Introducing Wild-card and Negation for Optimizing SPARQL Queries based on Rewriting RDF Graph and SPARQL Queries

Faisal Alkhateeb
Department of Computer Science, Jordan University of Science and Technology, Irbid, 22110, Jordan
Department of Computer Science, Yarmouk University, Irbid, 21163, Jordan

Keywords: RDF, SPARQL, Negation, Wild-card, Querying RDF and RDFS Graphs, Query Optimization.

Abstract: In this paper, we extend the SPARQL triple patterns to include two operators (the negation and the wild-card). We define the syntax and the semantics of these operators, in particular, when using them in the predicate position of SPARQL triple patterns. The use of the negation and wild-card operators and thus the semantics are different from the literature. Then, we show that these two operators could be used to enhance the evaluation performance of some SPARQL queries and to add extra expressiveness.

1 INTRODUCTION

RDF (Resource Description Framework (Manola and Miller, 2004)) is the most used language for the representation of knowledge and the description of documents within the semantic web.

SPARQL is the standard query language for RDF recommended by the W3C (Prud'hommeaux and Seaborne, 2008). SPARQL allows the use of negation operator in the Filter constraints (i.e., it is restricted to syntactic matching using predefined functions and thus applying post filtering). An extension of SPARQL, called PSPARQL have been developed in (Alkhateeb et al., 2009). It adds path expressions to SPARQL. The negation operator is initially introduced in the syntax of path expressions of PSPARQL (see some examples and discussions in (Alkhateeb, 2008)) for querying RDF but without defining its semantics. It is shown in (Alkhateeb et al., 2008) that answering SPARQL queries modulo RDF Schema could be achieved by transforming them into PSPARQL queries. Later on it is used in other path expressions (such as (Zauner et al., 2010; Fionda et al., 2015) and SPARQL 1.1 (Harris and Seaborne, 2013)) without entailment (Glimm and Ogbuji, 2013), i.e., syntactic matching of path expressions over property paths of URIs.

The wild-card is introduced in path query languages (Alechina et al., 2003) for querying semi-structured data that replaces a whole predicate. To our knowledge, there is no RDF query language that uses wild-card, in particular, to replace a fragment part of a predicate URI (i.e., matching all predicates defined in the same namespace or URI prefix). In addition, the use of wild-card operator can be implemented using the `fn:strstarts` using FILTER constraints. However, it is better to evaluate constraints on the fly and not as a post processing phase as shown in (Alkhateeb, 2008).

Moreover, none of these languages use the two operators (the negation and the wild-card) in combination with each other and thus the semantics is not defined for the combination.

The contributions in this paper are: the use of these two operators in a different way than that in the literature; the definition of their semantics and a method for answering queries containing them over RDF graphs; and their exploitation for optimizing queries over RDF and RDFS graphs.

Paper Outline. The remainder of the paper is organized as follows. In Section 2, we introduce the RDF language and the SPARQL language. The proposed extension is presented in Section 3. We propose an approach for optimizing the evaluation of queries over RDF graphs considering RDFS semantics in Section 4.1. Finally, we conclude in Section 5.

2 PRELIMINARIES

2.1 RDF

RDF graphs are constructed over sets of URI references (or urirefs), blanks, and literals (Carroll and Klyne, 2004). The union of these three sets are called

Alkhateeb, F.
Introducing Wild-card and Negation for Optimizing SPARQL Queries based on Rewriting RDF Graph and SPARQL Queries.
In *Proceedings of the 12th International Conference on Web Information Systems and Technologies (WEBIST 2016) - Volume 2*, pages 181-187
ISBN: 978-989-758-186-1

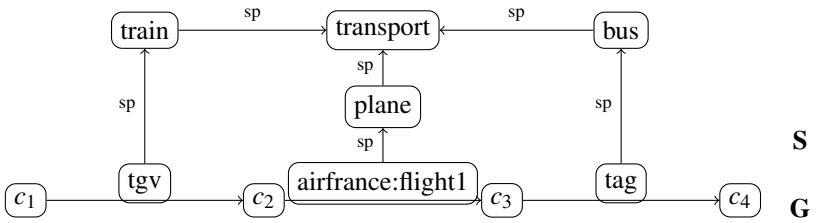

Figure 1: An RDF graph (G) with its schema representing information about transportation means between several cities.

the RDF terminology.

