Semantic Web 0 (0) 1
IOS Press

Querying Knowledge Graphs with Extended Property Paths

Valeria Fionda [a], Giuseppe Pirrò [b,*] and Mariano P. Consens [c]
[a] *DeMaCS, University of Calabria, Rende (CS), Italy*
E-mail: fionda@mat.unical.it
[b] *ICAR-CNR, Italian National Research Council, Rende (CS), Italy*
E-mail: pirro@icar.cnr.it
[c] *MIE, University of Toronto, Toronto, Canada*
E-mail: consens@cs.toronto.edu

Abstract. The increasing number of Knowledge Graphs (KGs) available today calls for powerful query languages that can strike a balance between expressiveness and complexity of query evaluation and, that can be easily integrated into existing query processing infrastructures. We present Extended Property Paths (EPPs), a significant enhancement of Property Paths (PPs), the navigational core included in the SPARQL query language. We introduce the EPPs syntax, which allows to capture in a succinct way a larger class of navigational queries than PPs and other navigational extensions of SPARQL, and provide formal semantics. We describe a translation from non-recursive EPPs (nEPPs) into SPARQL queries and provide novel expressiveness results about the capability of SPARQL sub-languages to express navigational queries. We prove that the language of EPPs is more expressive than that of PPs; using EPPs within SPARQL allows to express things that cannot be expressed when only using PPs. We also study the expressiveness of SPARQL with EPPs in terms of reasoning capabilities. We show that SPARQL with EPPs is expressive enough to capture the main RDFS reasoning functionalities and describe how a query can be rewritten into another query enhanced with reasoning capabilities. We complement our contributions with an implementation of EPPs as the SPARQL-independent iEPPs language and an implementation of the translation of nEPPs into SPARQL queries. What sets our approach apart from previous research on querying KGs is the possibility to evaluate both nEPPs and SPARQL with nEPPs queries under the RDFS entailment regime on existing query processors. We report on an experimental evaluation on a variety of real KGs.

Keywords: SPARQL, Property Paths, Navigational Languages, Query-based reasoning, Expressive Power, Translation

1. Introduction

Knowledge Graphs (KGs) are becoming crucial in many application scenarios [1]. The Google Knowledge Graph [2], Facebook Open Graph [3], DBpedia [4], Yago [5], and Wikidata [6] are just a few examples. Devising powerful KG query languages that can strike a balance between expressiveness and complexity of query evaluation while at the same time having little impact on existing query processing infrastructures is crucial [7]. There is a large number of KGs encoded in RDF [8], the W3C standard for the publishing of structured data on the Web [9]. To query RDF data, a standard query language, called SPARQL [10, 11], has been designed. While an early version of SPARQL did not provide explicit navigational capabilities that are crucial for querying graph-like data, the most recent version (SPARQL 1.1) incorporates *Property Paths* (PPs). The main goal of PPs is to allow the writing of navigational queries in a more succinct way and support basic transitive closure computations. However, it has been widely recognized that PPs offer very limited expressiveness [12–15]; notably, PPs lack any form of tests within a path, a feature that can be very useful when dealing with graph data. For example, a query like find *Italian exclusive friends*, that is, "friends that are not friend of any other friends, and are Italian" requires both path difference and tests.

*Corresponding author. E-mail: pirro@icar.cnr.it.

Surprisingly, these features neither are available in PPs nor in any previous navigational extension of SPARQL (e.g., NRE [16]). In this paper we introduce *Extended Property Paths* (EPPs), a comprehensive language including a set of navigational features to extend the current navigational core of SPARQL. In particular, EPPs integrate features like path conjunction, difference, and repetitions, as well as powerful types of tests. A preliminary description of the language appeared in the proceedings of the AAAI'15 conference [17].

1.1. EPPs by Example

We introduce the main features of EPPs by describing a few examples. An excerpt of a KG is given in Fig. 1. Intuitively, an EPP expression defines a binary relation on the nodes of the graph upon which it is evaluated.

Example 1. (**Path Difference**). *Find pairs of cities located in the same country but not in the same region.*

Navigational Languages such as Nested Expression (NRE) and PPs cannot express such request due to the lack of path difference (the result has to exclude cities in the same region). With EPPs, the request can be expressed as follows (the full syntax will be presented in Section 3.1):

```
?x ((:country/^:country)~(:region/^:region)) ?y
```

The symbol ^ *denotes backward navigation from the object to the subject of a triple. Path difference* ~ *enables to discard from the set of cities in the same country (i.e.,* `:country/^:country`*) those that are in the same region (i.e.,* `:region/^:region`*). A SPARQL-independent evaluation pattern of the EPP expression*[1] *considers all the bindings of the variable* `?x` *(representing one of the cities that are wanted) and then evaluates the expression from each binding. The result is the set of bindings for the variable* `?y`*, representing the other city. From* `:Rome`*, the evaluation of the expression* `((:country/^:country)~(:region/^:region))` *reaches* `:Florence` *and* `:Carrara`. ◀

Example 2. (**Path Conjunction**). *Find pairs of cities located in the same country and in the same region.*

```
?x ((:country/^:country)&(:region/^:region)) ?y
```

In this case, path conjunction & *enables to keep from the set of nodes satisfying the first subexpression those that also satisfy the second one. From* `:Florence`*, the evaluation of the expression* `((:country/^:country)&(:region/^:region))` *reaches the cities* `:Florence` *and* `:Carrara`. ◀

Example 3. (**Tests**). *Find pairs of cities governed by the same political party founded before 2010.*

```
?x (:leaderParty&&TP(_o,:formationYear&&
T(_o < 2010))/^:leaderParty) ?y
```

`TP` *denotes a test for the existence of a path whose parameters specify the position in the triple from which the test starts (*`_o` *denotes the object of the last traversed triple), and a path (in this case* `:formationYear&& T(_o` < 2010*). The path is composed by logical* AND *(&&) of two tests. The first checks the existence of an edge* `:formationYear` *and the second, which starts from the object of the last traversed triple (i.e.,* `:formationYear`*), checks that the value is less than* 2010*. PPs and NREs-based languages lack tests like* TP *to check the existence of paths satisfying conditions. Starting from* `:Rome`*, the first logical* AND *(via &&) of two tests is performed; one checks for the existence of an edge* `:leaderParty`*, which leads to* `:Democratic_Party`*, while the other (i.e.,* TP*) starts from the object of the previous navigational step, that is, the object of* `(:Rome, :leaderParty, :Democratic_Party)`*. From* `:Democratic_Party`*, another logical* AND *(via &&) of two tests is evaluated. The first checks the existence of an edge* `:formationYear` *and enables to reach the node* 2007*; the second, which starts from the object of the previous step (i.e.,* 2007*), checks that the value is* < 2010*; in this case the test succeeds and the evaluation continues from* `:Democratic_Party` *by navigating the edge* `:leaderParty` *backward and reaching the nodes* `:Florence` *and* `:Rome` *included in the results.* ◀

Composing all the previous features together, we can express a more complex query.

Example 4. (**Path Conjunction, Difference and Tests**).*Find pairs of cities located in the same country but not in the same region. Such cities must be governed by the same political party, which has been founded before 2010.*

```
?x ((:country/^:country)~(:region/^:region)) &
(:leaderParty&&TP(_o,:formationYear&&T(_o < 2010))
/^:leaderParty ?y
```

[1] We provide a detailed algorithm in Section 7.

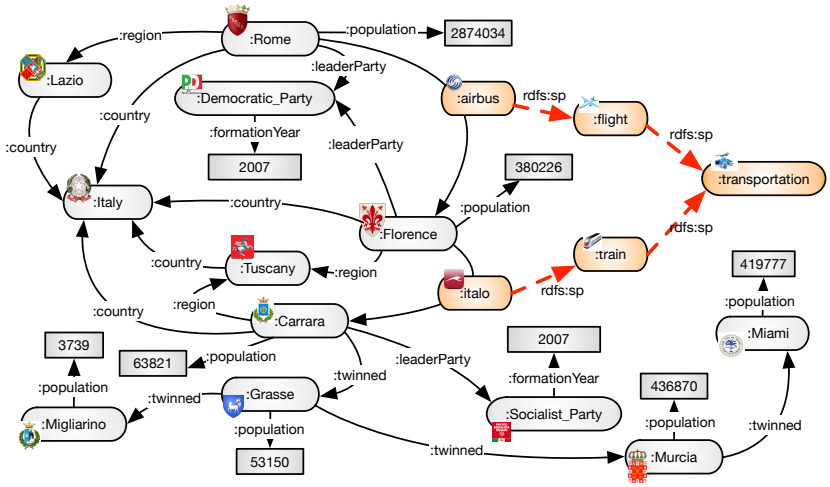


Fig. 1. An excerpt of Knowledge Graph taken from DBpedia.

From `:Rome`*, the evaluation of the first subexpression, including* `:country` *and* `:region`*, allows to reach the nodes* `:Florence` *and* `:Carrara`*. The evaluation of the second part of the path conjunction allows to reach the nodes* `:Rome` *and* `:Florence`*. From* `:Rome` *we reach* `:Florence`*.* ◀

Example 4 cannot be expressed by using NREs-based languages or PPs. These languages lack both path difference (we want cities in the same country but *not* in the same region) and conjunction (additionally, they must be governed by the same political party). We have discussed in the previous examples how a SPARQL-independent algorithm can evaluate EPP expressions. However, since our primary goal is to allow powerful navigation queries on existing KGs query processing infrastructures, we devised a translation of non-recursive EPPs into SPARQL. Our approach follows the same reasoning as the translation of non-recursive PPs into SPARQL used by the current SPARQL standard [18]. The advantage of using EPPs to write non-recursive navigational queries instead of writing them directly into SPARQL is that the same request can be expressed more succinctly and without the need to deal with intermediate variables.

