Detecting and Diagnosing Syntactic and Semantic Errors in SPARQL Queries

Jesús M. Almendros-Jiménez
Dept. of Informatics
University of Almería, Spain
jalmen@ual.es

Antonio Becerra-Terón
Dept. of Informatics
University of Almería, Spain
abecerra@ual.es

Alfredo Cuzzocrea
DIA Department
University of Trieste and
ICAR-CNR, Italy
alfredo.cuzzocrea@dia.units.it

ABSTRACT

In this paper we present a tool to syntactically and semantically validate SPARQL queries. With this aim, we extract triple patterns and filter conditions from SPARQL queries and we use the OWL API and an OWL ontology reasoner in order to detect wrong expressions. Given an ontology and a query, the tool reports different kinds of programming errors: wrong use of vocabulary, wrong use of resources and literals, wrong filter conditions and wrong use of variables in triple patterns and filter conditions. When the OWL ontology reasoner is used the tool reports a diagnosis.

Keywords

SPARQL; RDF; OWL; Debbuging

1. INTRODUCTION

The Semantic Web has adopted *SPARQL* [6] as query language. While SPARQL queries are usually simple, and the SPARQL (SQL-like) syntax can be easily learned, it does not mean that SPARQL programmers can make no mistakes. It can happen for several reasons.

Firstly, from the database perspective, SPARQL data have (in most of cases) a sophisticated schema. The schema defines an ontology of concepts and relationships modeled in *RDF* and *OWL*. RDF data have a simpler schema, but the RDF model is a graph, and the programmer needs to work with paths and nodes from this graph. However, the programmers are more used to handle table-like data structures, typical in relational databases and SQL data. OWL data (RDF-like enriched data) have a very rich data schema, and OWL data handle concepts and properties, representing relationships. Concept definitions in OWL can be sophisticated, involving complex subconcept and equivalence relationships, and property definitions can be equipped with a rich semantics, enabling inverse, (a)symmetric, (ir)reflexive, (non) functional and transitive relationships. Secondly, the SPARQL programmer can ignore the (complete) structure of data, or even knowing this structure, he/she can be frustrated when the answer is empty or incomplete, mainly due to filter conditions that make the query unsatisfiable or too restrictive. Query unsatisfiability can be also consequence of the rich mechanisms for defining concepts and properties, where the program can require inconsistent queries. A query can be called *inconsistent* when matching of variables is incompatible with ontology consistency. In general, the *missing-answer problem* is a consequence of query unsatisfiability and, in particular, of query inconsistency. Finally, analyzing the main SPARQL implementations we have found that debugging mechanisms are missing. SPARQL interpreters are usually equipped with syntactic analysis limited to grammar checking.

SPARQL programming errors can be due to several reasons. Firstly, queries can include *wrongly typed expressions* which mainly occur in triple patterns. The most typical case is when triple patterns are incorrectly instantiated:

```
?x :age "Alice"
```

Here *age* has range *xsd:integer*, and thus *"Alice"* is not a suitable value. Combining triple patterns more typing errors can be found. For instance:

```
?x :father ?y . ?x :born ?y
```

in which assuming that *father* range is *person*, and *born* range is *country*, then *?y* cannot be bounded, and thus the answer will be empty. This also happens when variables are bounded to *literals* and *resources* at the same time:

```
?x rdf:type :human . ?y :age ?x
```

However, there are cases not forbidden (in general) by ontologies. The same resource can be both an individual and a concept. Thus, the following pattern is well-typed:

```
?x rdf:type ?x
```

Not only variables can be wrongly typed, also concepts, properties and individuals can occur in wrong positions. For instance, with RDF and OWL vocabularies we can express queries like the following:

```
?x rdf:type rdf:type . ?x owl:SymmetricProperty ?y
```

Unfortunately, existing SPARQL implementations do not check types for triple patterns. In fact, they do not check the vocabulary of concepts, properties and individuals either, and wrong spelling (for instance "fahter" instead of "father") leads to empty answers without any warning:

```
?x :fahter ?y
```

Some cases correspond to *inconsistent queries*, in which matching of variables is incompatible with ontology consistency. Inconsistent queries are well-typed but variable binding is not possible from the ontology definition. For instance, assuming *father* relation is irreflexive, a wrong triple pattern is:

```
?x :father ?x
```