Terminology. An *RDF terminology*, noted $\mathcal{T}$, is the union of 3 pairwise disjoint infinite sets of *terms*: the set $\mathcal{U}$ of *urirefs*, the set $\mathcal{L}$ of *literals* and the set $\mathcal{B}$ of *blanks*. The *vocabulary* $\mathcal{V}$ denotes the set of *names*, i.e., $\mathcal{V} = \mathcal{U} \cup \mathcal{L}$. We use the following notations for the elements of these sets: a blank will be prefixed by _:, a literal will be expressed between quotation marks, remaining elements will be urirefs.

This terminology grounds the definition of RDF graphs.

Definition 1 (RDF graph)**.** *An* RDF triple *is an element of* $\mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$. *An* RDF graph *is a set of RDF triples.*

For simplicity and without loss of generality, we do not consider blank nodes in the above definition of RDF graphs since they could simply be treated as constants (as if they were URIs) without considering their existential semantics.

Example 1 (RDF graph)**.** *RDF can be used for representing information about cities, transportation means between cities, and relationships between the transportation means (see Figure 1).*

For instance, a triple ⟨b, plane, c⟩ *means that there exists a transportation mean* plane *from* b *to* c.

From now, in a triple $t = \langle$b, plane, c$\rangle$, b is called the *subject*, plane is the *predicate* and c is the *object* of the triple t. Also, ⟨tgv sp train⟩ means that tgv is a sub-property of train.

2.2 SPARQL

For a complete description of SPARQL, the reader is referred to the SPARQL specification (Prud'hommeaux and Seaborne, 2008) or to (Pérez et al., 2009; Polleres, 2007) for its formal semantics.

2.2.1 SPARQL Syntax

The basic building blocks of SPARQL queries are *graph patterns* which are shared by all SPARQL query forms. Informally, a *graph pattern* can be a triple pattern, a basic graph pattern (i.e., a set of triple patterns), a union of graph patterns, an optional graph pattern, or a constraint. Let $\mathcal{X}$ be the set of *variables*.

Definition 2 (SPARQL Triple Pattern)**.** *A* SPARQL triple pattern *is a set of triples of* $(\mathcal{U} \cup \mathcal{X}) \times (\mathcal{U} \cup \mathcal{X}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{X})$.

In the following, a variable will be prefixed by ? (like ?C_1).

Example 2 (SPARQL Triple Pattern)**.** *The following is a SPARQL triple pattern:*

⟨?city1 train ?city2⟩

that could be used to retrieve pairs of cities connected by a train *relation.*

Definition 3 (SPARQL Graph Pattern)**.** *A* SPARQL graph pattern *is defined inductively in the following way:*

- *every SPARQL triple pattern is a SPARQL graph pattern;*
- *if P, P′ are SPARQL graph patterns and K is a SPARQL constraint, then (P* AND *P′), (P* UNION *P′), (P* OPT *P′), and (P* FILTER *K) are SPARQL graph patterns.*

A SPARQL constraint K is a boolean expression involving terms from $(\mathcal{V} \cup \mathcal{X})$, e.g., a numeric test. We do not specify these expressions further (see (Muñoz et al., 2009) for a more complete treatment).

Example 3 (SPARQL Graph Pattern)**.** *The following SPARQL graph pattern:*

```
{
  ?city1  ?p  ?city2 .
  Filter (NOT (regex(?p, "bus")))
  }
```

could be used to retrieve pairs of cities connected by any relation that is not a bus. Note that the negation operator is used in the above query in the Filter constraint for post-filtering the values of ?p after finding all matches of ?p in the triple pattern.

2.2.2 SPARQL Semantics

In the following, we characterize SPARQL query answering using maps. More precisely, we use SPARQL queries restricted to conjunction of triple patterns and SPARQL FILTER constraints and the reader may

refer to (Pérez et al., 2006a) for other compound SPARQL graph patterns).

