{(b, d), (a, d)\}$. An example of an unsatisfiable DA expression is $((R \circ R - R) \circ R) - (R \circ R \circ R)$. □

We recall the following result. It is actually well known (Andréka, Givant, & Németi, 1997) that relational composition together with union and complementation leads to an undecidable algebra; the following result simplifies matters by showing that undecidability already holds for expressions over a single relation symbol and using set difference instead of complementation. The following result has been proven by reduction from the universality problem for context-free grammars.

Theorem 13 (Tan et al., 2014)**.** *The satisfiability problem for DA-expressions is undecidable.*

4.1 Expressing MINUS

The main problem we face in reducing from DA to the SPARQL fragments stated in Theorem 11, is to emulate the difference operator. We review here more generally how to emulate the MINUS operator, which is the most meaningful counterpart of the relational difference operator in the SPARQL context.

The MINUS operator is defined as follows. For two patterns P_1 and P_2 and a graph G, we define

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

where we reuse the $\smallsetminus$ operation on sets of mappings, already seen in the definition of OPT in Section 2.3.

For the fragment SPARQL($\neg$bound), expressibility of MINUS is already known:

Lemma 14 (Arenas & Pérez, 2011)**.** *MINUS is expressible in SPARQL*($\neg$bound)*. More precisely, for any two patterns P_1 and P_2 and any graph G, we have $[\![P_1 \text{MINUS} P_2]\!]_G = [\![P]\!]_G$ where P is the pattern*

$$\big(P_1 \text{ OPT } (P_2 \text{ AND } (?u, ?v, ?w))\big) \text{ FILTER } \neg\text{bound}(?u).$$

Here, $?u$, $?v$ and $?w$ are fresh variables not used in P_1 or P_2.

Our task is to find similar expressions in the two other fragments SPARQL($=,\neq$) and SPARQL($=_c$). We will actually only be able to express MINUS up to projection, and under some mild assumptions on the graph G.

As for projection, its counterpart in SPARQL is the operation SELECT, defined as follows. Let P be a pattern and let S be a finite set of variables. Then $\mathrm{SELECT}_S P$ restricts the solution mappings coming from P to the variables listed in S. Formally, for any graph G, we define

$$[\![\mathrm{SELECT}_S P]\!]_G = \{\mu|_{S\cap\mathrm{dom}(\mu)} \mid \mu \in [\![P]\!]_G\}.$$

The assumptions on the graph G we need to make have to do with its active domain. Intuitively, the active domain of a graph is the set of all entries of triples in the graph. Formally, we define

$$\mathrm{adom}(G) = \{s \mid \exists p, o : (s,p,o) \in G\} \cup \{p \mid \exists s, o : (s,p,o) \in G\} \cup \{o \mid \exists s, p : (s,p,o) \in G\}.$$

We can easily express the active domain in SPARQL, in the following sense. Using three variables $?u$, $?v$, $?w$, consider the pattern

$$\mathit{adom} = (?u, ?v, ?w) \text{ UNION } (?w, ?u, ?v) \text{ UNION } (?v, ?w, ?u).$$

Then for any graph, we have

$$\begin{aligned} \mathrm{adom}(G) &= \{\mu(?u) \mid \mu \in [\![\mathit{adom}]\!]_G\} \\ &= \{\mu(?v) \mid \mu \in [\![\mathit{adom}]\!]_G\} \\ &= \{\mu(?w) \mid \mu \in [\![\mathit{adom}]\!]_G\}. \end{aligned}$$

We are now ready to state the counterpart of Lemma 14 for SPARQL($=,\neq$).

Lemma 15. *MINUS is expressible in SPARQL($=,\neq$), up to projection and on graphs with at least two distinct elements. More precisely, for any two patterns P_1 and P_2 and any graph G such that* $\mathrm{adom}(G)$ *has at least two distinct elements, we have the equality* $[\![P_1 \text{ MINUS } P_2]\!]_G = [\![\mathrm{SELECT}_{\mathrm{var}(P_1)} P]\!]_G$*, where P is the pattern*

$$\Big(\big(P_1 \text{ OPT } ((P_2 \text{ AND } \mathit{adom} \text{ AND } \mathit{adom}') \text{ FILTER } ?u \neq ?u')\big) \\ \text{ AND } \mathit{adom} \text{ AND } \mathit{adom}'\Big) \text{ FILTER } ?u = ?u'.$$

Here, adom' is a copy of the adom pattern with different variables $?u'$, $?v'$ and $?w'$. These variables, and the variables $?u$, $?v$ and $?w$ used in adom, are fresh variables not used in P_1 or P_2.

Proof. To prove the equality stated in the Theorem we are going to consider both inclusions. For easy reference we name some subpatterns of P as follows.

- P_2' denotes $(P_2 \text{ AND } \mathit{adom} \text{ AND } \mathit{adom}') \text{ FILTER } ?u \neq ?u'$;
- P_3 denotes $P_1 \text{ OPT } P_2'$.

- Thus, P is (P_3 AND *adom* AND *adom*$'$) FILTER $?u = ?u'$.

To prove the inclusion from right to left, let $\mu \in [\![P]\!]_G$. Then $\mu = \mu_3 \cup \varepsilon$, where $\mu_3 \in [\![P_3]\!]_G$ and ε is a mapping defined on $\{?u, ?v, ?w, ?u', ?v', ?w'\}$ such that $\varepsilon(?u) = \varepsilon(?u')$. In particular, $\mu_3 \sim \varepsilon$. Since $P_3 = P_1$ OPT P_2', there are two possibilities for μ_3:

- $\mu_3 \in [\![P_1]\!]_G$ and there is no $\mu_2' \in [\![P_2']\!]_G$ such that $\mu_3 \sim \mu_2'$. Then $\mu_3 = \mu|_{\mathrm{var}(P_1)}$, so it remains to show that there does not exist $\mu_2 \in [\![P_2]\!]_G$ such that $\mu_3 \sim \mu_2$. Assume the contrary. Since adom(G) has at least two distinct elements, μ_2 can be extended to a mapping $\mu_2' \in [\![P_2']\!]_G$. Then $\mu_2 \sim \mu_2' \sim \mu_3$, which is a contradiction.