Example 5. *The SPARQL query corresponding to the translation of the EPP expression in Example 4 is shown in Fig. 2, where* `?v1`*,* `?v2`*,* `?v3` *and* `?v4` *are variables automatically generated by the translation algorithm.*

```
SELECT ?x ?y WHERE {
{?x :country ?v1.?y :country ?v1.}
MINUS{?x :region ?v2.?y :region ?v2. }
?x :leaderParty ?v3. ?y :leaderParty ?v3.
FILTER EXISTS{?v3 :formationYear ?v4.
FILTER(?v4 < 2010)} }
```

Fig. 2. SPARQL translation of the EPP expression in Example 4. ◀

Example 6. (**Arbitrary Path Length with Tests**). *Find cities reachable from* `:Carrara` *connected via a path of arbitrary length with labels* `:twinned` *considering only those cities reachable by a chain of intermediate cities having* `:population` *greater than* 10000*. The EPP expression capturing this request is:*

```
:Carrara (:twinned&&
TP(_o, :population&&T(_o>10000)))* ?y
```

The expression involves arbitrary length paths plus tests. The evaluation checks from the node `:Carrara` *the existence of paths of arbitrary length (denoted by* **) where each node reached in the path must satisfy the test* `TP`*. Starting from* `:Carrara`*, with a path* `:twinned` *of length one,* `:Grasse` *is reached. From this node the test* `TP` *is evaluated to check the existence of a triple* (`:Grasse`, `:population`, `n`) *with* `n>10000`*. Since* `:Grasse` *passes the* `TP` *test, starting from it the path* `:twinned` *is evaluated again reaching* `:Murcia`*, which also passes the* `TP` *test and* `:Migliarino` *that does not pass the* `TP` *test. The evaluation continues from* `:Murcia` *and stops when reaching the node* `:Miami`*, which passes the* `TP` *test. Overall, we reach* `:Carrara`*,* `:Grasse`*,* `:Murcia`*, and* `:Miami`*.* ◀

The EPP expression in Example 6 cannot be translated into a basic SPARQL query because it makes use of the closure operator * (requiring the evaluation of (:twinned &&TP(_o, :population&&T(_o>10000))) an a-priori unknown number of times). To give semantics to this kind of EPP expressions we introduce the evaluation function EALP1 (Fig. 8), which extends the function ALP1 defined for PPs in the SPARQL standard [10]. EPPs also support *path repetitions* (handled via EALP1), that are a concise way of expressing the union of concatenations of an expression between a *min* and *max* number of times.

Example 7. (**Path Repetitions**). *By using path repetitions between 1 and 2, the expression in Example 6 can be rewritten as follows:*

:Carrara (:twinned &&
TP(_o, :population&&T(_o>10000)))$\{\{1,2\}\}$?y

◀

So far we have presented examples of isolated EPPs expressions. We now consider their usage in SPARQL.

Example 8. (**EPPs within SPARQL**). *Find pairs of cities (A,B) and their populations such that: (i) A and B are in the same country, but not in the same region; (ii) there exists some transportation from A to B.*

```
SELECT ?cityA ?cityB ?popA ?popB WHERE {
?cityA :population ?popA.
?cityB :population ?popB.
{ /* BEGIN EPPs pattern */
?cityA ((:country/^country)
~(:region/^:region))
&:transportation ?cityB.
} /* END EPPs pattern */
}
```

Fig. 3. EPPs used inside SPARQL as for Example 8.

The query in Example 3 allows to obtain the population of the pairs of cities satisfying the EPP expression by introducing two additional patterns, where the variables ?popA *and* ?popB *are bound to population information. When the query is evaluated on the graph reported in Fig. 1 it produces no results; for instance the pair* (:Rome, :Florence) *is connected by an* :airbus *that is a kind of* :plane*, which is a mean of* :transportation*, but there is no edge whose label is* :transportation. ◀

The previous example does not take the KG RDFS schema into account. When considering transportation services without specifying the exact type of service, one would be able to actually discover the connection between :Rome and :Florence. This can be achieved by performing sub-property inference according to the RDFS entailment regime. One crucial aspect of EPPs is that they can capture the main RDFS inference types by encoding each inference rule in a prototypical EPP expression (see Section 5.2), with the advantage that the resulting expressions *can be translated into SPARQL and evaluated on existing processors* (via ALP1).

Example 9. (**EPPs and Reasoning**). *The EPPs in Example 8 can be automatically rewritten into an EPP supporting RDFS reasoning as follows:*

```
SELECT ?cityA ?cityB ?popA ?popB WHERE {
?cityA :population ?popA.
?cityB :population ?popB.
{ /* BEGIN EPPs pattern */
?cityA
 ((:country/^country)~(:region/^:region))&
(TP(_p,(rdfs:sp*/rdfs:sp)
&&T(_o=:transportation))
||T(_p=:transportation)))))
 ?cityB.
}} /* END EPPs pattern */
```

The translation to SPARQL this query is reported in Fig. 4. When this query is evaluated on the graph in Fig. 1 it produces (?cityA→:Rome, ?cityB→:Florence, ?popA→2874034, ?popB→380226). ◀

1.2. Contributions and Organization

The contribution of the paper are both theoretical and practical.

- We introduce two languages EPPs and iEPPs to query KGs. They have the same syntax but different semantics; one based on multiset (Section 3.2) and complying with SPARQL, and the other based on sets (Section 7.1).
- We provide a translation from non-recursive EPPs into SPARQL queries (Section 4). The benefit of our translation is twofold; on one hand, it allows to evaluate nEPPs (a larger class of queries than non-recursive PPs) into existing SPARQL processors; on the other hand, the usage of our translation paves the way toward readily incorporating EPPs in the current SPARQL standard.
- Building upon our translation, we also show how a SPARQL query can be rewritten into another SPARQL query that incorporates reasoning capabilities and can be evaluated on existing SPARQL processors (Section 5).

```
SELECT ?cityA ?cityB ?popA ?popB WHERE
{ ?cityA: population ?popA. ?cityB : population ?popB.
/* BEGIN EPPs-\rhoDF to SPARQL TRANSLATION */
{ {?cityA ? pN_0_0_0 ? middleN_0_0.
FILTER(?pN_0_0_0 =:country)}
UNION
{?cityA ?pN_0_0_0 ?middleN_0_0.
FILTER EXISTS {?pN_0_0_0 sp* ?middleN_0_0_0_1_0.
?middleN_0_0_0_1_0 sp ?endN_0_0_0_1_0.
FILTER( ?endN_0_0_0_1_0 = :country)}
{?cityB ?pN_0_0_1 ?middleN_0_0.
FILTER(( ?pN_0_0_1 = :country)}
UNION { ?cityB ?pN_0_0_1 ?middleN_0_0.
FILTER EXISTS {?pN_0_0_1 sp * ?middleN_0_0_1_0_1_0.
?middleN_0_0_1_0_1_0 sp ?endN_0_0_1_0_1_0.
FILTER( ?endN_0_0_1_0_1_0 = :country)}
}
MINUS
{?cityA ? pN_0_1_0 ? middleN_0_1.
FILTER( ?pN_0_1_0 = :region)}}
UNION {?cityA ?pN_0_1_0 ?middleN_0_1.
FILTER EXISTS {?pN_0_1_0 sp * ?middleN_0_1_0_0_1_0.
?middleN_0_1_0_0_1_0 sp ? endN_0_1_0_0_1_0.
FILTER( ? endN_0_1_0_0_1_0 = : region)}
{ ?cityB ?pN_0_1_1 ?middleN_0_1.
FILTER( ?pN_0_1_1 = :region)}
UNION {?cityB ?pN_0_1_1 ?middleN_0_1.
FILTER EXISTS {?pN_0_1_1 sp * ?middleN_0_1_1_0_1_0.
?middleN_0_1_1_0_1_0 sp ? endN_0_1_1_0_1_0.
FILTER( ? endN_0_1_1_0_1_0 = region)}
}
{ ?cityA ?pN_1 ?cityB. FILTER( ? pN_1 = : transportation) }
UNION {?cityA ? pN_1 ? cityB.
FILTER EXISTS { ?pN_1 sp * ?middleN_1_0_1_0.
?middleN_1_0_1_0 sp ?endN_1_0_1_0.
FILTER( ? endN_1_0_1_0 = : transportation)} }}
/* END rhoDF-EPPs to SPARQL TRANSLATION */
}
```

Fig. 4. SPARQL translation of Example 9.

- We implement the nEPPs to SPARQL translation as an extension of the Jena library and an iEPPs query processor. Both are available on-line[2].
- We perform an extensive experimental evaluation on a variety of real data sets (Section 8).

From a theoretical point of view:

- We introduce iEPPs as a SPARQL-independent language and discuss its complexity (Section 7.2).
- We report novel expressiveness results about the capability of SPARQL in expressing navigational queries. We show that SPARQL is expressive enough to capture nEPPs (Section 4.2).
- We prove that the language of EPPs is more expressive than that of PPs and, as a by-product, that the fragment of SPARQL including EPPs, AND and UNION is more expressive than the fragment of SPARQL including PPs, AND and UNION (Section 6.1).
- We provide a novel study about the expressiveness of SPARQL in terms of the main reasoning capabilities of RDFS (defined as ρdf [19]) when considering different navigational cores (Section 6.2). We show that SPARQL is expressive enough to capture ρdf.

[2]https://extendedpps.wordpress.com

The remainder of the paper is organized as follows. We provide some background definitions in Section 2. Section 3 presents the EPPs syntax and semantics. Section 4 formalizes the translation of non-recursive EPPs into SPARQL queries. Section 5 shows how EPPs support reasoning. The expressiveness of EPPs is analyzed in Section 6. The iEPPs language is described in Section 7. The implementation and the evaluation of EPPs and iEPPs are discussed in Section 8. Section 9 discusses related literature. We conclude in Section 10.

2. Preliminaries

In this section we provide some background about RDF, SPARQL and SPARQL property paths. An RDF triple[3] is a tuple of the form $\langle s, p, o\rangle \in \mathbf{I} \times \mathbf{I} \times \mathbf{I} \cup \mathbf{L}$, where $\mathbf{I}$ and $\mathbf{L}$ are countably infinite sets of IRIs and literals respectively. An RDF graph G is a set of triples. The set of terms of an RDF graph (i.e., the set of IRIs and literals appearing in the graph) is denoted by terms(G) while nodes(G) denotes the set of terms used as a subject or object of a triple. In what follows we will focus on the fragment of SPARQL including the SELECT query form and provide a formalization of its semantics along the lines of Angles and Gutierrez [20] that is faithful to the semantics of the W3C standard.