Definition 4 (Map)**.** *Let $V_1 \subseteq (\mathcal{T} \cup \mathcal{X})$, and $V_2 \subseteq \mathcal{T}$ be two sets of terms. A map from V_1 to V_2 is a function $\mu : V_1 \to V_2$ such that $\forall x \in (V_1 \cap \mathcal{V})$, $\mu(x) = x$.*

If μ is a map, then the domain of μ, denoted by $dom(\mu)$, is the subset of $\mathcal{X}$ on which μ is defined. The restriction of μ to a set of terms X is defined by $\mu|_X = \{\langle x,y\rangle \in \mu |\ x \in X\}$. If P is a SPARQL graph pattern, then we use $\mathcal{X}(P)$ to denote the set of variables occurring in P and $\mu(P)$ to denote the graph pattern obtained by the substitution of $\mu(b)$ to each variable $b \in \mathcal{X}(P)$.

Definition 5 (Answers to SPARQL Triple Pattern)**.** *Let P be a SPARQL triple pattern, K be a SPARQL constraint, and G be an RDF graph. The set $\mathcal{S}(P,G)$ of* answers *to P in G is defined in the following way:*

$$\mathcal{S}(P,G) = \{\mu|_{\mathcal{X}(P)} |\ \mu(P) \in G\}$$
$$\mathcal{S}(P \texttt{ FILTER } K, G) = \{\mu \in \mathcal{S}(P,G) \mid \mu(K) = \top\}$$

3 EXTENDED SPARQL TRIPLE PATTERNS

In this section, we define the syntax and the semantics of an extension to SPARQL triple patterns. The proposed extension includes two new operators (wild-card and negation) used in the predicate position of triple patterns.

3.1 Extended SPARQL Triple Patterns Syntax

SPARQL allows to use only variables or urirefs in the predicate position of triple patterns. In the proposed extension, the negation operator, the wild-card operator, or a combination of both of them could be used in the predicate position. In the following, we define all possible predicate expressions.

Definition 6 (Predicate Expression)**.** *The set of* predicate expressions *(denoted by $\mathcal{E}$) is defined by:*

- *if $a \in \mathcal{U}$ then $a \in \mathcal{E}$, $!a \in \mathcal{E}$, $a{:}\# \in \mathcal{E}$, $!(a{:}\#) \in \mathcal{E}$.*
- *if $x \in \mathcal{X}$, then $x \in \mathcal{E}$.*

Where # is the wild-card[1] *operator and ! is the negation operator.*

Definition 7 (Extended SPARQL Triple Pattern)**.** *An* Extended SPARQL triple pattern *is a set of triples of* $(\mathcal{U} \cup \mathcal{X}) \times \mathcal{E} \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{X})$.

[1]Note that the symbol ? could be used in place of the symbol #.

It should be noticed that the negation operator is initially introduced in the syntax of PSPARQL (Alkhateeb, 2008) for querying RDF but without defining its semantics. It is also used in other path expressions (such as RPL (Zauner et al., 2010) and SPARQL 1.1 (Harris and Seaborne, 2013)) without entailment (Glimm and Ogbuji, 2013).

The wild-card is introduced in (Alechina et al., 2003) for querying semi-structured data that replaces a whole predicate. SPARQL 1.1 allows the use of `elt?` in path expressions to match a path that connects a subject and object of the path by zero or one matches of `elt`. To our knowledge, there is no RDF query language that uses wild-card, in particular, to replace a fragment of a predicate (as illustrated in the following examples). Moreover, none of these languages use the two operators (the negation and the wild-card) in combination with each other and thus the semantics is not defined for the combination.

Example 4 (Extended SPARQL Triple Pattern)**.** *The following extended SPARQL triple:*

`⟨?city1 airfrance:# ?city2⟩`

could be basically used to retrieve all pairs of cities connected by any property that belongs to the `airfrance`*. Contrary, the following extended SPARQL triple could be used to retrieve all pairs of cities connected by any property that does not belong to the* `airfrance`*:*

`⟨?city1 !(airfrance:#) ?city2⟩`

3.2 Extended SPARQL Triple Patterns Semantics

The semantics of a predicate expression E in an RDF graph G is defined by a set of pairs of terms in G satisfying E.