Even when answering this question against a consistent ontology is not possible, and thus the answer will be empty, existing SPARQL implementations are not able to detect it. In some cases, the inconsistency is not trivial and ontology reasoning has to be used. For instance, let us consider the following triples:

```
?x :father ?y . ?y rdf:type ?z .
  ?z rdfs:subClassOf :motorcycle
```

Here, in order to detect the inconsistency of the query, we have to reason that *father* has range *person* and *person* is not a subclass of *motorcycle*. This is also the case of the following triples:

```
?x rdf:type :mammal . ?x rdf:type :motorcycle
```

in which *mammal* and *motorcycle* are disjoint and, therefore, the user should be warned. Some cases depend on the cardinality, for instance:

```
?x father ?y . ?x father ?z . ?y age 30 . ?z age 45
```

Here *father* and *age* properties are functional, and thus none of the variables can be bound. More complex cases involve filter conditions, for instance:

```
SELECT ?x
  WHERE { ?x :father ?y . ?x :father ?z
        FILTER (?y != ?z) }
```

in which the Boolean expression *?y != ?x* contradicts the functional property of *father*.

In this paper, we present a tool to validate SPARQL queries. With this aim, we extract triple patterns and filter conditions from SPARQL queries and we use the OWL API and an OWL ontology reasoner in order to detect wrong expressions. Given an ontology and a query, the tool reports different kinds of programming errors: wrong use of vocabulary, wrong use of resources and literals, wrong filter conditions and wrong use of variables in triple patterns and filter conditions. When the OWL ontology reasoner is used, the tool reports a diagnosis. The tool has been implemented in *XQuery*, and it uses the *SPARQL to SPIN*[1] transformation to get the SPARQL code in XML format. Once the code is transformed, an XQuery function traverses the SPARQL code in order to extract triple patterns and filters. Next, the validation process is carried out by calling the *OWL API* as well as the OWL reasoner HermiT from XQuery, which reports a diagnosis of inconsistent queries. The current implementation covers the cases of triple patterns and filters in SELECT, ASK, CONSTRUCT, DESCRIBE, and OPTIONAL triple patterns.

While programming errors have been studied in SQL [1, 4, 2], as far as we know, the same topic has not studied for SPARQL yet, (except in a recent work [8] in which authors study satisfiability of FILTER conditions). We have tested the best state-of-art SPARQL implementations (see Table 1) and we have found that they are not equipped with type checking and debugging facilities. Only benchmarking datasets [7, 3] are publicly available to analyze the performance of SPARQL implementations. There are also works about analysis of data [5], but less attention has been paid to SPARQL code. Thus, our work opens up a promising line of research, and our work can be seen as a first approximation to the solution.

[1]http://spinrdf.org/

2. VALIDATION OF SPARQL QUERIES

For the validation process, we distinguish between *syntactic* and *semantic* validation, by using the OWL API and the OWL Reasoner, respectively. Type checking is carried out by the OWL API and the OWL Reasoner, and thus it can be considered as both syntactic and semantic checking. Inconsistency of queries is only detected by the OWL Reasoner, and thus it is considered as semantic checking.

2.1 Syntactic Validation of SPARQL Queries

The syntactic validation uses the OWL API in order to carry out the following kinds of validation: **(a)** *Wrong use of vocabulary*, **(b)** *Wrong use of resources and literals* and **(c)** *Wrong filter conditions*. We analyze triple patterns *s p o*, where *s* is a subject, *p* is a property, and *o* is an object. In addition, we analyze filter conditions *l op r*, where *op* can be one of $<$, $>$, $>=$, $<=$, $=$, $!=$, and *l* and *r* are the left and right hand side of the operation, respectively. We assume that individuals, properties and concepts can share the same name, and also that object and data properties can have the same name[2]. In order to carry out the syntactic validation, we propose the following rules, expressing the cases in which a syntactic error (i.e. **(a)**, **(b)** and **(c)**) is found. We also assume an input ontology with namespace *ns*. Finally, except when it is specified, *s*, *p* and *o* (and *l*, *r*) can be variables or ontology items (i.e., individuals, properties and concepts).