- $\mu_3 = \mu_1 \cup \mu_2'$ with $\mu_1 \in [\![P_1]\!]_G$ and $\mu_2' \in [\![P_2']\!]_G$. In particular, μ_3 is defined on $?u$ and $?u'$ and $\mu_3(?u) \neq \mu_3(?u')$. On the other hand, since $\mu_3 \sim \varepsilon$, and $\varepsilon(?u) = \varepsilon(?u')$, also $\mu_3(?u) = \mu_3(?u')$. This is a contradiction, so the possibility under consideration cannot happen.

To prove the inclusion from left to right, let $\mu_1 \in [\![P_1 \,\mathrm{MINUS}\, P_2]\!]_G$. Assume, for the sake of argument, that there would exist $\mu_2' \in [\![P_2']\!]_G$ such that $\mu_1 \sim \mu_2'$. Mapping μ_2' contains a mapping $\mu_2 \in [\![P_2]\!]_G$, by definition of P_2'. Since $\mu_1 \sim \mu_2'$, also $\mu_1 \sim \mu_2$ which is not possible.

So, we now know that there does not exist $\mu_2' \in [\![P_2']\!]_G$ such that $\mu_1 \sim \mu_2'$. Hence, $\mu_1 \in [\![P_3]\!]_G$. Note that the six variables $?u$, $?u'$, $?v$, $?v'$, $?w$, and $?w'$ do not belong to var(P_1). Since G is nonempty, μ_1 can thus be extended to a mapping $\mu \in [\![P]\!]_G$. We conclude $\mu_1 \in [\![\mathrm{SELECT}_{\mathrm{var}(P_1)} P]\!]_G$ as desired. □

The analogous result for the fragment SPARQL($=_c$) is as follows. Fix two distinct constants a and b arbitrarily.

Lemma 16. *MINUS is expressible in SPARQL($=_c$), up to projection and on graphs in which a and b appear. More precisely, for any two patterns P_1 and P_2 and any graph G such that a and b belong to* adom(G)*, we have the equality* $[\![P_1 \,\mathrm{MINUS}\, P_2]\!]_G = [\![\mathrm{SELECT}_{\mathrm{var}(P_1)} P]\!]_G$*, where P is the pattern*

$$\Big(\big(P_{e_1} \text{ OPT } ((P_{e_2} \text{ AND } \mathit{adom}_{?u}) \text{ FILTER } ?u = a)\big) \text{ AND } \mathit{adom}_{?u}\Big) \text{ FILTER } ?u = b.$$

As always, in the above expression, the variables $?u$, $?v$ and $?w$ used in *adom* are taken to be fresh variables not used in P_1 or P_2.

The correctness proof of the above Lemma is analogous to the proof given for Lemma 15; instead of exploiting the inconsistency between $?u \neq ?u'$ and $?u = ?u'$ as done in that proof, we now exploit the inconsistency between $?u = a$ and $?u = b$.

4.2 Reduction from the Downward Algebra

We are now ready to formulate the reduction from the satisfiability problem for DA to the satisfiability problem for the three fragments mentioned in Theorem 11. We precisely formulate the reduction and prove the Theorem for the fragment SPARQL(¬bound) first. After that we will discuss how the reduction must be adapted for the other two fragments.

We say that an RDF graph G *represents* a binary relation J if $J = \{(s, o) \mid \exists p : (s, p, o) \in G\}$. Intuitively, we view an RDF graph as a binary relation by ignoring the middle column.

Lemma 17. *For every DA-expression e there exists a SPARQL($\neg$bound) pattern P_e with the following properties:*

1. *there exist two distinct fixed variables $?x$ and $?y$ such that for every RDF graph G and every $\mu \in [\![P_e]\!]_G$, $?x$ and $?y$ belong to $\text{dom}(\mu)$;*
2. *for every binary relation J and RDF graph G that represents J, we have*
$$e(J) = \{(\mu(?x), \mu(?y)) \mid \mu \in [\![P_e]\!]_G\};$$

Proof. By induction on the structure of e. If e is R then P_e is the triple pattern $(?x, ?z, ?y)$.

If e is of the form $e_1 \cup e_2$, then P_e is P_{e_1} UNION P_{e_2}.

If e is of the form $e_1 \circ e_2$, then P_e is P'_{e_1} AND P'_{e_2}, where P'_{e_1} and P'_{e_2} are obtained as follows. First, by renaming variables, we may assume without loss of generality that P_{e_1} and P_{e_2} have no variables in common other than $?x$ and $?y$. Let $?z$ be a fresh variable. Now in P_{e_1}, rename $?y$ to $?z$, yielding P'_{e_1}, and in P_{e_2}, rename $?x$ to $?z$, yielding P'_{e_2}.

Finally, if e is of the form $e_1 - e_2$, we use the expression P from Lemma 14 applied to P_{e_1} and P_{e_2}. As before we may assume without loss of generality that P_{e_1} and P_{e_2} have no variables in common other than $?x$ and $?y$. □

From the above lemma we clearly have that e is satisfiable if and only if P_e is satisfiable. We thus have a reduction from satisfiability for DA to satisfiability for SPARQL($\neg$bound), showing undecidability of the latter problem.

We now discuss the two remaining fragments.