Definition 8 (Extended Semantics: Simple)**.** *Given a predicate expression $E \in \mathcal{E}$ and an RDF graph G, then the semantics of E (denoted by $[\![E]\!]_G$) in G is defined as:*

$$[\![!a]\!]_G = \{\langle x,y\rangle |\ \langle x,b,y\rangle \in G \wedge b \neq a\}$$
$$[\![a:\#]\!]_G = \{\langle x,y\rangle |\ \langle x,b,y\rangle \in G \wedge a \leq b\}$$
$$[\![!a:\#]\!]_G = \{\langle x,y\rangle |\ \langle x,b,y\rangle \in G \wedge a \not\leq b\}$$

where $a \leq b$ holds if the string b starts-with *the string a.*

Basically, the predicate expression $!a$ matches any property other than a; and $a:\#$ (respectively, $!a:\#$) matches any property that starts with a (respectively, matches any property that does not starts with a).

For instance, $[\![\texttt{airfrance:\#}]\!]_G$ over the graph G of Figure 1 is $\{\langle c_2, c_3\rangle\}$, $[\![\texttt{bus}]\!]_G = \{\}$ and $[\![\texttt{!bus}]\!]_G = \{\langle c_1, c_2\rangle, \langle c_2, c_3\rangle, \langle c_3, c_4\rangle\}$.

The following definition generalizes Def. 8 for RDFS semantics (in particular, sub-property relationship).

Definition 9 (Extended Semantics: RDFS)**.** *Given a predicate expression $E \in \mathcal{E}$ and an RDF graph G, then the semantics of E in G is defined as:*

$$[\![!a]\!]_G^{rdfs} = \{\langle x,y\rangle |\ \langle x,b,y\rangle \in G \wedge b \not\preceq a\}$$
$$[\![a:\#]\!]_G^{rdfs} = \{\langle x,y\rangle |\ \langle x,b,y\rangle \in G \wedge (\exists p; b \preceq a:p)\}$$
$$[\![!a:\#]\!]_G^{rdfs} = \{\langle x,y\rangle |\ \langle x,b,y\rangle \in G \wedge (\nexists p; b \preceq a:p)\}$$

where $b \preceq a$ holds if b is a sub-property *of a*[2].

Based on the above definition, $[\![\texttt{bus:\#}]\!]_G^{rdfs} = \{\langle c_3, c_4\rangle\}$ and $[\![\texttt{!bus}]\!]_G^{rdfs} = \{\langle c_1, c_2\rangle, \langle c_2, c_3\rangle\}$ (Note that $\langle c_3, c_4\rangle$ is not in the result since `tag` is a sub-property of `bus`).

The following definition extends Def. 5 and uses Def. 8 (respectively, Def. 9 for RDFS semantics) to take into account predicate expressions including negation and wild-card operators.

Definition 10 (Answers to Extended SPARQL Triple Pattern)**.** *The evaluation of an extended SPARQL triple pattern $t = \langle S, E, O\rangle$ over an RDF graph G is defined as the following set of maps:*

$[\![t]\!]_G = \{\mu \mid dom(\mu) = \{S, O\} \cap \mathcal{X} \cup \mathcal{X}(E)$ *and* $\langle \mu(S), \mu(O)\rangle \in [\![\mu(E)]\!]_G$[3]$\}$ *such that $\mu(E)$ is the predicate expression obtained by substituting the variable ?x appearing in E by $\mu(?x)$.*

Hence, $[\![\langle \texttt{?city1 (airfrance:\#)* ?city2} \rangle]\!]_G = \{\langle ?city1 \leftarrow c_2, ?city1 \leftarrow c_3\rangle\}$.

4 QUERYING RDF AND RDFS GRAPHS

In this section, we show how the wild-card and negation operators can be used not only for optimizing the evaluation of some queries but also for capturing extra expressiveness over RDF(S) graphs.

4.1 Querying RDF Graphs

We believe that the wild-card and negation operators can enhance the evaluation performance of some kind of SPARQL queries for querying RDF graphs since they allow filtering results on-the-fly (e.g., query of Example 4 matches only properties that belongs to airfrance).

However, to express the query of Example 4 in SPARQL 1.0 and SPARQL 1.1, a new variable should be introduced in the query and then doing post-filtering using filter constraint.