Syn-1**(a)** *s p ns:k* where *k* does not belong to *ns* vocabulary. *ns* can be also *rdf/rdfs/owl*.

Syn-2**(a)** *s ns:k o* where *k* is not a property of *ns*. *ns* can be also *rdf/rdfs/owl*.

Syn-3**(a)** *ns:k p o* where *k* does not belong to *ns* vocabulary. *ns* can be also *rdf/rdfs/owl*.

Syn-4**(b)** *s ns:k o* where *o* is a resource, *k* is a data property and it is not an object property of *ns*.

Syn-5**(b)** *s ns:k o* where *o* is a literal, and *k* is not a data property and it is an object property of *ns*.

Syn-6**(b)** *s p o* where *p* is a literal.

Syn-7**(b)** *s p o* where *s* is a literal.

Syn-8**(c)** *l op r*, where *l* and *r* are literals of different type.

The syntactic rules can be checked by using the ontology signature: the full vocabulary (rules Syn-1, Syn-3), the property vocabulary (rules Syn-2, Syn-4, Syn-5), as well as the syntactic form of the triple pattern components (rules Syn-6, Syn-7 and Syn-8). Let us remark that Syn-6 and Syn-7 are already wrong according to the the SPARQL grammar (and thus they are usually checked by SPARQL implementations), but we have included them for completeness.

2.2 Semantic Validation of SPARQL Queries

The semantic validation uses an OWL ontology reasoner

[2]In general, ontology profiles do not consider object and data properties disjoint, while it happens in, for instance, OWL DL.

Name	*URL*
ARQ SPARQL Jena	https://jena.apache.org/documentation/query/
Protégé SPARQL Tab	http://protegewiki.stanford.edu/wiki/SPARQL_Query
Twinkle: SPARQL Tools	http://www.ldodds.com/projects/twinkle/
Virtuoso SPARQL Query Editor	http://dbpedia.org/sparql
SPARQLer - General purpose processor	http://www.sparql.org/sparql.html
Redland Rasqal RDF Query Demonstration	http://librdf.org/query
OpenUpLabs	http://openuplabs.tso.co.uk/sparql
wordnet.rkbexplorer.com	http://wordnet.rkbexplorer.com/sparql/
DBPedia SNORQL	http://dbpedia.org/snorql/
SPARQL editor | The British National Bibliography	http://bnb.data.bl.uk/flint-sparql
YASGUI SPARQL Editor	http://cliopatria.swi-prolog.org/yasgui/index.html
SPARQL Carsten Editor	http://sparql.carsten.io/

Table 1: SPARQL implementations

in order to detect wrongly typed and inconsistent queries. With this aim, the main idea is to consider the variables occurring in triples and filter conditions as individuals of the ontology to be queried. In other words, each variable *?x* becomes in *ns:x* where *ns* is the namespace of the ontology. Additionally triple patterns in which *?x* occurs, become property and concept assertions about *ns:x*, and filter conditions in which *?x* occurs become *owl:sameAs* and *owl:differentFrom* assertions. A new ontology is built from the original one in which these property and concept assertions, extracted from triple patterns and filter conditions, are added. Assuming the original ontology is consistent, the ontology reasoner is used to detect the consistency of this new ontology. In case the new ontology is inconsistent, the SPARQL query is wrongly typed or inconsistent.

Next, we will give rules for constructing these property and concept assertions from triple patterns and filter conditions. In order to use the ontology reasoner, there are additional modifications in the original ontology. (1) Firstly, concepts for which it is known that the intersection is empty have to be explicitly defined as disjoint. (2) Secondly, two additional concepts are included in the ontology: *DR* and *DT*. They represent ontology resources and literals (i.e., datatypes), respectively. They are declared as disjoint. *DT* has to be also disjoint with the rest of concepts. (3) Thirdly, additional concepts *DTinteger*, *DTstring*, etc., are included in the ontology for each datatype. They are declared as disjoint (except for compatible datatypes). They are defined as subclasses of *DT*.