4.3 SPARQL($=, \neq$)

For this fragment we consider a minor variant of satisfiability for DA-expressions where we restrict attention to binary relations over at least two elements. Formally, the *active domain* of a binary relation J is the set of all entries in pairs belonging to J, so $\text{adom}(J) := \{x \mid \exists y : (x, y) \in J \text{ or } (y, x) \in J\}$. Then a DA-expression e is called *two-satisfiable* if $e(J)$ is nonempty for some J such that $\text{adom}(J)$ has at least two distinct elements.

Clearly, two-satisfiability is undecidable as well, for if it were decidable, then satisfiability would be decidable too. Indeed, e is satisfiable if and only if it is two-satisfiable, or satisfiable by a binary relation J over a single element. Up to isomorphism there is only one such J (the singleton $\{(x, x)\}$), so that case could be checked separately.

Lemma 17 can now be adapted by claiming the second property only for binary relations J over at least two distinct elements. In the proof for the case where e is $e_1 - e_2$, we can then use Lemma 15.

Using the adapted lemma, we can now reduce two-satisfiability for DA to satisfiability for SPARQL($=, \neq$). All we need extra is a test whether the graph represents a binary relation over at least two distinct elements. We can use the following pattern *test* (using fresh variables $?u, \ldots, ?w'$ as usual):

$$\begin{aligned}&(((?u, ?v, ?w) \text{ AND } (?u', ?v', ?w')) \text{ FILTER } ?u \neq ?u')\\ &\qquad \text{UNION } (((?u, ?v, ?w) \text{ AND } (?u', ?v', ?w')) \text{ FILTER } ?w \neq ?w')\\ &\qquad\qquad \text{UNION } (((?u, ?v, ?w) \text{ AND } (?u', ?v', ?w')) \text{ FILTER } ?u \neq ?w')\end{aligned}$$

Then, e is two-satisfiable if and only if P_e AND *test* is satisfiable.

4.4 SPARQL($=_c$)

For this fragment we consider a further variant of two-satisfiability, called *ab-satisfiability*, for two arbitrary fixed constants $a, b \in I$. A DA-expression is called ab-satisfiable if $e(J)$ is nonempty for some binary relation J where $a, b \in \mathrm{adom}(J)$.

DA-expressions do not distinguish between isomorphic binary relations. Hence, ab-satisfiability is equivalent to two-satisfiability, and thus still undecidable.

We now again adapt Lemma 17, as follows. The second property is now claimed only for binary relations J where $a, b \in \mathrm{adom}(J)$. In the proof for the case $e = e_1 - e_2$, we now use Lemma 16.

We then obtain that e is ab-satisfiable if and only if P_e AND $test_{ab}$ is satisfiable, where $test_{ab}$ is the following pattern which tests whether the graph represents a binary relation with a and b in its active domain:

$$(((?u, ?v, ?w) \text{ UNION } (?w, ?v, ?u)) \text{ AND } ((?u', ?v', ?w') \text{ UNION } (?w', ?v', ?u'))) \\ \text{FILTER } ?u = a \text{ FILTER } ?u' = b$$

Remark 18. Recall that literals cannot appear in first or second position in an RDF triple. Patterns using constant-equality predicates can be unsatisfiable because of that reason. For example, using the literal 42, the pattern $(?x, ?y, ?z)$ FILTER $?y = 42$ is unsatisfiable. However, we have seen here that the use of constant-equality predicates leads to undecidability of satisfiability for a much more fundamental reason, that has nothing to do with literals, namely, the ability to emulate set difference.

5. Satisfiability of Well-Designed Patterns

The *well-designed* patterns (Pérez et al., 2009) have been identified as a well-behaved class of SPARQL patterns, with properties similar to the conjunctive queries for relational databases (Abiteboul et al., 1995). Standard conjunctive queries are always satisfiable, and conjunctive queries extended with equality and nonequality constraints, possibly involving constants, can only be unsatisfiable if the constraints are inconsistent. An analogous behavior is present in what we call *AF-patterns*: patterns that only use the AND and FILTER operators. We will formalize this in Proposition 19. We will then show in Theorem 21 that a well-designed pattern is satisfiable if and only if its reduction to an AF-pattern is satisfiable. In other words, as far as satisfiability is concerned, well-designed patterns can be treated like AF-patterns.

5.1 Satisfiability of AF-Patterns

In Section 3.1 we have associated a set of schemes $\Gamma(P)$ to every pattern P. When $\Gamma(P)$ is empty, P is unsatisfiable (Lemma 4).

Now when P is an AF-pattern and $\Gamma(P)$ is nonempty, the satisfiability of P will turn out to depend solely on the equalities, nonequalities, constant-equalities, and constant-nonequalities occurring as filter conditions in P. We will denote the set of these constraints by $C(P)$.

Any set Σ of constraints is called *consistent* if there exists a mapping that satisfies every constraint in Σ.

We establish:

Proposition 19. *An AF-pattern P is satisfiable if and only if $\Gamma(P)$ is non-empty and $C(P)$ is consistent.*

Proof. The only-if direction of this proposition is given by Lemma 4 together with the observation that if $\mu \in [\![P]\!]_G$, then μ satisfies every constraint in $C(P)$. Since P is satisfiable, such G and μ exist, so $C(P)$ is consistent.