2.1. Background on SPARQL

Let $\mathcal{V}$ be a countably infinite set of variables, such that $\mathcal{V} \cap (\mathbf{I} \cup \mathbf{L}) = \varnothing$. A (solution) mapping μ is a partial function μ: $\mathcal{V} \rightarrow \mathbf{I} \cup \mathbf{L}$. The *empty mapping*, denoted μ_0, is the mapping satisfying $\mathrm{dom}(\mu_0) = \varnothing$. Two mappings, say μ_1 and μ_2, are *compatible* (resp., not compatible), denoted by $\mu_1 \sim \mu_2$ (resp., $\mu_1 \not\sim \mu_2$), if $\mu_1(?X) = \mu_2(?X)$ for all variables $?X \in \big(\mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)\big)$ (resp., if $\mu_1(?X) \neq \mu_2(?X)$ for some $?X \in (\mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2))$). If $\mu_1 \sim \mu_2$ then we write $\mu_1 \cup \mu_2$ for the mapping obtained by extending μ_1 according to μ_2 on all variables in $\mathrm{dom}(\mu_2) \setminus \mathrm{dom}(\mu_1)$. Note that two mappings with disjoint domains are always compatible, and that the empty mapping μ_0 is compatible with any other mapping. Given a finite set of variables $W \subset \mathcal{V}$, the restriction of a mapping μ to W, denoted $\mu_{|W}$, is a mapping μ' satisfying

[3]To simplify the discussion we do not consider blank nodes in this section; we will adddress this issue later in Section 2.4.

$\mathrm{dom}(\mu') = \mathrm{dom}(\mu) \cap W$ and $\mu'(?X) = \mu(?X)$ for every $?X \in \mathrm{dom}(\mu) \cap W$.

A *selection formula* is defined recursively as follows: (i) If $?X, ?Y \in \mathcal{V}$ and $c \in \mathbf{I} \cup \mathbf{L}$ then $(?X = c)$, $(?X = ?Y)$ and $\mathrm{bound}(?X)$ are atomic selection formulas; (ii) If F and F' are selection formulas then $(F \wedge F')$, $(F \vee F')$ and $\neg(F)$ are boolean selection formulas. The evaluation of a selection formula F under μ, denoted $\mu(F)$, is defined in a three-valued logic (i.e. with values `true`, `false`, and *error*) as follows:

- If F is $?X = c$ and $?X \in \mathrm{dom}(\mu)$, then $\mu(F) =$ `true` when $\mu(?X) = c$ and $\mu(F) =$ `false` otherwise. If $?X \notin \mathrm{dom}(\mu)$ then $\mu(F) = error$.
- If F is $?X = ?Y$ and $?X, ?Y \in \mathrm{dom}(\mu)$, then $\mu(F) =$ `true` when $\mu(?X) = \mu(?Y)$ and $\mu(F) =$ `false` otherwise. If either $?X \notin \mathrm{dom}(\mu)$ or $?Y \notin \mathrm{dom}(\mu)$ then $\mu(F) = error$.
- If F is $\mathrm{bound}(?X)$ and $?X \in \mathrm{dom}(\mu)$ then $\mu(F) =$ `true` else $\mu(F) =$ `false`.
- If F is a complex selection formula then it is evaluated following the three-valued logic presented in Table 1.

Table 1
Three-valued logic for evaluating selection formulas.

p	q	$p \wedge q$	$p \vee q$
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`true`	*error*	*error*	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`
`false`	*error*	`false`	*error*
error	`true`	*error*	`true`
error	`false`	`false`	*error*
error	*error*	*error*	*error*

p	$\neg p$
`true`	`false`
`false`	`true`
error	*error*

We use the symbol Ω to denote a multiset and $\mathrm{card}(\mu, \Omega)$ to denote the cardinality of the mapping μ in the multiset Ω. Moreover, it applies that $\mathrm{card}(\mu, \Omega) = 0$ when $\mu \notin \Omega$. We use Ω_0 to denote the multiset containing the only mapping μ_0, that is $\mathrm{card}(\mu_0, \Omega_0) > 0$ (Ω_0 is called the join identity). The domain of a solution mapping Ω is defined as $\mathrm{dom}(\Omega) = \bigcup_{\mu \in \Omega} \mathrm{dom}(\mu)$. The *SPARQL algebra for multisets of mappings* is composed of the operations of projection, selection, join, difference, left-join, union and minus. Let Ω_1, Ω_2 be multisets of mappings, W be a set of variables and F be a selection formula.

Definition 10. **(Operations over multisets of mappings).** Let Ω_1 and Ω_2 be multiset of mappings, then:

Projection: $\pi_W(\Omega_1) = \{\mu' \mid \mu \in \Omega_1, \mu' = \mu_{|W}\}$, $\mathrm{card}(\mu', \pi_W(\Omega_1)) = \sum_{\mu \in \Omega_1 \text{ s.t. } \mu' = \mu_{|W}} \mathrm{card}(\mu, \Omega_1)$

Selection: $\sigma_F(\Omega_1) = \{\mu \in \Omega_1 \mid \mu(F) =$ `true`$\}$ where $\mathrm{card}(\mu, \sigma_F(\Omega_1)) = \mathrm{card}(\mu, \Omega_1)$

Union: $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$ where $\mathrm{card}(\mu, \Omega_1 \cup \Omega_2) = \mathrm{card}(\mu, \Omega_1) + \mathrm{card}(\mu, \Omega_2)$

Join: $\Omega_1 \bowtie \Omega_2 = \{\mu = (\mu_1 \cup \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\}$, $\mathrm{card}(\mu, \Omega_1 \bowtie \Omega_2) =$ $= \sum_{\mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \text{ s.t. } \mu = (\mu_1 \cup \mu_2)} \mathrm{card}(\mu_1, \Omega_1) \times \mathrm{card}(\mu_2, \Omega_2)$.

Difference: $\Omega_1 \setminus_F \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, (\mu_1 \nsim \mu_2) \vee (\mu_1 \sim \mu_2 \wedge (\mu_1 \cup \mu_2)(F) =$ `false`$)\}$ where $\mathrm{card}(\mu_1, \Omega_1 \setminus_F \Omega_2) = \mathrm{card}(\mu_1, \Omega_1)$

Minus: $\Omega_1 - \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \nsim \mu_2 \vee \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2) = \varnothing\}$ where $\mathrm{card}(\mu_1, \Omega_1 - \Omega_2) = \mathrm{card}(\mu_1, \Omega_1)$.

Left Join: $\Omega_1 ⟕_F \Omega_2 = \sigma_F(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus_F \Omega_2)$ where $\mathrm{card}(\mu, \Omega_1 ⟕_F \Omega_2) = \mathrm{card}(\mu, \sigma_F(\Omega_1 \bowtie \Omega_2)) + \mathrm{card}(\mu, \Omega_1 \setminus_F \Omega_2)$.

2.2. SPARQL Patterns

We now introduce SPARQL graph patterns. A *graph pattern* is defined recursively as follows:

- A tuple from $(\mathbf{I} \cup \mathbf{L} \cup \mathcal{V}) \times (\mathbf{I} \cup \mathcal{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathcal{V})$ is a graph pattern called a *triple pattern*[4].
- If $\mathcal{P}_1$ and $\mathcal{P}_2$ are patterns then $(\mathcal{P}_1$ AND $\mathcal{P}_2)$, $(\mathcal{P}_1$ UNION $\mathcal{P}_2)$, $(\mathcal{P}_1$ OPTIONAL $\mathcal{P}_2)$, $(\mathcal{P}_1$ MINUS $\mathcal{P}_2)$ and $(\mathcal{P}_1$ NOT-EXISTS $\mathcal{P}_2)$ are graph patterns.
- If $\mathcal{P}_1$ is a pattern and C is a filter constraint (as defined below) then $(\mathcal{P}_1 \mathrm{FILTER} C)$ is a pattern.

A *filter constraint* is defined recursively as follows: (i) If $?X, ?Y \in \mathcal{V}$ and $c \in \mathbf{I} \cup \mathbf{L}$ then $(?X = c)$, $(?X = ?Y)$ and $\mathrm{bound}(?X)$ are *atomic filter constraints*; (ii) If C_1 and C_2 are filter constraints then $(!C_1)$, $(C_1 \,||\, C_2)$ and $(C_1 \,\&\&\, C_2)$ are *complex filter constraints*. Given a filter constraint C, we denote by $f(C)$ the selection formula obtained from C. Note that there exists a simple and direct translation from filter constraints to selection formulas and vice-versa.

Given a triple pattern t and a mapping μ such that $var(t) \subseteq \mathrm{dom}(\mu)$, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. Overloading the above definition, we denote by $\mu(\mathcal{P})$ the graph pattern obtained by the recursive substitution of variables in every triple pattern and filter constraint occurring in the graph pattern P according to μ.

[4] We assume that any triple pattern contains at least one variable.