Concept disjointness is extensively used by type checking and thus it is crucial to declare. In practice, it is enough to declare primitive sibling concepts as disjoint. Datatypes have to be converted into concepts (disjoint with the rest of concepts) in order to use the OWL Reasoner and detect wrongly typed expressions. Let us also remark that (2) and (3) do not depend on the ontology to be queried.

The semantic validation covers the following cases: **(a)** *Wrong use of variables in triple patterns* and **(b)** *Wrong use of variables in filter conditions*. We have the following (non overlapping) rules that express which concept and property assertions are added to the original ontology. We assume that triples are syntactically correct according to previous syntactic rules, and variables *?x* are always added as *ns:x* to the ontology. Except when it is specified, *s*, *p* and *o* (and *l*, *r*) can be variables or ontology items (i.e., individuals, properties and concepts).

Sem-1**(a)** *s ns:k o*, where *ns* is not *rdf/rdfs/owl* and *k* is an object property of *ns* (thus *o* is not a literal). In this case *s ns:k o* is added to the ontology.

Sem-2**(a)** *s ns:k ?w*, where *ns* is not *rdf/rdfs/owl*, *k* is a data property of *ns* and it is not an object property of *ns*. In this case *?w rdf:type DTt* is added to the ontology, for each range *t* of *ns:k*. Additionally *s rdf:type D* is added for each domain of *ns:k*.

Sem-3**(a)** *s ns:k l*, where *ns* is not *rdf/rdfs/owl*, and *l* is a literal (thus *ns:k* is a data property). In this case *s ns:k l* is added to the ontology.

Sem-4**(a)** *s ns:k o*, where *ns* is not *rdf/rdfs/owl*, and *k* is both an object and a data property of *ns* (*o* is not a literal, otherwise this is the previous case). In this case *s rdf:type D* is added for each domain of *ns:k*.

Sem-5**(a)** *s ns:k l*, where *ns* is *rdf/rdfs/owl* and *l* is a literal. In this case *s rdf:type DR* is added to the ontology.

Sem-6**(a)** *s ns:k ?w*, where *ns* is *rdf/rdfs/owl* and *k* is a data property of *ns*. In this case *s rdf:type DR* and *?w rdf:type DT* are added to the ontology.

Sem-7**(a)** *s ns:k o*, where *ns* is *rdf/rdfs/owl*, and *k* is an object property of *ns*. In this case *s rdf:type DR* and *o rdf:type DR* are added to the ontology.

Sem-8**(a)** *s ?v o*. In this case *?v rdf:type DR* is added to the ontology.

Sem-9**(b)** *?l op v* where *v* is a literal of type *t*. In this case *?l rdf:type Dt* is added to the ontology.

Sem-10**(b)** *v op ?r* where *v* is a literal of type *t*. In this case *?r rdf:type Dt* is added to the ontology.

Sem-11**(b)** *?l = ?r*. In this case *?l owl:sameAs ?r* is added to the ontology.

Sem-12**(b)** *?l != ?r*. In this case *?l owl:differentFrom ?r* is added to the ontology.

Sem-13**(b)** *?l op ?r*, where *op* is one of >,>=,<,<=. In this case *?l rdf:type DT* and *?r rdf:type DT* are added to the ontology.

Sem-1 is the most used triple pattern. In this case either *o* is *?w* or an ontology item *ns:i*. In the first case, *?w* is added as resource and, in the second one, *ns:i* is (again) added as resource. In the case Sem-2, given that *?w* will be bounded to a literal, the type *?w* as well as the type of *s* are added. Each range of *k* to *?w* and each domain of *k* to *s* are added. Sem-3 is similar to the case Sem-1, adding the literal as value of the property *k*. When *k* is both data and object property (i.e., case Sem-4) the type of *?w* cannot be fixed, and thus only the domain of *k* to *s* is added. The special cases of *rdf/rdfs/owl* are handled by rules Sem-5, Sem-6 and Sem-7. In this case terminological axioms are not added to the ontology. Otherwise, the original semantics of the ontology will be lost. Here the membership to *DR* and *DT* is used. In the case Sem-8, *o* can be a resource or a literal, but none of them gives additional information for *?v*, since even when *o*

is a literal, *?v* can be an object or a data property, because object and data properties are not necessarily disjoint. In case of filter conditions in which one of the sides is a literal (Sem-9 and Sem-10), then the type of this literal is added to the ontology for the other side. In the case of equalities (Sem-11), *owl:sameAs* is used to identify elements in the ontology. In case of inequalities (Sem-12), *owl:differentFrom* is used. Finally, in the case of numeric operators (Sem-13), *DT* is added as type of both sides.