For the if direction, since P does not have the UNION and OPT operators, $\Gamma(P)$ is a singleton $\{S\}$. Since $C(P)$ is consistent, there exists a mapping $\mu : S \to U$ satisfying every constraint in $C(P)$. Let G be the graph consisting of all triples $(\mu(u), \mu(v), \mu(w))$ where (u, v, w) is a triple pattern in P. It is straightforward to show by induction on the height of Q that for every subpattern Q of P, we have $\mu|_{S'} \in [\![Q]\!]_G$, where $\Gamma(Q) = \{S'\}$. Hence $\mu \in [\![P]\!]_G$ and P is satisfiable. □

Note that $\Gamma(P)$ can "blow up" only because of possible UNION and OPT operators, which are missing in an AF-pattern. Hence, for an AF-pattern P, we can efficiently compute $\Gamma(P)$ by a single bottom-up pass over P. Morever, $C(P)$ is a conjunction of possibly negated equalities and constant equalities. It is well known that consistency of such conjunctions can be decided in polynomial time (Kroening & Strichman, 2008). Hence, we conclude:

Corollary 20. *Satisfiability for AF-patterns can be checked in polynomial time.*

5.2 AF-Reduction of Well-Designed Patterns

A well-designed pattern is defined as a union of union-free well-designed patterns. Since a union is satisfiable if and only if one of its terms is, we will focus on union-free patterns in what follows. Formally, a union-free pattern P is called *well-designed* (Pérez et al., 2009) if

1. for every subpattern of P of the form Q FILTER C, all variables mentioned in C also occur in Q; and
2. for every subpattern Q of P of the form Q_1 OPT Q_2, and every $?x \in \text{var}(Q_2)$, if $?x$ also occurs in P outside of Q, then $?x \in \text{var}(Q_1)$.

We associate to every union-free pattern P an AF-pattern $\rho(P)$ obtained by removing all applications of OPT and their right operands; the left operand remains in place. Formally, we define the following:

- If P is a triple pattern, then $\rho(P)$ equals P.
- If P is of the form P_1 AND P_2, then $\rho(P) = \rho(P_1)$ AND $\rho(P_2)$.
- If P is of the form P_1 FILTER C, then $\rho(P) = \rho(P_1)$ FILTER C.
- If P is of the form P_1 OPT P_2, then $\rho(P) = \rho(P_1)$.

The announced result is now given by the following theorem, which is proved directly from results by Pérez et al. (2009).[4]

[4] We thank an anonymous referee for offering the given proof of the only-if direction.

Theorem 21. *Let P be a union-free well-designed pattern. Then P is satisfiable if and only if $\rho(P)$ is.*

Proof. We are going to refer to Lemma 4.3 and Proposition 4.5 by Perez et al. (2009). Indeed, Lemma 4.3 gives us the if-direction of Theorem 21. The cited paper introduced the notion of a reduction $P' \trianglelefteq P$. Whenever $P' \trianglelefteq P$, also $\rho(P) \trianglelefteq P'$ and $\rho(P) = \rho(P')$.

Now for the only-if direction, assume P is satisfiable, so there exists G and μ so that $\mu \in [\![P]\!]_G$. Then there exists $P' \trianglelefteq P$ such that $\mu \in [\![\text{and}(P')]\!]_G$ (Proposition 4.5). Here, and(P') denotes the pattern obtained from P' by replacing every OPT by AND. By the above we have $\rho(P) = \rho(P')$.

Now the following claim is easy to verify for every union-free pattern P': *If $\mu \in [\![\text{and}(P')]\!]_G$ then $\mu|_{\text{var}(\rho(P'))} \in [\![\rho(P')]\!]_G$.* By that claim, we obtain that $[\![\rho(P')]\!]_G$ is nonempty so $\rho(P') = \rho(P)$ is satisfiable, as desired. □

Since $\rho(P)$ can be efficiently computed from P, the above Theorem and Corollary 20 imply:

Corollary 22. *Satisfiability of union-free well-designed patterns can be tested in polynomial time.*

6. Experimental Evaluation

We want to evaluate experimentally the positive results presented so far:

1. Wrong literal reduction (Proposition 24);

2. Satisfiability checking for SPARQL(bound, $=$, $\neq_c$) and SPARQL(bound, $\neq$, $\neq_c$) by computing $\Gamma(P)$ (Theorem 3);

3. Satisifiability checking for well-designed patterns, by reduction to AF-patterns (Proposition 19 and Theorem 21).

Our experiments follow up on those reported earlier by Picalausa and Vansummeren (2011). As test datasets of real-life SPARQL queries, we use logs of the SPARQL endpoint for DBpedia.[5] This data source contains the "query dumps" from the year 2012, divided into 14 logfiles. Out of these we chose the three logs 20120913, 20120929 and 20121031 to obtain a span of roughly three months; we then took a sample of 100 000 queries from each of them. A typical query in the log has size between 75 and 125 (size measured as number of nodes in the syntax tree). About 10% of the queries in each log is not usable because they have syntax errors or because they use features not covered by our analysis.

The implementation of the tests was done in Java 7 under Windows 7, on an Intel Core 2 Duo SU94000 processor (1.40GHz, 800MHz, 3MB) with 3GB of memory (SDRAM DDR3 at 1067MHz).

Our tests measure the time needed to perform the analyses of SPARQL queries presented above. The timings are averaged over all queries in a log, and each experiment is repeated five times to smooth out accidental quirks of the operating system. Although we give

[5] `ftp://download.openlinksw.com/support/dbpedia/`

Table 1: Timings of experiments (averaged over five repeats). Times are in ms. Baseline is time to read and parse 1000 000 queries; WL stands for baseline plus time for wrong-literal reduction. $\Gamma(P)$ stands for WL plus time for computing $\Gamma(P)$. AF stands for WL, plus testing well-designedness, plus doing AF-reduction and testing satisfiability (Proposition 19). The percentages show the increases relative to the baseline.

logfile	baseline	WL		$\Gamma(P)$		AF	
20120913	39 422	41 254	5%	44 395	8%	48 329	10%
20120929	34 281	35 868	5%	38 102	7%	41 087	9%
20121031	32 286	33 186	3%	34 419	4%	36 993	8%

absolute timings, the main emphasis is on the percentage of the time needed to analyse a query, with respect to the time needed simply to read and parse that query. If this percentage is small this demonstrates efficient, linear time complexity in practice. It will turn out that this is indeed achieved by our experiments, as shown in Table 1.