In the case of SPARQL 1.0, the operator `regex` allows for implementing the "starts with" function as shown in the following query.

```
SELECT ?city1, ?city2
FROM ...
WHERE {
  ?city1  ?newVar  ?city2 .
  Filter (regex(?newVar, "^airfrance"))
  }
```

In the case of SPARQL 1.1, the use of wild-card operator can be implemented using the `fn:strstarts` using FILTER constraints as shown in the following query.

```
SELECT ?city1, ?city2
FROM ...
WHERE {
  ?city1  ?newVar  ?city2 .
  Filter (fn:strstarts(str(?newVar),
                       "airfrance"))
  }
```

In both cases, variables are bound during query execution for the triple $\langle$`?city1 ?newVar ?city2`$\rangle$ and these instantiations are then tested with the specified function in the FILTER clause. In both cases also, a new variable is introduced (`?newVar`) and it is required to be used in the FILTER clause to constrain the results. As this variable is not used in the query of Example 4, the projection operator (SELECT clause) thus is needed to restrict the result (i.e., to restrict the result to only a set of values for the variables `?city1` and `?city2`).

Though the evaluation of SPARQL queries composed of AND and FILTER expressions (as the ones in the above queries) is PTIME (Pérez et al., 2006b; Schmidt et al., 2010), the evaluation of queries requiring the projection operator (SELECT clause) over RDF graphs will be NP-hard problem (see Section 7.1 in (Pérez et al., 2010) and (Chandra and Merlin, 1977) for the evaluation problem for relational conjunctive queries). Also, (Alkhateeb, 2008) demonstrated practically that evaluating queries with constraints on the fly is better than a post processing phase. Additionally, the constraints in the FILTER clause can be used for syntactic matching only while the negation and the wild-card operators can be used for RDFS sub-property matching. For instance, the following query:

$\langle$`?city1 !(transport:#) ?city2`$\rangle$

[2] a is neither an RDF(S) term nor a prefixed fragment of an RDF(S).

[3] (for RDFS $[\![\mu(E)]\!]_G^{rdfs}$)

when considering RDFS semantics (i.e., sub-property semantics) should retrieve pairs of cities connected by a property that is not a `transport` (not prefixed by `transport`) or any of its sub-properties. Implementing this query in SPARQL 1.1 (as the following one) may retrieve pairs of cities connected by a property that is a sub-property of `transport` (e.g., `bus`). Besides, there is no clear method on how to deal with this in SPARQL 1.0 and SPARQL 1.1 while we provide in Section 4.2 a detailed discussion and examples on how to use wild-card and negation to deal with sub-property matching.

```
SELECT ?city1, ?city2
FROM ...
WHERE {
  ?city1  ?newVar  ?city2 .
  Filter (NOT(fn:strstarts(str(?newVar),
                           "transport")))
  }
```

This case is even worse if one wants to retrieve pairs of cities connected by a sequence of flights belonging to `airfrance` company. This query could be expressed using wild-card as follows:

⟨?city1 (airfrance:#)* ?city2⟩

where * is the closure operator used in path expressions (e.g., regular expressions) to match a sequence of predicates (c.f. (Alkhateeb et al., 2009; Pérez et al., 2010)).

Nonetheless, without using wild-cards, a variable must be introduced with the closure operator to express the query and the evaluation of such queries could be intractable for large graphs in particular for finding *simple* paths (i.e., paths without loops) (Mendelzon and Wood, 1995; Arenas et al., 2012) .

The wild-card could be combined with the negation operator to provide more interesting queries as in the following simple one:

⟨?city1 !(airfrance:#) ?city2⟩

that retrieves pairs of cities connected by any property that is not belonging to `airfrance` company.

4.2 Querying RDFS Graphs

Evaluating SPARQL queries over RDF graphs considering RDFS semantics needs additional treatment either for the RDF graph to be queried (e.g., calculating the closure graph) or transforming the input query to some form of path queries (see (Alkhateeb et al., 2009; Pérez et al., 2010; Alkhateeb and Euzenat, 2014) for querying a core fragment of RDFS {`sc, sp, type, domain, range`}).

In this section, we propose a novel approach for evaluating SPARQL queries over RDFS graphs without the need to the closure operator *. In this approach, we rely on the fact that RDF Schema have usually small size with respect to the RDF data graph. The proposed approach consists of several steps that are presented in the following.