3. EXAMPLES

The following examples are cases of syntactically wrong triple patterns and filter conditions for the ontology example.

```
(1) ?x :agex ?y
(2) ?x :age :jesus
(3) ?x :father 10
(4) FILTER (10 = "Alice")
```

The tool reports the following answers when the queries are validated:

```
(1) The property 'agex' does not exists
(2) The property 'age' is not an object property
(3) The property 'father' is not a data property
(4) Wrong types in filter: string and integer
```

Case (1) illustrates the rule Syn-1 while (2) illustrates Syn-4. The case (3) illustrates Syn-5, and case (4) illustrates the rule Syn-8. (1), (2) and (3) use OWL API, while (4) can be checked from the SPIN representation.

With regard to semantic errors, we illustrate the rule Sem-1, with the following example of a SPARQL query:

```
SELECT ?x
WHERE { ?x :father ?x }
```

Here *father* is an object property and *?x* is a variable. Therefore the triple *?x :father ?x* is added to the ontology. Since *father* is irreflexive, it causes inconsistency, and the tool (via the OWL reasoner) answers as follows:

```
Inconsistent query:
IrreflexiveObjectProperty(#father)
```

The rule Sem-1 is also applied to the following query:

```
SELECT ?x
WHERE { ?x :father ?y . ?x :born ?y }
```

in which *?y* should be a person and and a country at the same time. The tool answers (via the OWL reasoner) as follows:

```
Inconsistent query:
DisjointClasses(#Person #DT #Country)
ObjectPropertyDomain(#father #Person)
ObjectPropertyRange(#born #Country)
```

Let us now consider the following example:

```
SELECT ?x
  WHERE { ?x :age ?y . FILTER(?y = 'Alice')}
```

which illustrates the rules Sem-2 and Sem-9. Here *?y* is a variable in the range of a data property (which is not an object property), and each range of the data property is used for typing *?y*, by rule Sem-2. In this case, *?y rdf:type :DTinteger* is added to the ontology, (also *?x rdf:type Person* is added). Now, by rule Sem-9, from *?y='Alice'*, then *?y rdf:type :DTstring* is also added to the ontology. Since *DTinteger* and *DTstring* are disjoint the tool answers as follows:

```
Inconsistent query:
DisjointClasses(#DTinteger #DTstring)
```

Let us now consider the following ASK query in order to illustrate the rule Sem-3:

```
ASK  { :james :age 20 }
```

Here, we would like to know whether *james*'s age is *20*. Let us suppose that *james*'s *age* has been set to *45* by the ontology. Since *age* is a functional property, then the tool will answer as follows, due to *:james :age 20* has been added to the ontology by rule Sem-3:

```
Inconsistent query:
FunctionalDataProperty(#age)
```

Let us now consider the following example in which rules Sem-7 and Sem-9 are applied. Since *?y* is used in the place of a resource, then it cannot be used in a filter condition with an integer.

```
SELECT ?x
  WHERE { ?x rdf:type ?y . FILTER (?y = 10)}
```

The tool works as follows. From the condition *?y=10*, rule Sem-9 adds *?y rdf:type :DTinteger* to the ontology; and from *?x rdf:type ?y*, rule Sem-7 adds *?y rdf:type :DR* to the ontology. Now, the answer of the tool is as follows:

```
Inconsistent query:
DisjointClasses(#DR #DT)
SubClassOf(#DTinteger #DT)
```

With regard to wrong use of variables in property positions, rules Sem-2 and Sem-8 allow us to detect the following inconsistent query:

```
SELECT ?x
WHERE { ?x ?z ?y . ?t :age ?z }
```

since *?z rdf:type :DTinteger* and *?z rdf:type :DR* are added to the ontology. Here, the answer of the tool is as follows:

```
Inconsistent query:
DisjointClasses(#DR #DT)
SubClassOf(#DTinteger #DT)
```