In the following subsections we discuss the results in more detail.

6.1 Wrong Literal Reduction

Testing for and removing triple patterns with wrong literals in a pattern P is performed by the reduction $\lambda(P)$ defined in the Appendix. From the definition of $\lambda(P)$ it is clear that it can be computed by a single bottom-up traversal of P and this is indeed borne out by our experiments. Table 1 shows that on average, wrong-literal reduction takes between 3 and 5% of the time needed to read and parse the input.

Interestingly, some real-life queries with literals in the wrong position were indeed found; one example is the following:

```
SELECT DISTINCT *
WHERE { 49  dbpedia-owl:wikiPageRedirects  ?redirectLink .}
```

6.2 Computing $\Gamma(P)$

In Section 3 we have seen that satisfiability for the decidable fragments can be tested by computing $\Gamma(P)$, but that the problem is NP-complete. Intuitively, the problem is intractable because $\Gamma(P)$ may be of size exponential in the size of P. This actually occurs in real life; a common SPARQL query pattern is to use many nested OPTIONAL operators to gather additional information that is not strictly required by the query but may or may not be present. We found in our experiments queries with up to 50 nested OPT operators, which naively would lead to a $\Gamma(P)$ of size 2^{50}. A shortened example of such a query is shown in Figure 1.

In practice, however, the blowup of $\Gamma(P)$ can be avoided as follows. Recall that Theorem 3 states that P is satisfiable if and only if $\Gamma(P)$ is nonempty. The elements of $\Gamma(P)$ are sets of variables. Looking at the definition of $\Gamma(P)$, a set may be removed from $\Gamma(P)$ only

```
SELECT DISTINCT *
WHERE {
?s a <http://dbpedia.org/ontology/EducationalInstitution>,
<http://dbpedia.org/ontology/University> .
?s <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Brazil> .
OPTIONAL {?s <http://dbpedia.org/ontology/affiliation> ?ontology_affiliation .}
OPTIONAL {?s <http://dbpedia.org/ontology/abstract> ?ontology_abstract .}
OPTIONAL {?s <http://dbpedia.org/ontology/campus> ?ontology_campus .}
OPTIONAL {?s <http://dbpedia.org/ontology/chairman> ?ontology_chairman .}
OPTIONAL {?s <http://dbpedia.org/ontology/city> ?ontology_city .}
OPTIONAL {?s <http://dbpedia.org/ontology/country> ?ontology_country .}
OPTIONAL {?s <http://dbpedia.org/ontology/dean> ?ontology_dean .}
OPTIONAL {?s <http://dbpedia.org/ontology/endowment> ?ontology_endowment .}
OPTIONAL {?s <http://dbpedia.org/ontology/facultySize> ?ontology_facultySize .}
OPTIONAL {?s <http://dbpedia.org/ontology/formerName> ?ontology_formerName .}
OPTIONAL {?s <http://dbpedia.org/ontology/head> ?ontology_head .}
OPTIONAL {?s <http://dbpedia.org/ontology/mascot> ?ontology_mascot .}
OPTIONAL {?s <http://dbpedia.org/ontology/motto> ?ontology_motto .}
OPTIONAL {?s <http://dbpedia.org/ontology/president> ?ontology_president .}
OPTIONAL {?s <http://dbpedia.org/ontology/principal> ?ontology_principal .}
OPTIONAL {?s <http://dbpedia.org/ontology/province> ?ontology_province .}
OPTIONAL {?s <http://dbpedia.org/ontology/rector> ?ontology_rector .}
OPTIONAL {?s <http://dbpedia.org/ontology/sport> ?ontology_sport .}
OPTIONAL {?s <http://dbpedia.org/ontology/state> ?ontology_state .}
OPTIONAL {?s <http://dbpedia.org/property/acronym> ?property_acronym .}
OPTIONAL {?s <http://dbpedia.org/property/address> ?property_address .}
OPTIONAL {?s <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?property_lat .}
OPTIONAL {?s <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?property_long .}
OPTIONAL {?s <http://dbpedia.org/property/established> ?property_established .}
OPTIONAL {?s <http://dbpedia.org/ontology/logo> ?ontology_logo .}
OPTIONAL {?s <http://dbpedia.org/property/website> ?property_website .}
OPTIONAL {?s <http://dbpedia.org/property/location> ?property_location .}
FILTER ( langMatches(lang(?ontology_abstract), "es") ||
langMatches(lang(?ontology_abstract), "en") )
FILTER ( langMatches(lang(?ontology_motto), "es") ||
langMatches(lang(?ontology_motto), "en") )
}
```

Figure 1: A real-life query with many nested OPTIONAL operators, retrieving as much information as possible about universities in Brazil.

by the application of a FILTER. Hence, only variables that are mentioned in FILTER conditions can influence the emptiness of $\Gamma(P)$; other variables can be ignored. For example, in the query in Figure 1, only two variables appear in a filter, namely `?ontology_abstract` and `?ontology_motto`, so that the maximal size of $\Gamma(P)$ is reduced to 2^2.

In our experiments, it turns out that typically few variables are involved in filter conditions. Hence, the above strategy works well in practice.

Another practical issue is that, in this paper, we have only considered filter conditions that are bound checks, equalities, and constant-equalities, possibly negated. In practice, filter conditions typically apply built-in SPARQL predicates such as the predicate `langMatches` in Figure 1. For the experimental purpose of testing the practicality of computing $\Gamma(P)$, however, such predicates can simply be treated as bound checks. In this way we can apply our experiments to 70% of the queries in the testfiles.