R1	$[\![\langle \alpha, u, \beta \rangle]\!]_G := \Omega = \{\, \mu \mid \mathrm{dom}(\mu) = (\{\alpha,\beta\} \cap \mathcal{V}) \text{ and } \mu(\langle \alpha, u, \beta \rangle) \in G \,\}, \mathrm{card}(\mu, \Omega) = 1$
R2	$[\![\langle \alpha, !(u_1 \mid \ldots \mid u_n), \beta \rangle]\!]_G := \Omega = \{\, \mu \mid \mathrm{dom}(\mu) = (\{\alpha,\beta\} \cap \mathcal{V}),\ \exists u \in \mathbf{I} \text{ such that } u \notin \{u_1,\ldots,u_n\} \text{ and } \mu(\langle \alpha, u, \beta \rangle) \in G \}$ and $\mathrm{card}(\mu, \Omega) = \big\lvert \{ u \mid u \in \mathbf{I}, u \notin \{u_1,\ldots,u_n\}, \mu(\langle \alpha, u, \beta \rangle) \in G \} \big\rvert$
R3	$[\![\langle \alpha, \hat{}\mathtt{elt}, \beta \rangle]\!]_G := \Omega = [\![\langle \beta, \mathtt{elt}, \alpha \rangle]\!]_G$
R4	$[\![\langle \alpha, \mathtt{elt}_1/\mathtt{elt}_2, \beta \rangle]\!]_G := \Omega = \pi_{\{\alpha,\beta\}\cap\mathcal{V}} \big([\![\langle \alpha, \mathtt{elt}_1, ?v \rangle]\!]_G \bowtie [\![\langle ?v, \mathtt{elt}_2, \beta \rangle]\!]_G \big)$
R5	$[\![\langle \alpha, (\mathtt{elt}_1 \mid \mathtt{elt}_2), \beta \rangle]\!]_G := \Omega = [\![\langle \alpha, \mathtt{elt}_1, \beta \rangle]\!]_G \cup [\![\langle \alpha, \mathtt{elt}_2, \beta \rangle]\!]_G$
R6	$[\![\langle x_\mathrm{L}, (\mathtt{elt})^*, ?v_\mathrm{R} \rangle]\!]_G := \Omega = \{\, \mu \mid \mathrm{dom}(\mu) = \{?v_\mathrm{R}\} \text{ and } \mu(?v_\mathrm{R}) \in \mathtt{ALP1}(x_\mathrm{L}, \mathtt{elt}, G) \}, \mathrm{card}(\mu, \Omega) = 1$
R7	$[\![\langle ?v_\mathrm{L}, (\mathtt{elt})^*, ?v_\mathrm{R} \rangle]\!]_G := \Omega = \{\, \mu \mid \mathrm{dom}(\mu) = \{?v_\mathrm{L}, ?v_\mathrm{R}\} \text{ and } \mu(?v_\mathrm{L}) \in \mathrm{terms}(G) \text{ and } \mu(?v_\mathrm{R}) \in \mathtt{ALP1}(\mu(?v_\mathrm{L}), \mathtt{elt}, G) \},$ $\mathrm{card}(\mu, \Omega) = 1$
R8	$[\![\langle ?v_\mathrm{L}, (\mathtt{elt})^*, x_\mathrm{R} \rangle]\!]_G := \Omega = [\![\langle x_\mathrm{R}, (\hat{}\mathtt{elt})^*, ?v_\mathrm{L} \rangle]\!]_G$
R9	$[\![\langle x_\mathrm{L}, (\mathtt{elt})^*, x_\mathrm{R} \rangle]\!]_G := \Omega = \begin{cases} \{\mu_0\}, \text{if } \exists\, \mu \in [\![\langle x_\mathrm{L}, (\mathtt{elt})^*, ?v \rangle]\!]_G : \mu(?v) = x_\mathrm{R}, \text{and } \mathrm{card}(\mu_0, \Omega) = 1 \\ \varnothing, \text{otherwise} \end{cases}$

Fig. 5. Standard query semantics of SPARQL Property Paths, where $\alpha, \beta \in (\mathbf{I} \cup \mathbf{L} \cup \mathcal{V})$; $u, u_1, ..., u_n \in \mathbf{I}$; $x_\mathrm{L}, x_\mathrm{R} \in (\mathbf{I} \cup \mathbf{L})$; $?v_\mathrm{L}, ?v_\mathrm{R} \in \mathcal{V}$; $?v \in \mathcal{V}$ is a fresh variable.

Function $\mathtt{ALP1}(\gamma, \mathtt{elt}, G)$
Input: $\gamma \in (\mathbf{I} \cup \mathbf{L})$,
$\mathtt{elt}$ is a PP expression,
G is an RDF graph.
1: *Visited* := $\varnothing$
2: $\mathtt{ALP2}(\gamma, \mathtt{elt}, \mathit{Visited}, G)$
3: **return** *Visited*

Function $\mathtt{ALP2}(\gamma, \mathtt{elt}, \mathit{Visited}, G)$
Input: $\gamma \in (\mathbf{I} \cup \mathbf{L})$, $\mathtt{elt}$ is a PP expression,
$\mathit{Visited} \subseteq (\mathbf{I} \cup \mathbf{L})$, G is an RDF graph.
4: **if** $\gamma \notin \mathit{Visited}$ **then**
5: add γ to *Visited*
6: **for all** $\mu \in [\![\langle ?x, \mathtt{elt}, ?y \rangle]\!]_G$ such that $\mu(?x) = \gamma$ and $?x, ?y \in \mathcal{V}$ **do**
7: $\mathtt{ALP2}(\mu(?y), \mathtt{elt}, \mathit{Visited}, G)$

Fig. 6. Auxiliary functions used for defining the semantics of PP expressions of the form $\mathtt{elt}^*$.

2.3. Semantics of SPARQL graph patterns

The evaluation of a SPARQL graph pattern $\mathcal{P}$ over an RDF graph G is defined as a function $[\![\mathcal{P}]\!]_G$ which returns a multiset of solution mappings. Let $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ be graph patterns and C be a filter constraint. The evaluation of a graph pattern $\mathcal{P}$ over a graph G is defined recursively as follows:

- If $\mathcal{P}$ is a triple pattern t_p then $[\![t_p]\!]_G = \{\mu \mid dom(\mu) = var(t_p) \text{ and } \mu(t_p) \in G\}$ where $var(t_p)$ is the set of variables in t_p and the cardinality of each mapping is 1.
- If $\mathcal{P} = (\mathcal{P}_1 \text{ AND } \mathcal{P}_2)$, then $[\![\mathcal{P}]\!]_G = [\![\mathcal{P}_1]\!]_G \bowtie [\![\mathcal{P}_2]\!]_G$
- If $\mathcal{P} = (\mathcal{P}_1 \text{ UNION } \mathcal{P}_2)$, then $[\![\mathcal{P}]\!]_G = [\![\mathcal{P}_1]\!]_G \cup [\![\mathcal{P}_2]\!]_G$
- If $\mathcal{P} = (\mathcal{P}_1 \text{ OPTIONAL } \mathcal{P}_2)$, then: then
  (a) if $\mathcal{P}_2$ is $(\mathcal{P}_3 \text{ FILTER } C)$ then $[\![\mathcal{P}]\!]_G = [\![\mathcal{P}_1]\!]_G \mathbin{⟕}_{f(C)} [\![\mathcal{P}_3]\!]_G$
  (b) else $[\![\mathcal{P}]\!]_G = [\![\mathcal{P}_1]\!]_G \mathbin{⟕}_{(\mathtt{true})} [\![\mathcal{P}_2]\!]_G$
- If $\mathcal{P} = (\mathcal{P}_1 \text{ MINUS } \mathcal{P}_2)$, then $[\![\mathcal{P}]\!]_G = [\![\mathcal{P}_1]\!]_G - [\![\mathcal{P}_2]\!]_G$
- If $\mathcal{P} = (\mathcal{P}_1 \text{ NOT-EXISTS } \mathcal{P}_2)$, then $[\![(\mathcal{P}_1 \text{ NOT-EXISTS } \mathcal{P}_2)]\!]_G = \{\mu \mid \mu \in [\![\mathcal{P}_1]\!]_G \wedge [\![\mu(\mathcal{P}_2)]\!]_G = \varnothing\}$
- If $\mathcal{P} = (\mathcal{P}_1 \text{ FILTER } C)$, then $[\![\mathcal{P}_1 \text{ FILTER } C]\!]_G = \sigma_{f(C)}([\![P_1]\!]_G)$

2.4. SPARQL Property Paths

Property paths (PPs) have been incorporated into the SPARQL standard with two main motivations; first, to provide explicit graph navigational capabilities (thus allowing the writing of SPARQL navigational queries in a more succinct way); second, to introduce the transitive closure operator * previously not available in SPARQL. The design of PPs was influenced by earlier proposals (e.g., PSPARQL [21], nSPARQL [11]).

Definition 11. (**Property Path Pattern**). A *property path pattern* (or *PP pattern* for short) is a tuple $\mathcal{P} = \langle \alpha, \mathtt{elt}, \beta \rangle$ with $\alpha \in (\mathbf{I} \cup \mathbf{L} \cup \mathcal{V})$, $\beta \in (\mathbf{I} \cup \mathbf{L} \cup \mathcal{V})$, and $\mathtt{elt}$ is a *property path expression* (*PP expression*) that is defined by the following grammar (where $u, u_1, \ldots, u_n \in \mathbf{I}$):

$$\mathtt{elt} := u \mid\ !(u_1 \mid \ldots \mid u_n) \mid\ !(\hat{}u_1 \mid \ldots \mid \hat{}u_n) \mid\ !(u_1 \mid \ldots \mid u_j \mid \ldots \mid \hat{}u_q \mid \ldots \mid \hat{}u_n) \mid\ \mathtt{elt}/\mathtt{elt} \mid\ (\mathtt{elt} \mid \mathtt{elt}) \mid\ (\mathtt{elt})^* \mid\ \hat{}\mathtt{elt}$$

The SPARQL standard introduces additional types of PP expressions [18]; since these are merely syntac-

Table 2
Syntax of EPPs. [1]If omitted is _s; [2]If omitted is_o.

epp ::=	'^' epp | epp '+' | epp '?' | epp'*' | epp '/' epp | epp '|' epp | '(' epp ')' | [POS][1] test [POS][2] | epp '&' epp | epp '∼' epp | epp'{'*l*,*h*'}' | epp'{{'l,h'}}'
test ::=	'!' test | test '&&' test | test '||' test | '(' test ')' | base
base ::=	iri | 'TP('POS ',' epp ')' | 'T(' EExp ')'
POS ::=	'_s' | '_p' | '_o'

tic sugar (they are defined in terms of expressions covered by the grammar given above), we ignore them in this paper. As another slight deviation from the standard, we do not permit blank nodes in PP patterns. PP patterns with blank nodes can be simulated using fresh variables. The SPARQL standard distinguishes between two types of property path expressions: *connectivity patterns* (or recursive PPs) that include closure (*), and *syntactic short forms* or non-recursive PPs (nPPs) that do not include it. As for the evaluation of PPs, the W3C spec. *informally* mentions the fact that nPPs can be evaluated via a translation into equivalent SPARQL basic expressions (see [10], Section 9.3). Property path patterns can be combined with graph patterns inside SPARQL patterns (using PP expressions in the middle position of a pattern).