With regard to rule Sem-11, we can consider the following example of SPARQL query:

```
SELECT ?x
  WHERE { ?y :father ?z  . FILTER (?y = ?z) . FILTER
       (?y != ?z) }
```

Here, the filter conditions are incompatible. In this case, *?y owl:sameAs ?z* and *?y owl:differentFrom ?z* are added to the ontology by rules Sem-11 and Sem-12, respectively. In this case, the answer of the tool is as follows:

```
Inconsistent query:
SameIndividual(#y #z )
DifferentIndividuals(#y #z)
```

The following example also illustrates rule Sem-12, in which *?y owl:differentFrom ?z* is added for the following query:

```
SELECT ?x
  WHERE { ?x :father ?y  .
    ?x :father ?z . FILTER (?y != ?z) }
```

The answer of the tool is as follows:

```
Inconsistent query:
DifferentIndividuals(#y #z )
FunctionalObjectProperty(#father)
```

Finally, rule Sem-13 is illustrated by the following query, in which *?y* and *?z* are compared by > and thus both ones have type *DT* which is incompatible with *Person* (i.e., the range of *father*):

```
SELECT ?x
  WHERE { ?x :age ?y  . ?u :father ?z .
    FILTER (?y > ?z) }

Inconsistent query:
DisjointClasses(#Person #DT)
ObjectPropertyRange(#father #Person)
```

There are some cases in which inconsistency cannot be detected. For instance, let us suppose the following query:

```
SELECT ?x
  WHERE { ?x :age ?y . ?x :age ?z .
    FILTER (?y != ?z) }
```

Here, even knowing that *age* is a functional property, we cannot detect with the equality *?y != ?z* that it cannot be answered. It is due that *owl:differentFrom* cannot be used for literals in the OWL reasoner.

Acknowledgements

This work was supported by the EU (FEDER) and the Spanish MINECO Ministry (Ministerio de Economía y Competitividad) under grant CAVI-TEXTUAL TIN2013-44742-C4- 4-R.

4. CONCLUSIONS AND FUTURE WORK

We have designed a tool for detecting and diagnosing wrong SPARQL queries. A set of rules has been defined in order to use the OWL API and an OWL reasoner to check wrongly typed and inconsistent queries, reporting the details of the diagnosis. The first question arising is the completeness of the method. The defined rules cover a wide number of cases, but a deeper study of completeness is required. In particular, the last example of previous Section shows a limitation of the approach. This limitation is imposed by the ontology reasoner. There are also other limitations in filter conditions. For instance, let us suppose that adult class is defined as subclass of person (i.e., older than 18), and a filter condition of a SPARQL query requests adults users whose age is smaller than 10. In this case, the query is inconsistent with the ontology, but it cannot still be detected. We have only considered the case of equalities and inequalities in FILTER conditions and sameAs/differentFrom to check them, but more cases of filter conditions[3] can be considered. We would like to implement a Java version of our tool, integrated with Jena ARQ SPARQL Engine. We also plan to implement a Web tool for validating SPARQL code in which the ontology URI or SPARQL endpoint can be specified.

5. REFERENCES

[1] Stefan Brass and Christian Goldberg. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software*, 79(5):630–644, 2006.

[2] Benjamin Dietrich and Torsten Grust. A SQL Debugger Built from Spare Parts: Turning a SQL: 1999 Database System into Its Own Debugger. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 865–870. ACM, 2015.

[3] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[3]https://www.w3.org/TR/rdf-sparql-query/

[4] Muhammad Akhter Javid and Suzanne M Embury. Diagnosing faults in embedded queries in database applications. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 239–244. ACM, 2012.

[5] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. Test-driven evaluation of linked data quality. In *Proceedings of the 23rd international conference on World Wide Web*, pages 747–758. ACM, 2014.

[6] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[7] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^ 2Bench: a SPARQL performance benchmark. In *2009 IEEE 25th International Conference on Data Engineering*, pages 222–233. IEEE, 2009.

[8] Xiaowang Zhang, Van Den Bussche Jan, and Francois Picalausa. On the satisfiability problem for SPARQL patterns. *J. Artif. Int. Res*, 56(1):403–428, 2016.