1. We first build an index (hash table) for the RDF Schema terms. Each entry in the index contains an RDFS term `t` (i.e., RDFS terms that are appearing in the RDF Schema of the RDF data graph) and the corresponding ancestor terms represented as a path from the root term to the parent term of `t` denoted by $\texttt{PA}_t$. Table 1 represents the index table for the RDF Schema terms of Figure 1. It contains two columns: one for the RDF Schema term and the other for the path ancestor term in the RDF Schema. For instance, the term `tag` has the path ancestor term `transport/bus` since there exists a path from `tag` to `transport` (i.e., ⟨`tag sp bus`⟩ and ⟨`bus sp transport`⟩).
2. Given an RDF data graph G, we construct an RDF graph G' in the following way; for every triple ⟨`s, p, o`⟩ $\in G$:
   - add ⟨`s`, $\texttt{PA}_p$`/p, o`⟩ to G', if `p` is a term appearing in the RDF Schema and `p` $\notin$ {`sc, sp, type, domain, range`}.
   - add ⟨`s, p, o`⟩ to G', otherwise;

   Graph G' in Figure 2 corresponds to the RDF graph of Figure 1, where ⟨c_3 `transport/bus/tag` c_4⟩ $\in G'$ since ⟨c_3 `tag` c_4⟩ $\in G$ and `transport/bus` is the path ancestor of `tag`.

   It should be noticed that G' is constructed one time and not for each query. It can be updated once the original RDF G is modified.
3. Given a SPARQL triple t = ⟨`s p o`⟩, t is transformed to t' = ⟨`s` $\texttt{PA}_p$`/p:# o`⟩.

   According to this step, the following SPARQL query:

   ⟨`?city1 transport ?city2`⟩

   is transformed to the following one t' using the index structure in Table 1:

   ⟨`?city1 transport:# ?city2`⟩

   Note that `transport` has no path of ancestor terms since it is a root term in the RDF Schema. Therefore, we just append the term with the wild-card operator. However, the SPARQL triple ⟨`?city1 bus ?city2`⟩ will be converted ⟨`?city1 transport/bus:# ?city2`⟩
4. Evaluate the triple t' over G'.

The same approach can be applied when using the negation operator ! in the triple patterns by applying

Table 1: Index structure for the terms of the RDF Schema of Figure 1.

RDF Schema Term	Path Ancestor Terms
transport	null
plane	transport
airfrance:flight1	transport/plane
bus	transport
tag	transport/bus
train	transport
tgv	transport/train

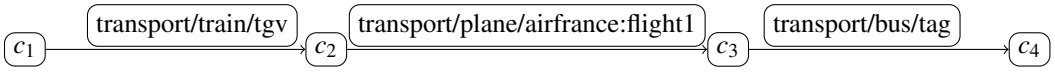

Figure 2: Transformed RDF graph (G') corresponding to the RDF graph in Figure 1.

the above steps and preceding the predicate by !. To illustrate this, consider the following triple t:

⟨?city1 !bus ?city2⟩

that should retrieve the set of pairs of citied connected by a property, which is not a bus. This query is transformed to the following one t':

⟨?city1 !(transport/bus:#) ?city2⟩

In this case, by evaluating the transformed triple t' over G', the pair $\langle c_3, c_4 \rangle$ will not be among the result. That is, $\langle c_3, c_4 \rangle$ is not an answer since tag is a bus (i.e., tag is a sp of bus).

SPARQL 1.1 (Glimm and Kroetzsch, 2010; Harris and Seaborne, 2013) used the negation operator in the property paths. However, property paths are evaluated without entailment (see Section 10.1 in (Glimm and Ogbuji, 2013)). Therefore, the query ⟨?city1 !bus ?city2⟩ when evaluated against G will retrieve $\langle c_3, c_4 \rangle$ as a pair in the result since syntactically speaking tag is not a bus.

5 DISCUSSION AND CONCLUSIONS

In this paper, we proposed an extension to SPARQL that introduces two operators (the negation and the wild-card) in SPARQL triple patterns. We have defined the syntax as well as the semantics of these operators, in particular, when using them in the predicate position of SPARQL triple patterns. Then, we have shown that these two operators could be used to enhance the evaluation performance of some SPARQL queries and to add extra expressiveness considering RDF(S) semantics. This is even more important for distributed path evaluation (Hartig and Pirrò, 2015).