2.5. Property Path Semantics

The semantics of Property Paths (PPs) is shown in Fig. 5. The semantics uses the evaluation function $[\![\langle \alpha, \texttt{elt}, \beta \rangle]\!]_G$, which takes as input a PP pattern and a graph and returns a multiset of solution mappings. In Fig. 5 we do not not report all the combinations of types of patterns as they can be derived in a similar way. For connectivity patterns the SPARQL standard introduces auxiliary functions called `ALP1` and `ALP1` that stand for `A`rbitrary `L`ength `P`aths (see Fig. 6); in this case the evaluation does not admit duplicates (thus solving a problem in an early version of the semantics that was based on counting [12, 22]).

3. Extended Property Paths

We now introduce our navigational extension of SPARQL called Extended Property Paths (EPPs). We present the syntax in Section 3.1 and the SPARQL-based formal semantics in Section 3.2.

3.1. Extended Property Paths Syntax

EPPs extend PPs and NREs-like languages with path conjunction/difference, repetitions and more types of tests. The idea is to *use EPP expressions in the predicate position of a property pattern (Definition 11) in lieu of PP expressions*. The importance of the new features considered by EPPs is witnessed by the fact that some of them (e.g., conjunction) are present in standards like XPath 2.0 [23]. Nevertheless, to the best of our knowledge no previous navigational extension of SPARQL has considered these features. As our goal is to extend the current SPARQL standard we refer the reader to Section 7 for a treatment of EPPs as a language independent from SPARQL.

Definition 12. (**Extended Property Path Pattern**). An *extended property path pattern* (or *EPP pattern* for short) is a tuple $\mathcal{EP} = \langle \alpha, \texttt{epp}, \beta \rangle$ with $\alpha \in (\mathbf{I} \cup \mathbf{L} \cup \mathcal{V})$, $\beta \in (\mathbf{I} \cup \mathbf{L} \cup \mathcal{V})$, and epp is an *extended property path expression* (*EPP expression*) that is defined by the grammar reported in Table 2.

EPPs introduce the following features: path conjunction ($\texttt{epp}_1\&\texttt{epp}_2$), path difference ($\texttt{epp}_1 \sim \texttt{epp}_2$), path repetitions between l and h times (denoted by $\texttt{epp}\{l,h\}$ for set, and $\texttt{epp}\{\{l,h\}\}$ for bag semantics). EPPs allow different types of tests (test) *within* a path by specifying the *starting/ending* positions (POS) of a test; it is possible to test from each of the subject, predicate and object positions in triples, mapped in the EPPs syntax to the position symbols _s, _p and _o, respectively. Positions do not need to be always specified; by default a test starts from the subject (_s) and ends on the object (_o) of the triple being evaluated. A test (test) can be a simple check for the existence of a IRI in forward/reverse direction. EPPs allow to express negated property sets by using the production test with the difference that the set of negated IRIs use the symbol '||' as separator instead of '|' used by PPs. A test can also be a nested EPP, i.e., $\texttt{TP}(\texttt{POS}, \texttt{epp})$, which corresponds to the evaluation of the expression epp starting from a position POS (of the last triple evaluated) and returns true if, and only if, there exists at least one node that can be reached via epp. In a test of type T, EExp (not reported here for sake of space) extends

Table 3

EPPs SPARQL-based semantics. The function $\mathbf{E}_T$ handles tests. $\Pi(\text{POS}, t)$ projects the element in position POS of a triple $t \in G$. Moreover, $u \in \mathbf{I}$; $?v_L, ?v_R \in \mathcal{V}$ and $?v_n \in \mathcal{V}$ is a fresh variable. `Evaluate` is a function that checks if the triple t satisfies `EExp`.

R1	$[\![\langle ?v_L, \hat{}\,\text{epp}, ?v_R\rangle]\!]_G := [\![\langle ?v_R, \text{epp}, ?v_L\rangle]\!]_G$
R2	$[\![\langle ?v_L, \text{epp}_1/\text{epp}_2, ?v_R\rangle]\!]_G := \pi_{\{?v_L,?v_R\}}\left([\![\langle ?v_L, \text{epp}_1, ?v\rangle]\!]_G \bowtie [\![\langle ?v, \text{epp}_2, ?v_R\rangle]\!]_G\right)$
R3	$[\![\langle ?v_L, (\text{epp})^*, ?v_R\rangle]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\}, \mu(?v_L) \in \text{terms}(G) \text{ and } \mu(?v_R) \in \texttt{EALP1}(\mu(?v_L), \text{epp}, G, 0, *)\}, \text{card}(\mu, \Omega) = 1$
R4	$[\![\langle ?v_L, (\text{epp})^+, ?v_R\rangle]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\}, \mu(?v_L) \in \text{terms}(G) \text{ and } \mu(?v_R) \in \texttt{EALP1}(\mu(?v_L), \text{epp}, G, 1, *)\}, \text{card}(\mu, \Omega) = 1$
R5	$[\![\langle ?v_L, (\text{epp})?, ?v_R\rangle]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\}, \mu(?v_L) = \mu(?v_R) \text{ or } \mu \in [\![\langle ?v_L, (\text{epp}), ?v_R\rangle]\!]_G\}, \text{card}(\mu, \Omega) = 1$
R6	$[\![\langle ?v_L, (\text{epp}_1 \mid \text{epp}_2), \beta\rangle]\!]_G := [\![\langle ?v_L, \text{epp}_1, ?v_R\rangle]\!]_G \cup [\![\langle \alpha, \text{epp}_2, ?v_R\rangle]\!]_G$
R7	$[\![\langle ?v_L, \text{epp}_1 \,\&\, \text{epp}_2, ?v_R\rangle]\!]_G := [\![\langle ?v_L, \text{epp}_1, ?v_R\rangle]\!]_G \bowtie [\![\langle ?v_L, \text{epp}_2, ?v_R\rangle]\!]_G$
R8	$[\![\langle ?v_L, \text{epp}_1 \sim \text{epp}_2, ?v_R\rangle]\!]_G := [\![\langle ?v_L, \text{epp}_1, ?v_R\rangle]\!]_G - [\![\langle ?v_L, \text{epp}_2, ?v_R\rangle]\!]_G$
R9	$[\![\langle ?v_L, \text{epp}\{\{l,h\}\}, ?v_R\rangle]\!]_G := \bigcup_{i=l}^{h} [\![\langle ?v_L, \text{epp}^i, ?v_R\rangle]\!]_G$
R9'	$[\![\langle ?v_L, \text{epp}\{l,h\}, ?v_R\rangle]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\}, \mu(?v_L) \in \text{terms}(G) \text{ and } \mu(?v_R) \in \texttt{EALP1}(\mu(?v_L), \text{epp}, G, l, h)\}, \text{card}(\mu, \Omega) = 1$
R10	$[\![\langle ?v_L, \text{POS}_1\ \texttt{test}\ \text{POS}_2, ?v_R\rangle]\!]_G := \mathbf{E}_T[\![?v_L\ \text{POS}_1\ \texttt{test}\ \text{POS}_2\ ?v_R]\!]_G$
R11	$\mathbf{E}_T[\![?v_L\ \text{POS}_1\ u\ \text{POS}_2\ ?v_R]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\}, \mu(?v_L) = \Pi(\text{POS}_1, t),\ \mu(?v_R) = \Pi(\text{POS}_2, t), t.p = u, t \in G\},$ $\text{card}(\mu, \Omega) = \lvert\{t \mid t \in G, t.p = u, \mu(?v_L) = \Pi(\text{POS}_1, t),\ \mu(?v_R) = \Pi(\text{POS}_2, t)\}\rvert$
R12	$\mathbf{E}_T[\![?v_L\ \text{POS}_1\ \text{TP}(\text{POS}_n, \text{epp}_n)\ \text{POS}_2\ ?v_R]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\}, \exists \mu' \in [\![\Pi(\text{POS}_n, t)\ \text{epp}_n\ ?v_n]\!], \text{dom}(\mu') = \{?v_n\},$ $\mu(?v_L) = \Pi(\text{POS}_1, t),\ \mu(?v_R) = \Pi(\text{POS}_2, t), t \in G\}, \text{card}(\mu, \Omega) = \lvert\{t \mid t \in G, \exists \mu' \in [\![\Pi(\text{POS}_n, t)\ \text{epp}_n\ ?v_n]\!],$ $\text{dom}(\mu') = \{?v_n\}, \mu(?v_L) = \Pi(\text{POS}_1, t),\ \mu(?v_R) = \Pi(\text{POS}_2, t)\}\rvert$
R13	$\mathbf{E}_T[\![?v_L\ \text{POS}_1\ \text{T}(\texttt{EExp})\ \text{POS}_2\ ?v_R]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\}, \mu(?v_L) = \Pi(\text{POS}_1, t), \mu(?v_R) = \Pi(\text{POS}_2, t), t \in G,$ $\texttt{Evaluate}(\texttt{EExp}, t) = \texttt{true}\}, \text{card}(\mu, \Omega) = \lvert\{t \mid t \in G, \texttt{Evaluate}(\texttt{EExp}, t) = \texttt{true},$ $\mu(?v_L) = \Pi(\text{POS}_1, t),\ \mu(?v_R) = \Pi(\text{POS}_2, t)\}\rvert$
R14	$\mathbf{E}_T[\![?v_L\ (\text{POS}_1\ \texttt{test}_1\ \text{POS}_2)\ \&\&\ (\text{POS}_1\ \texttt{test}_2\ \text{POS}_2)\ ?v_R]\!]_G := \mathbf{E}_T[\![?v_L\ (\text{POS}_1\ \texttt{test}_1\ \text{POS}_2)?v_R]\!]_G \bowtie \mathbf{E}_T[\![?v_L\ (\text{POS}_1\ \texttt{test}_2\ \text{POS}_2)?v_R]\!]_G$
R15	$\mathbf{E}_T[\![?v_L\ (\text{POS}_1\ \texttt{test}_1\ \text{POS}_2)\ ||\ (\text{POS}_1\ \texttt{test}_2\ \text{POS}_2)\ ?v_R]\!]_G := \mathbf{E}_T[\![?v_L\ (\text{POS}_1\ \texttt{test}_1\ \text{POS}_2)?v_R]\!]_G \cup \mathbf{E}_T[\![?v_L\ (\text{POS}_1\ \texttt{test}_2\ \text{POS}_2)?v_R]\!]_G$
R16	$\mathbf{E}_T[\![?v_L\ \text{POS}_1\ !\texttt{test}\ \text{POS}_2\ ?v_R]\!]_G := \{\mu \mid \text{dom}(\mu) = \{?v_L, ?v_R\},\ \mu(?v_L) = \Pi(\text{POS}_1, t),\ \mu(?v_R) = \Pi(\text{POS}_2, t), t \in G\} -$ $\mathbf{E}_T[\![?v_L\ \text{POS}_1\ \texttt{test}\ \text{POS}_2 ?v_R]\!]_G$