With the above practical adaptations, our experiments show that computing $\Gamma(P)$ is efficient: Table 1 shows that it requires, on average, between 4 and 8% of the time needed to read and parse the input, and these timings even include the wrong-literal reduction

All in all, our experiments encountered very few unsatisfiable queries. This observation is corroborated by the findings of a recent new statistical analysis of practical SPARQL usage (Han, Feng, Zhang, Wang, Rao, & Jiang, 2016). Of course, that users in practice do not write unsatisfiable expressions is only good news. Satisfiability remains a basic problem that we need to understand, because many other problems can be reduced to it.

6.3 Satisfiability Testing for Well-Designed Patterns

In Section 5 we have seen that testing satisfiability of a well-designed pattern can be done by testing satisfiability of the AF-reduction (Theorem 21). The latter can be done by testing nonemptiness of $\Gamma(P)$ and testing consistency of the filter conditions (Proposition 19).

Computing the AF-reduction can be done by a simple bottom-up traversal of the pattern. Moreover, for an AF-pattern P, computing $\Gamma(P)$ poses no problems since it is either empty or a singleton. As far as testing consistency of filter conditions is concerned, our experiments yield a rather baffling observation: almost all well-designed patterns in the test sets have no filters at all. We cannot explain this phenomenon, but it implies that we have not been able to test the performance of the consistency checks on real-life SPARQL queries.

Anyhow, Table 1 shows that doing the entire analysis of wrong-literal reduction, testing well-designedness, AF-reduction, computing $\Gamma(P)$, and consistency checking (in the few cases where the latter was necessary), incurs at most a 10% increase relative to reading and parsing the input.

6.4 Scalability

The experiments described above were run on sets of 100 000 queries each. We also did a modest scaling experiment where we varied the number of queries from 5 000 to 200 000. Table 2 shows that the performance scales linearly.

Table 2: Scalability experiment (times in ms). Timings clearly scale linearly for increasing input size.

input size	200 000	100 000	50 000	10 000	5 000	Pearson coeficient
baseline	74 168	39 422	21 315	3 596	1 851	0.999924005
WL	77 800	41 253	21 876	3 762	1 942	0.999989454
$\Gamma(P)$	81 730	44 395	23 552	4 016	2 036	0.999900948
AF	91 470	48 329	26 023	4 463	2 254	0.999044542

7. Extension to SPARQL 1.1

As already mentioned in the Introduction, SPARQL 1.0 has been extended to SPARQL 1.1 with a number of new operators for building patterns. The main new features are property paths; grouping and aggregates; BIND; VALUES; MINUS; EXISTS and NOT EXISTS-subqueries; and SELECT. A complete analysis of SPARQL 1.1 goes beyond the scope of the present paper. Nevertheless, in this section, we briefly discuss how our results may be extended to this new setting.

Property paths provide a form of regular path querying over graphs. This aspect of graph querying has already been extensively investigated, including questions of satisfiability and other kinds of static analysis such as query containment (Kostylev, Reutter, & Vrgoč, 2014; Kostylev, Reutter, Romero, & Vrgoč, 2015). Therefore we do not discuss property paths any further here.

The SPARQL 1.1 features that we discuss can be grouped in two categories: those that cause undecidability, and those that are harmless as far as satisfiability is concerned. We begin with the harmless category.

7.1 SELECT Operator and EXISTS-Subqueries

SPARQL 1.1 allows patterns of the form $\text{SELECT}_S P$, where S is a finite set of variables and P is a pattern. The novelty compared to 1.0 is that this can be applied to subexpressions. The semantics is that of projection; we have already seen it in Section 4.1.

This feature in itself does not influence the satisfiability of patterns. Indeed, patterns extended with SELECT operators can be reduced to patterns without said operators. The reduction amounts simply to rename the variables that are projected out by fresh variables that are not used anywhere else in the pattern; then the SELECT operators themselves can be removed. The resulting, SELECT-free, pattern is equivalent to the original one if we omit the fresly introduced variables from the solution mappings in the final result. In particular, the two patterns are equisatisfiable.

Example 23. Rather than giving the formal definition of SELECT-reduction and formally stating and proving the equivalence, we give an example. Consider the pattern P:

$$(c, p, ?x) \text{ OPT } ((?x, p, ?y) \text{ AND SELECT}_{?y}(?y, q, ?z) \text{ AND SELECT}_{?y}(?y, r, ?z))$$

Renaming projected-out variables by fresh variables and omitting the SELECT operators yields the following pattern P':

$$(c, p, ?x) \text{ OPT } ((?x, p, ?y) \text{ AND } (?y, q, ?z_1) \text{ AND } (?y, r, ?z_2))$$

Pattern P' is equivalent to P in the sense that for any graph G, we have $[\![P]\!]_G = \{\hat{\mu} \mid \mu \in [\![P']\!]_G\}$, where $\hat{\mu}$ denotes the mapping obtained from μ by omitting the values for $?z_1$ and $?z_2$ (if at all present in $\text{dom}(\mu)$). □

Now that we know how to handle SELECT operators, we can also handle EXISTS-subqueries. Indeed, a pattern P FILTER EXISTS(Q) (with the obvious SQL-like semantics) is equivalent to $\text{SELECT}_{\text{var}(P)}(P \text{ AND } Q)$.

7.2 Features Leading to Undecidability

In Section 4 we have seen that as soon as one can express the union, composition and difference of binary relations, the satisfiability problem becomes undecidable. Since union and composition are readily expressed in basic SPARQL (UNION and AND), the key lies in the expressibility of the difference operator. In this subsection we will see that various new features of SPARQL 1.1 indeed allow expressing difference.

7.2.1 MINUS Operator and NOT EXISTS Subqueries

Each of these two features can quite obviously be used to express difference, so we do not dwell on them any further.

7.2.2 Grouping and Aggregates

A known trick for expressing difference using grouping and counting (Celko, 2005) can be emulated in the extension of SPARQL 1.0 with grouping. We illustrate the technique with an example.