As shown in the paper, the use of the negation and wild-card operators and thus the semantics are different from the literature. More precisely, as discussed in the paper, the negation operator is used in SPARQL 1.1 and other path languages without entailment regime. On the other hand, the proposed approach deals with RDFS semantics (considering sp relations). The proposed approach could be generalized to deal with sub-class (sc) relations by allowing the wild-card to be used in the place of subject and object of a triple (For instance, ⟨?C p c2:#⟩).

Additionally, the proposed extended SPARQL triple patterns can be seamlessly integrated in path languages that are built on top SPARQL triple patterns such as PSPARQL, cpSPARQL (Alkhateeb and Euzenat, 2014), SPARQL 1.1, and nSPARQL (Pérez et al., 2010). This will allow representing more complex queries like:

⟨?city1 next::plan/[self::!(airfrance:#)] ?city2⟩

that can be used to retrieve pairs of cities connected by a plane but is not belonging to airfrance.

REFERENCES

Alechina, N., Demri, S., and de Rijke, M. (2003). A modal perspective on path constraints. *Journal of Logic and Computation*, 13:1–18.

Alkhateeb, F. (2008). *Querying RDF(S) with regular expressions*. Thèse d'informatique, Université Joseph Fourier, Grenoble (FR).

Alkhateeb, F., Baget, J.-F., and Euzenat, J. (2008). Constrained regular expressions in SPARQL. In Arabnia, H. and Solo, A., editors, *Proceedings of the international conference on semantic web and web services (SWWS), Las Vegas (NV US)*, pages 91–99.

Alkhateeb, F., Baget, J.-F., and Euzenat, J. (2009). Extending SPARQL with regular expression patterns (for querying RDF). *Journal of web semantics*, 7(2):57–73.

Alkhateeb, F. and Euzenat, J. (2014). Constrained regular expressions for answering RDF-path queries modulo RDFS. *International journal of web information systems*, 10(1):24–50.

Arenas, M., Conca, S., and Pérez, J. (2012). Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 629–638.

Carroll, J. J. and Klyne, G. (2004). RDF concepts and abstract syntax. Recommendation, W3C.

Chandra, A. K. and Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM.

Fionda, V., Pirrò, G., and Consens, M. P. (2015). Extended property paths: writing more SPARQL queries in a succinct way. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Glimm, B. and Kroetzsch, M. (2010). SPARQL beyond subgraph matching. In *Proc. 9th International Semantic Web Conference (ISWC), Shanghai (CN)*, pages 59–66.

Glimm, B. and Ogbuji, C. (2013). SPARQL 1.1 entailment regimes. Recommendation, W3C.

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

Hartig, O. and Pirrò, G. (2015). A context-based semantics for sparql property paths over the web. In *The Semantic Web. Latest Advances and New Domains*, pages 71–87. Springer.

Manola, F. and Miller, E. (2004). RDF primer. Recommendation, W3C. http://www.w3.org/TR/REC-rdf-syntax/.

Mendelzon, A. and Wood, P. (1995). Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258.

Muñoz, S., Pérez, J., and Gutierrez, C. (2009). Simple and efficient minimal RDFS. *Journal of web semantics*, 7(3):220–234.

Pérez, J., Arenas, M., and Gutierrez, C. (2006a). Semantics and complexity of SPARQL. In *Proc. 5th International Semantic Web Conference (ISWC), Athens (GA US)*, pages 30–43.

Pérez, J., Arenas, M., and Gutierrez, C. (2006b). Semantics and complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference*, pages 30–43, Athens (GA US).

Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM transactions on database systems*, 34(3):16.

Pérez, J., Arenas, M., and Gutierrez, C. (2010). nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270. http://www.sciencedirect.com/science/article/B758F-4Y95V3X-1/2/9e5098d690fbe4d05a099f4c90a29a10.

Polleres, A. (2007). From SPARQL to rules (and back). In *Proc. 16th World Wide Web Conference (WWW)*, pages 787–796.

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

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

Zauner, H., Linse, B., Furche, T., and Bry, F. (2010). A RPL through RDF: Expressive navigation in RDF graphs. In *Proc. 4th International Conference on Web reasoning and rule systems (RR), Bressanone/Brixen (IT)*, volume 6333 of *Lecture Notes in Computer Science*, pages 251–257.