the production [110] in the SPARQL grammar [5] where `BuiltInCall`[6] is substituted with a new production called `Extended-BuiltInCall`, which enables to use in EPPs tests available in SPARQL as built-in conditions also augmented with positions (POS). Built-in conditions are constructed using elements of the set $\mathbf{I} \cup \mathbf{L}$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), (in)equality symbol(s) ($=,<,>,\leqslant,\geqslant$), unary (e.g., `isURI`,) and binary (e.g., `STRSTARTS`) functions. Tests can also be combined by using the logical operators AND (&&), OR (||) and NOT (!). We refer to non-recursive EPPs (nEPPs) as those expressions that do not include closure operators (i.e., * and $^+$) and set-semantics repetitions ($\{l,h\}$). The reader can refer to the Website of the EPPs project[7] for further details about the implementation.

3.1.1. Positions and Tests

To clarify the intuition behind tests and positions, we introduce the function $\Pi(\text{POS}, t)$, which projects the element in position POS of a triple t. If we have t=$\langle \texttt{u}_1, \texttt{p}_1, \texttt{u}_2\rangle$, the test $\texttt{T}(\_\texttt{p}{=}\texttt{p}_1)$ it is translated to

[5] http://www.w3.org/TR/sparql11-query/#rExpression
[6] http://www.w3.org/TR/sparql11-query/#rBuiltInCall
[7] http://extendedpps.wordpress.com

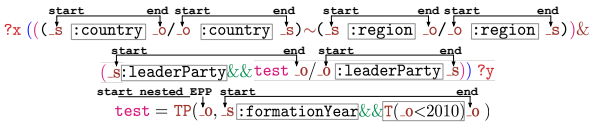


Fig. 7. Expression in Example 4 with positions.

$\Pi(\_\texttt{p}, \langle \texttt{u}_1, \texttt{p}_1, \texttt{u}_2\rangle){=}\texttt{p}_1$ that checks $\texttt{p}_1{=}\texttt{p}_1$, and, in this case, returns true; however, it returns false for $\texttt{T}(\_\texttt{o}{=}\texttt{u}_3)$. Fig. 7 shows the expression from Example 4 including default positions and positions to traverse backward edges. Note that the subexpression (`_o:leaderParty_s`) means that the edge `:leaderParty` is traversed from the object to the subject and, thus, backward.

3.2. Extended Property Paths Semantics

We now introduce the semantics of EPPs in terms of SPARQL. We use the function $[\![\langle \alpha, \text{epp}, \beta\rangle]\!]_G$ where instead of a PP expression `elt` now appears an EPP expression epp. This semantics lays the foundations for the translation algorithm (see Section 4) that given a (concise) nEPP expression produces a semantically equivalent (more verbose) SPARQL query. In the semantics shown in Table 3 we only report the case $\alpha, \beta \in \mathcal{V}$ (and use the symbols $?v_L$ and $?v_R$ to de-

Function EALP1$(\gamma, \texttt{epp}, G, l, h)$
Input: $\gamma \in \mathbf{I}$, epp is an EPP expression,
G is an RDF graph, l, h are integer s.t. $h \leqslant l$

1: $Visited = \varnothing$
2: $i = 0$
3: $\Gamma = \{\gamma\}$
4: **while** $i < l$ **do**
5: $\bar{\Gamma} = \varnothing$
6: **for all** $u \in \Gamma$ **do**
7: $\bar{\Gamma} = \bar{\Gamma} \cup \{\mu(?y) \mid \mu \in [\![(\langle ?x, \texttt{epp}, ?y\rangle)]\!]_G$ such that $\mu(?x) = u, ?x, ?y \in \mathcal{V}\}$
8: $i = i + 1$
9: $\Gamma = \bar{\Gamma}$
10: **if** $h = {}^*$ **then**
11: EALP2$(\Gamma, \texttt{epp}, Visited, G, h)$
12: **else**
13: EALP2$(\Gamma, \texttt{epp}, Visited, G, h\text{-}l)$
14: **return** *Visited*

Function EALP2$(\Gamma, \texttt{epp}, Visited, G, h)$
Input: $\Gamma \subseteq \mathbf{I}$, epp is an EPP expression, $Visited \subseteq \mathbf{I}$,
G is an RDF graph, h is an integer

1: **for all** $u \in \Gamma$ s.t. $u \notin Visited$ **do**
2: add u to $Visited$
3: **if** $h = {}^*$ or $h > 0$ **then**
4: $\bar{\Gamma} = \{\mu(?y) \mid \mu \in [\![(\langle ?x, \texttt{epp}, ?y\rangle)]\!]_G$, such that $\mu(?x) = u, ?x, ?y \in \mathcal{V}\}$
5: **if** $\bar{\Gamma} \neq \varnothing$ **then**
6: **if** $h = {}^*$ **then**
7: EALP2$(\bar{\Gamma}, \texttt{epp}, Visited, G, {}^*)$
8: **else**
9: EALP2$(\bar{\Gamma}, \texttt{epp}, Visited, G, h\text{-}1)$

Fig. 8. Auxiliary functions used to define the semantics EPP expressions.

note the left and right variable in the pattern); the other cases (e.g., $\alpha \in \mathbf{I}, \beta \in \mathcal{V}$) are similar. We denote with t a triple $\langle s, p, o\rangle \in G$; $t.x$ with $x \in \{s, p, o\}$ is used to access an element of the triple. Finally, the notation $\texttt{epp}^i$ is a shorthand for the concatenation (i.e., via the operator '/') of epp i times. A peculiar construct of EPPs is the test POS_1 `test` POS_2, which is handled at a high level by rule R10. In particular tests make usage of the semantic function $\mathbf{E}_T$, which handles the different kinds of tests via rules R11-R16. Moreover, POS_1 and POS_2 denotes the positions (i.e., subject _s, predicate _p or object _o) of the elements of a triple that have to be projected. We now provide some examples of R11-R13 by using the graph in Figure 1.

Example 13. *Consider the following EPP expression:* _o :leaderParty _s. *This type of test is handled via rule R11 in Table 10 and considers all triples t∈G where* :leaderParty *appears in predicate position. In the set of mapping obtained by applying rule R11 on such triples, the left variable (i.e,* $?v_L$*) is bound to the object (since* POS_1=_o*) while the right variable (i.e,* $?v_R$*) to the subject (since* POS_2 = _s*). In particular, the set of mappings is:* $\{(?v_L \rightarrow$:Democratic_Party, $?v_R \rightarrow$:Rome$), (?v_L \rightarrow$:Democratic_Party, $?v_R \rightarrow$:Florence$), (?v_L \rightarrow$:Socialist_Party, $?v_R \rightarrow$:Carrara$)\}$.

Example 14. *Consider the following EPP expression:* _s TP(_o,:leaderParty) _o, *which is handled via rule R12 in Table 10. In this case, the triples* $t \in G$ *considered are those such that from their object, the EPP* :leaderParty *has a solution* $(\exists\, \mu' \in [\![\Pi(\text{POS}_n, t)\ \ \texttt{epp}_n\ ?v_n]\!]$*). In more detail, these triples have one among* :Rome, :Florence *or* :Carrara *in the object position (in particular, are the two triples* $\langle$:Rome, :airbus, :Florence$\rangle$ *and* $\langle$:Florence, :italo, :Carrara$\rangle$*). To obtain the set of mappings from these triples, the left variable in rule R12 (i.e,* $?v_L$*) will be bound to their subject (since* POS_1=_s*) and the right variable (i.e,* $?v_R$*) to their object (since* POS_2 = _o*). Overall, the set of mappings is:*$\{(?v_L \rightarrow$:Rome, $?v_R \rightarrow$:Florence$), (?v_L \rightarrow$:Florence, $?v_R \rightarrow$:Carrara$)\}$.

Example 15. *Consider the following EPP expression:* _s T(_o>400000) _p *handled via rule R13 in Table 10. The set of triples* $t \in G$ *that are of interest in this case are those in which the object has a value greater than* 400000 *(*Evaluate$(EExp, t) =$ true*). These are:* $\langle$:Rome, ::population, 2874034$\rangle$, $\langle$:Murcia, :population, 436870$\rangle$ *and* $\langle$:Miami, :population, 419777$\rangle$*. In the set of mappings obtained applying rule R13 on these triples, the left variable (i.e,* $?v_L$*) is bound to the subject (since* POS_1=_s*) and the right variable (i.e,* $?v_R$*) to the predicate (since* POS_2 = _p*). The set of mappings is:* $\{(?v_L \rightarrow$:Rome, $?v_R \rightarrow$:population$), (?v_L \rightarrow$:Murcia, $?v_R \rightarrow$:population$),$ $(?v_L \rightarrow$:Miami, $?v_R \rightarrow$:population$)\}$.

Closure and Repetitions. The closure operators '*' and '+' and set-semantics repetitions ($\{l, h\}$) use the function EALP1 (Extended Arbitrary Length Paths) shown in Fig. 8, which extends the ALP1 function

Table 4
Fragments of SPARQL, using the SELECT query form, considered in this paper.