Consider the query $(?x, p, ?y)$ MINUS $(?x, q, ?y)$ asking for all pairs (a, b) such that (a, p, b) holds but (a, q, b) does not. We can express this query (with the obvious SQL-like semantics) as follows:

$\text{SELECT}_{?x,?y}\big((?x, p, ?y) \text{ OPT } ((?x, q, ?y) \text{ AND } (?xx, p, ?yy))\big)$
GROUP BY $?x, ?y$
HAVING $\text{count}(?xx) = 0$

Note that this technique of looking for the $(?x, ?y)$ groups with a zero count for $?xx$ is very similar to the technique used to express difference using a negated bound constraint (seen in the proof of Lemma 17).

7.2.3 BIND and VALUES

We have seen in Section 4.4 that allowing constant equalities in filter constraints allows us to emulate the difference operator. Two mechanisms introduced in SPARQL 1.1, BIND and VALUES, allow the introduction of constants in solution mappings. Together with equality constraints this allows us to express constant equalities, and hence, difference.

Specifically, using VALUES, we can express P FILTER $?x = c$ as

$$\text{SELECT}_{\text{var}(P)}(P \text{ AND VALUES}_{?x}(c)).$$

Using BIND, it can be expressed as

$$\text{SELECT}_{\text{var}(P)}((P \text{ BIND}_{?x'}\ (c)) \text{ FILTER } ?x = ?x')$$

where $?x'$ is a fresh variable. Note the use of SELECT, which, however, does not influence satisfiability as discussed above. We conclude that SPARQL(=) extended with BIND, or SPARQL(=) extended with VALUES, have an undecidable satisfiability problem.

8. Conclusion

The results of this paper may be summarized by saying that, as long as the kinds of constraints allowed in filter conditions cannot be combined to yield inconsistent sets of constraints, satisfiability for SPARQL patterns is decidable; otherwise, the problem is undecidable. Moreover, for well-designed patterns, satisfiability is decidable as well. All our positive results yield straightforward bottom-up syntactic checks that can be implemented in practice.

We thus have attempted to paint a rather complete picture of the satisfiability problem for SPARQL 1.0. Of course, satisfiability is only the most basic automated reasoning task. One may now move on to more complex tasks such as equivalence, implication, containment, or query answering over ontologies. Indeed, investigations along this line for limited fragments of SPARQL are already happening (Letelier, Pérez, Pichler, & Skritek, 2013; Wudage, Euzenat, Genevès, & Layaïda, 2012; Kollia & Glimm, 2013; Cuenca Grau, Motik, Stoilos, & Horrocks, 2012) and we hope that our work may serve to provide some additional grounding to these investigations.

We also note that in query optimization it is standard to check for satisfiability of subexpressions, to avoid executing useless code. Some specific works on SPARQL query optimization (Sequeda & Miranker, 2013; Groppe, Groppe, & Kolbaum, 2009) do mention that inconsistent constraints can cause unsatisfiability, but they have not provided sound and complete characterizations of satisfiability, like we have offered in this paper. Thus, our results will be useful in this direction as well.

Acknowledgment

We thank the anonymous referees, both on the original submission and on the revised submission, for their critical comments, which encouraged us to significantly improve the paper. This work has been funded by grant G.0489.10 of the Research Foundation Flanders (FWO).

Appendix A.

Literals in the wrong place in triple patterns are easily dealt with in the following manner. We define the *wrong-literal reduction* of a pattern P, denoted by $\lambda(P)$, as a set that is either empty or is a singleton containing a single pattern P':

- If P is a triple pattern (u, v, w) and u is a literal, then $\lambda(P) := \emptyset$; else $\lambda(P) := \{P\}$.
- $\lambda(P_1 \text{ UNION } P_2) := \lambda(P_1) \cup \lambda(P_2)$ if $\lambda(P_1)$ or $\lambda(P_2)$ is empty;
- $\lambda(P_1 \text{ UNION } P_2) := \{P_1' \text{ UNION } P_2' \mid P_1' \in \lambda(P_1) \text{ and } P_2' \in \lambda(P_2)\}$ otherwise.
- $\lambda(P_1 \text{ AND } P_2) := \{P_1' \text{ AND } P_2' \mid P_1' \in \lambda(P_1) \text{ and } P_2' \in \lambda(P_2)\}$.
- $\lambda(P_1 \text{ OPT } P_2) := \emptyset$ if $\lambda(P_1)$ is empty;
- $\lambda(P_1 \text{ OPT } P_2) := \lambda(P_1)$ if $\lambda(P_2)$ is empty but $\lambda(P_1)$ is nonempty;
- $\lambda(P_1 \text{ OPT } P_2) := \{P_1' \text{ OPT } P_2' \mid P_1' \in \lambda(P_1) \text{ and } P_2' \in \lambda(P_2)\}$ otherwise.
- $\lambda(P_1 \text{ FILTER } C) := \{P_1' \text{ FILTER } C \mid P_1' \in \lambda(P_1)\}$.

Note that the wrong-literal reduction never has a literal in the subject position of a triple pattern. The next proposition shows that, as far as satisfiability checking is concerned, we may always perform the wrong-literal reduction.

Proposition 24. *Let P be a pattern. If $\lambda(P)$ is empty then P is unsatisfiable; if $\lambda(P) = \{P'\}$ then P and P' are equivalent, i.e., $[\![P]\!]_G = [\![P']\!]_G$ for every RDF graph G. Moreover, if $\lambda(P) = \{P'\}$ then P' does not contain any triple pattern (u, v, w) where u is a literal.*

Proof. Assume P is a triple pattern (u, v, w) and u is a literal, so that $\lambda(P) = \emptyset$. Since u is a constant, $\mu(u)$ equals the literal u for every solution mapping μ. Since no triple in an RDF graph can have a literal in its first position, $[\![P]\!]_G$ is empty for every RDF graph G, i.e., P is unsatisfiable. If u is not a literal, $\lambda(P) = \{P\}$ and the claims of the Proposition are trivial.