Fragment	⋈ (AND)	∪ (UNION)	− (MINUS)	FILTER	PP	EPP	ALP1	EALP1
$S^{\{\bowtie\}}$	x							
$S^{\{\bowtie,\cup,\text{FILTER}\}}$	x	x		x				
$S^{\{\bowtie,\cup,-,\text{FILTER}\}}$	x	x	x	x				
$S^{\{\bowtie,\cup,\text{FILTER,ALP1}\}}$	x	x		x			x	
$S^{\{\bowtie,\cup,-,\text{FILTER,EALP1}\}}$	x	x	x	x				x
$S^{\{\bowtie,\cup,\text{PP}\}}$	x	x			x			
$S^{\{\bowtie,\cup,\text{EPP}\}}$	x	x				x		
$S^{\{\bowtie,\cup,\text{FILTER,PP,ALP1}\}}$	x	x		x	x		x	
$S^{\{\bowtie,\cup,\text{FILTER,EPP,EALP1}\}}$	x	x		x		x		x

defined in the W3C spec. (see Fig. 6). In particular, `EALP1` handles the set-semantic repetitions of an EPP expression `epp` between a minimum l and a maximum h of times. The closure operators '*' and '+' are handled by setting $l = 0$ (respectively, $l = 1$) and $h = *$. `EALP1` uses the global variable *Visited* to keep track of the nodes already checked that belong to the results. The main task carried out by `EALP1` is to skip the first $l - 1$ navigational steps so that the results are stored in *Visited* starting from the step l via `EALP2`. We now further clarify the behavior of `EALP1` and `EALP2`.

Example 16. *Consider the expression* `:Carrara (:twinned)*` ?e *evaluated according to EALP1 on the graph in Figure 1. As the expression involves the closure operator, EALP1 is called with the following parameters:* `EALP1(:Carrara, :twinned, G, 0, *)`. `EALP1` *initializes the global variable Visited to the empty set and the variable* Γ *to the set* {`:Carrara`} *(lines 1 and 3). The* **while** *cycle is never executed as* $l = 0$. *Since* $h = *$ *the function* `EALP2` *is called as:* `EALP2({:Carrara}, :twinned, ∅, G, *)`. *At this point, when the* **for** *cycle starts we have that* Γ = {`:Carrara`} *and Visited* = $\varnothing$ *(line1). Then,* `:Carrara` *is added to Visited (line 2) and the set* $\bar{\Gamma}$ *is computed, which includes all nodes reachable from* `text:Carrara` *by traversing a* `:twinned` *edge (line 4), that i,* $\bar{\Gamma}$={`:Grasse`}. *At this point,* `EALP2` *is called again with the parameters:* `EALP2({:Grasse}, :twinned, {:Carrara}, G, *)` *(line 6);* Γ *contains one IRI (i.e.,* `:Grasse`*) and the* **for** *cycle is executed only once:* `:Grasse` *is added to Visited (line 2) and* $\bar{\Gamma}$ = {`:Migliarino`, `:Murcia`} *(line 4).* `EALP2` *is called again with the parameters:* `EALP2({:Migliarino, :Murcia}, :twinned, {:Carrara, :Grasse}, G, *)` *(line 6). This time* Γ *contains two IRIs (i.e.,* `:Migliarino` *and* `:Murcia`*) and the* **for** *cycle is executed twice one for each such IRIs. With* `:Migliarino` *we have that* $\bar{\Gamma} = \varnothing$ *and* `EALP2` *is not called anymore.*

With `:Murcia` *we have that* $\bar{\Gamma}$ = {`:Miami`} *and* `EALP2` *is called as:* `EALP2({:Miami}, :twinned, {:Carrara, :Grasse, :Migliarino, :Murcia}, G, *)`. *Since* Γ *contains one IRI only (i.e.,* `:Miami`*) the* **for** *cycle is executed only once:* `:Miami` *is added to Visited,* $\bar{\Gamma} = \varnothing$ *and* `EALP2` *is not called anymore. Since Visited is a global variable, the result of the execution is:* {`:Carrara`, `:Grasse`, `:Migliarino`, `:Murcia`, `:Miami`}. ◀

Example 17. *Consider the EPP expression* `:Carrara (:twinned){1,2}` ?e *evaluated on the graph in Figure 1. This time EALP1 is called with the parameters:* `EALP1(:Carrara, :twinned, G, 1, 2)`. `EALP1` *initializes the global variable Visited to the empty set and the variable* Γ *to the set* {`:Carrara`} *(lines 1 and 3). The* **while** *cycle is executed for one iteration only since* $l = 1$. *The set* $\bar{\Gamma}$ *is computed starting from* `:Carrara`*; in this case it is* $\bar{\Gamma}$={`:Grasse`}. `EALP2` *will be called on this set. In particular, since* $h = 2$ *the function* `EALP2` *is called with the following parameters:* `EALP2({:Grasse}, :twinned, ∅, G, 1)`. *The* **for** *cycle is executed only once, since* Γ = {`:Grasse`} *and Visited* = $\varnothing$ *(line1). After the execution* $\bar{\Gamma}$ = {`:Migliarino`, `:Murcia`} *and* `EALP2` *is called again as:* `EALP2({:Migliarino, :Murcia}, :twinned, {:Grasse}, G, 0)`. *As* $h = 0$ `:Migliarino` *and* `:Murcia` *are added to Visited; however, the* **for** *cycle will not be executed. The result is* {`:Grasse`, `:Migliarino`, `:Murcia`}. ◀

3.3. Fragments of SPARQL Considered

In the remainder of the paper we will focus on the SELECT query form and consider the SPARQL fragments shown in Table 4. These fragments are built using combinations of: (i) the operators ⋈ (AND), ∪ (UNION), − (MINUS), FILTER; (ii) the functions `ALP1` and `EALP1` (introduced in Section 3.2); (iii) PP and EPP languages.

4. Translation of nEPPs into SPARQL

The goal of this section is to formalize and describe a translation algorithm that given a non-recursive EPPs (nEPP) translates it into a SPARQL query. Our approach follows the same line of thought as the SPARQL standard for the translation of non-recursive property paths (nPPs) into SPARQL queries. As a by-product, our study formalizes the informal procedure mentioned in the W3C specification for non-recursive PPs (see [10], Section 9.3) and does it for a more expressive language.

4.1. Translation Algorithm: an overview

We now provide an overview of the translation algorithm $\mathcal{A}^t$. The algorithm takes as input a nEPP pattern $\mathcal{P}$=$\langle\alpha, \texttt{epp}, \beta\rangle$ and produces a semantically equivalent SPARQL query Q_e. The algorithm involves three main steps: **(i)** *building of the operational tree*; **(ii)** *propagation of variables and terms* along the nodes of the operational tree; **(iii)** *application of the translation rules*. Each of the three steps is discussed in detail in the following three subsections.

4.1.1. Operational Tree

Let $\mathcal{P}$=$\langle\alpha, \texttt{epp}, \beta\rangle$ be a nEPP pattern and $\tau_{\mathcal{P}}$ be the parse tree associated to the expression epp. Let $T = \{\texttt{root}, \hat{}, \&, \sim, |, /, \texttt{iri}, \texttt{TP}, \texttt{T}, \texttt{test}, ||, \&\&, !\}$[8] be the set of node types, $\Omega = \{\texttt{b}, \texttt{e}, \texttt{m}, \texttt{s}, \texttt{p}, \texttt{o}\}$ and $\Delta = \{\texttt{pos}_1, \texttt{pos}_2, \texttt{pos}\}$ be two sets of attributes. The operational tree $\pi_{\mathcal{P}} = (V, E, type, id, \omega, \delta)$ associated to the pattern $\mathcal{P}$ is a binary, ordered, labeled, rooted tree, where V is the set of nodes, $E \subset V \times V$ the set of edges, $type : V \rightarrow T$ is a function that associates to each node a type, id a function that associates to each node a unique identifier, $\omega : V \times \Omega \rightarrow \mathcal{U} \cup \mathcal{L} \cup V$ a function that associates to a pair (v, a), such that $v \in V$ and $a \in \Omega$ a URI, a literal or a variable identifier. Finally, $\delta : V \times \Delta \rightarrow \{\_s, \_p, \_o\}$ is a function that associates to a pair (v, a), such that $v \in V$ and $a \in \Delta$ a position symbol. The nodes of the operational tree can be subdivided in two categories: *operational nodes* that are labeled with the syntactic symbols $\hat{}, \&, \sim, |, /$, and *test nodes* that are labeled with u, TP, T(EExp), test, !, $||, \&\&, !$. Figure 9 reports, for each type of node, its set of attributes (i.e., the domain of the functions ω and δ). The attributes b (start) and e (end) denote the starting and ending points of the operation represented by each operational node. Concatenation nodes (/) have the additional attribute m that maintains the join variable.

	Node Attributes									
	id	b	e	m	s	p	o	pos1	pos2	pos
root	x	x	x							
^	x	x	x							
/	x	x	x	x						
&	x	x	x							
~	x	x	x							
|	x	x	x							
/	x	x	x							
test	x				x	x	x	x	x	
TP	x				x	x	x			x
T	x				x	x	x			
iri	x				x	x	x			
&&	x				x	x	x			
||	x				x	x	x			
!	x				x	x	x			

Fig. 9. Node attributes in the operational tree.

Test nodes have attributes s, p, o denoting the subject, predicate and object of the triple on which the test is to be checked. Additionally, since the test node test encodes a triple traversal it has also the attributes start (POS_1) and end (POS_2) that can be valued with _s, _p or _o, denoting the position of beginning and ending of the traversal. Finally, test nodes TP have the additional attribute POS (also valued with one among _s, _p or _o) that indicates the beginning of the existential test wrt the last triple.