If P is of the form $P_1 \text{ UNION } P_2$, or $P_1 \text{ AND } P_2$, or $P_1 \text{ FILTER } C$, the claims of the Proposition follow straightforwardly by induction.

If P is of the form $P_1 \text{ OPT } P_2$, there are three cases to consider.

- If $\lambda(P_1)$ is empty then so is $\lambda(P)$. In this case, by induction, P_1 is unsatisfiable, whence so is P.
- If $\lambda(P_1) = \{P_1'\}$ is nonempty but $\lambda(P_2)$ is empty, then $\lambda(P) = \{P_1'\}$. By induction, P_2 is unsatisfiable. Hence, P is equivalent to P_1, which in turn is equivalent to P_1' by induction. That P_1' does not contain any triple pattern with a literal in first position again follows by induction.
- If $\lambda(P_1) = \{P_1'\}$ and $\lambda(P_2) = \{P_2'\}$ are both nonempty, then $\lambda(P) = P_1' \text{ OPT } P_2'$. By induction, P_1 is equivalent to P_1' and so is P_2 to P_2'. Hence, P is equivalent to $P_1' \text{OPT} P_2'$ as desired. By induction, neither P_1' nor P_2' contain any triple pattern with a literal in first position, so neither does $P_1' \text{ OPT } P_2'$.

□

References

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases.* Addison-Wesley.

Andréka, H., Givant, S., & Németi, I. (1997). *Decision problems for equational theories of relational algebras*, Vol. 126 of *Memoirs.* AMS.

Angles, R., & Gutierrez, C. (2008). The expressive power of SPARQL. In Sheth, A., Staab, S., et al. (Eds.), *Proceedings 7th International Semantic Web Conference*, Vol. 5318 of *Lecture Notes in Computer Science*, pp. 114–129. Springer.

Arenas, M., Conca, S., & Pérez, J. (2012). Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In Mille, A., et al. (Eds.), *Proceedings 21st World Wide Web Conference*, pp. 629–638. ACM.

Arenas, M., & Pérez, J. (2011). Querying semantic web data with SPARQL. In *Proceedings 30st ACM Symposium on Principles of Databases*, pp. 305–316. ACM.

Arenas, M., Pérez, J., & Gutierrez, C. (2009). On the semantics of SPARQL. In De Virgilio, R., Giunchiglia, F., & Tanca, L. (Eds.), *Semantic Web Information Management—A Model-Based Perspective*, pp. 281–307. Springer.

Celko, J. (2005). *SQL for Smarties: Advanced SQL Programming* (Third edition). Elsevier.

Cuenca Grau, B., Motik, B., Stoilos, G., & Horrocks, I. (2012). Completeness guarantees for incomplete ontology reasoners: Theory and practice. *Journal of Artificial Intelligence Research*, *43*, 419–476.

Groppe, J., Groppe, S., & Kolbaum, J. (2009). Optimization of SPARQL by using coreSPARQL. In Cordeiro, J., & Filipe, J. (Eds.), *Proceedings 11th International Conference on Enterprise Information Systems*, pp. 107–112.

Han, X., Feng, Z., Zhang, X., Wang, X., Rao, G., & Jiang, S. (2016). On the statistical analysis of practical SPARQL patterns. In *Proceedings 19th International Workshop on the Web and Databases.*

Harris, S., & Seaborne, A. (2013). SPARQL 1.1 query language. W3C Recommendation.

Kollia, I., & Glimm, B. (2013). Optimizing SPARQL query answering over OWL ontologies. *Journal of Artificial Intelligence Research*, *48*, 253–303.

Kostylev, E., Reutter, J., Romero, M., & Vrgoč, D. (2015). SPARQL with property paths. In Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., et al. (Eds.), *Proceedings 14th International Semantic Web Conference*, Vol. 9366 of *Lecture Notes in Computer Science*, pp. 3–18. Springer.

Kostylev, E., Reutter, J., & Vrgoč, D. (2014). Containment of data graph queries. In *Proceedings 17th International Conference on Database Theory.* ACM.

Kroening, D., & Strichman, O. (2008). *Decision Procedures.* Springer.

Letelier, A., Pérez, J., Pichler, R., & Skritek, S. (2013). Static analysis and optimization of semantic web queries. *ACM Transactions on Database Systems*, *38*(4), article 25.

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, *34*(3), article 16.

Picalausa, F., & Vansummeren, S. (2011). What are real SPARQL queries like?. In De Virgilio, R., Giunchiglia, F., & Tanca, L. (Eds.), *Proceedings International Workshop on Semantic Web Information Management*, No. 7. ACM Press.

Polleres, A. (2007). From SPARQL to rules (and back). In Williamson, C., Zurko, M., et al. (Eds.), *Proceedings 16th World Wide Web Conference*, pp. 787–796. ACM.

Prud'hommeaux, E., & Seaborne, A. (2008). SPARQL query language for RDF. W3C Recommendation.

Schmidt, M., Meier, M., & Lausen, G. (2010). Foundations of SPARQL query optimization. In *Proceedings 13th International Conference on Database Theory*, pp. 4–33. ACM.

Sequeda, J., & Miranker, D. (2013). Ultrawrap: SPARQL execution on relational data. *Web Semantics*, *22*, 19–39.

Tan, T., Van den Bussche, J., & Zhang, X. (2014). Undecidability of satisfiability in the algebra of finite binary relations with union, composition, and difference. arXiv:1406.0349.

Wudage, M., Euzenat, J., Genevès, P., & Layaïda, N. (2012). SPARQL query containment under SHI axioms. In *Proceedings 26th AAAI Conference on Artificial Intelligence*, pp. 10–16.