The root r of $\pi_{\mathcal{P}}$ is a special node of type root having $id(r) = 0$ and attributes b (start) and e (end) valued with the pattern endpoints, that is, $\omega(r, \texttt{b}) = \alpha$ and $\omega(r, \texttt{e}) = \beta$. To build the operational tree, the nodes of the parse tree $\tau_{\mathcal{P}}$ are visited according to a pre-order traversal, that is, the parent first, then the left child and finally the right child, if one exists. In what follows, the function *parent* indicates the parent of a node. Moreover, the function *coor* applied to each node of $\tau_{\mathcal{P}}$ returns exactly one node of $\pi_{\mathcal{P}}$. For each node v of $\tau_{\mathcal{P}}$ visited, we have:

(1) If v is the root of $\tau_{\mathcal{P}}$, then a node c is added as the only child of r with $id(c) = 0\_0$. If v is a left child of some node of $\tau_{\mathcal{P}}$, a node c is added as the left child of $corr(parent(v))$ and $id(c) = id(parent(c)) + \text{“\_0”}$. If v is a right child of some

[8] Note that the ? and {{·}} syntactic operators are omitted since they are only syntactic sugar and can be rewritten by using | and /.

node of $\tau_{\mathcal{P}}$, then a node c is added as the right child of $corr(parent(v))$ with $id(c) = id(parent(c)) +$ "_1". Furthermore:

(1.1) If v is an operational node, then c has the same type of v and all its attributes are initialized with fresh variables. Moreover, $corr(v)$=c.

(1.2) If v is a test node and $corr(parent(v))$=c'' is an operational node, then c has type `test`, its attributes `s,p,o` are initialized with fresh variables and $\texttt{pos}_1$ and $\texttt{pos}_2$ are set to be equal to the position used in the test (or to the default positions if they are omitted). Moreover, a node c' is added as the only child of c with the same type of v and $id(c') = id(c) +$ "_0". Moreover, its attributes `s`, `p` and `o` are initialized with fresh variables. If $type(c') =$ `TP` then the attribute `pos` is initialized with the value specified in the existential test. Note that $corr(v) = c'$.

(1.3) If v is a test node, and c'=$corr(parent(v))$ is a test node, then c has the same type of v and all its attributes are initialized with fresh variables. We have that $corr(v)$=c.

The operational tree for the nEPP pattern of Example 2 is shown in Fig. 11 (a). Fresh variables for the attributes of a node n are generated using the template: ?`X`+_+*id*`(n)`, where `X`$\in \{$`b`, `e`, `m`, `s`, `p`, `o`$\}$, with + denoting string concatenation.

4.1.2. Propagation of Variables and Terms

Given an operational tree for a pattern $\mathcal{P}$, each of its nodes has attributes valued with variables or terms. The translation algorithm takes care of propagating these variables and terms during the generation of the SPARQL query associated to $\mathcal{P}$ via the Procedure `Propagate` (Fig. 10), which takes as input a node (the root at the beginning) and propagates values to its children. As an example, Fig. 11 (b) shows the operational tree after the propagation on the tree in Fig. 11 (a). An an example, by looking at R2 in the EPPs semantics shown in Table 3, we notice that path concatenation (/) makes usage of the join operator; specifically, it requires to introduce a fresh join variable in the translation. The propagation algorithm guarantees that both children of the concatenation node use the same join variable by applying the propagation rules reported in Fig. 10 (lines 25-30). By looking at Fig 11, such rules translates in the fact that the attribute `b` of node 0_0_0 of Fig. 11 (b) is propagated to the attribute

Function `Propagate`(n)
Input: n, a node of the *operational tree*.
Result: update n's children attributes.

1: Let n_i=$n.child(i)$
2: **if** n is a `test` node **then**
3: **if** n is `TP` **then**
4: **if** n_1 is a `test` node **then**
5: $n1.\text{POS}_1$=$n.\text{POS}$
6: **else** n_1.b=$n.\text{POS}$
7: **else**
8: **if** $n.\text{POS}_1$ = _s and $n.\text{POS}_2$ = _o **then**
9: $n_i.X$=$n.X$, $i \in \{1,2\}, X \in \{\text{s},\text{p},\text{o}\}$
10: **else if** $n.\text{POS}_1$ = _s and $n.\text{POS}_2$ = _p **then**
11: $n_i.s$=$n.s$, $n_i.p$=$n.o$, $n_i.o$=$n.p$ $i \in \{1,2\}$
12: **else if** $n.\text{POS}_1$ = _p and $n.\text{POS}_2$ = _s **then**
13: $n_i.s$=$n.p$, $n_i.p$=$n.o$, $n_i.o$=$n.s$ $i \in \{1,2\}$
14: **else if** $n.\text{POS}_1$ = _p and $n.\text{POS}_2$ = _o **then**
15: $n_i.s$=$n.p$, $n_i.p$=$n.s$, $n_i.o$=$n.o$ $i \in \{1,2\}$
16: **else if** $n.\text{POS}_1$ = _o and $n.\text{POS}_2$ = _s **then**
17: $n_i.s$=$n.o$, $n_i.p$=$n.p$, $n_i.o$=$n.s$ $i \in \{1,2\}$
18: **else if** $n.\text{POS}_1$ = _o and $n.\text{POS}_2$ = _p **then**
19: $n_i.s$=$n.o$, $n_i.p$=$n.s$, $n_i.o$=$n.p$ $i \in \{1,2\}$
20: **else if** n is ^ **then**
21: **if** n_1 is a `test` node **then**
22: $n_1.\text{POS}_1$=n.e; $n_1.\text{POS}_2$=n.b
23: **else** n_1.b=n.e; n_1.e=n.b
24: **else if** n is / **then**
25: **if** n_1 is a `test` node **then**
26: $n_1.\text{POS}_1$=n.b; $n_1.\text{POS}_2$=n.m
27: **else** n_1.b=n.b; n_1.e=n.m
28: **if** n_2 is a `test` node **then**
29: $n_2.\text{POS}_1$=n.m; $n_2.\text{POS}_2$=n.e
30: **else** n_2.b=n.m; n_2.e=n.e;
31: **else**
32: **if** n_i is a `test` node, $i \in \{1,2\}$ **then**
33: $n_i.\text{POS}_1$=n.b; $n_i.\text{POS}_2$=n.e
34: **else** $n_i.X$=$n.X$, $X \in \{\text{b},\text{e}\}$
35: $\forall i$ `Propagate`(n_i)

Fig. 10. Propagation of variables and terms.

`s` of node 0_0_0_0; the attribute `e` of node 0_0_0 is propagated to the attribute `e` of node 0_0_0_1; and the value of the attribute `m` (that is a fresh variable) is propagated to the attribute `o` of node 0_0_0_0 and to the attribute `s` of the node 0_0_0_1.

Furthermore, the propagation phase also ensures that the test are executed on the correct position of the triple and that the endpoints are correctly selected by applying the rules reported in Fig. 10 lines 8-19. By looking at Fig 11, the rule in lines 16-17 trans-

Table 5

Translating nEPPs into SPARQL (code). EBC extends SPARQL BuiltInCall with EPPs tests also augmented with positions (POS). nEPPs with double-brace path repetitions (epp{{l,h}}) are first translated into equivalent nEPPs via unions of concatenations.

$\mathbf{R_m}$	$\Theta^{\mathcal{P}}(\texttt{root})$=‘SELECT’ root.*b* root.*e* ‘WHERE {’$\Theta^{\mathcal{P}}(\texttt{root}.child(1))$‘}’
R0	$\Gamma(n)$:=*n*.s *n*.p *n*.o‘.’
R1	$\Theta^{\mathcal{P}}(n^{\hat{}})$:=$\Theta^{\mathcal{P}}(n.child(1))$
R2	$\Theta^{\mathcal{P}}(n^{/})$:=$\Theta^{\mathcal{P}}(n.child(1))\ \Theta^{\mathcal{P}}(n.child(2))$
R3	$\Theta^{\mathcal{P}}(n^{\mid})$:=‘{’$\Theta^{\mathcal{P}}(n.child(1))$‘} UNION{’$\Theta^{\mathcal{P}}(n.child(2))$‘}’
R4	$\Theta^{\mathcal{P}}(n^{\&})$:=$\Theta^{\mathcal{P}}(n.child(1))\ \Theta^{\mathcal{P}}(n.child(2))$
R5	$\Theta^{\mathcal{P}}(n^{\sim})$:=‘{’$\Theta^{\mathcal{P}}(n.child(1))$‘} MINUS{’$\Theta^{\mathcal{P}}(n.child(2))$‘}’
R6	$\Theta^{\mathcal{P}}(n^{\texttt{test}})$:= $\Theta^{\mathbf{t}}(n.child(1))$
R7	$\Theta^{\mathbf{t}}(n^{u})$:=$\Gamma(n)$ ‘FILTER(’*n*.p‘=’*u*‘)’
R8	$\Theta^{\mathbf{t}}(n^{\text{EBC}})$:=$\Gamma(n)$ ‘FILTER’ EBC
R9	$\Theta^{\mathbf{t}}(n^{\text{TP}})$:=$\Gamma(n)$ ‘FILTER EXISTS’{$\Theta^{\mathcal{P}}(n.child(1))$‘}’
R10	$\Theta^{\mathbf{t}}(n^{!})$:=$\Gamma(n)$ ‘MINUS{’ $\Theta^{\mathbf{t}}(n.child(1))$‘}’
R11	$\Theta^{\mathbf{t}}(n^{\hat{}})$:=$\Theta^{\mathbf{t}}(n.child(1))$
R12	$\Theta^{\mathbf{t}}(n^{\&\&})$:=$\Theta^{\mathbf{t}}(n.child(1))\ \Theta^{\mathbf{t}}(n.child(2))$
R13	$\Theta^{\mathbf{t}}(n^{\mid\mid})$:=‘{’$\Theta^{\mathbf{t}}(n.child(1))$‘} UNION{’$\Theta^{\mathbf{t}}(n.child(2))$‘}’

lates in the fact that the attribute s of node 0_0_0_0 of Fig. 11 (b) is propagated to the attribute o of node 0_0_0_0_0; the attribute p of node 0_0_0_0 is propagated to the attribute p of node 0_0_0_0_0; and the value of the attribute o is propagated to the attribute s of node 0_0_0_0_0.

4.1.3. Generating SPARQL code

The last step of the translation algorithm takes as input the result of the previous phases, that is, an operational tree where all attribute values are filled with the correct values (i.e., RDF terms, fresh variables and the variables or terms α and β derived from the nEPP pattern in input). At this point, to generate the SPARQL code for a given nEPP pattern, the translation algorithm leverages the translation rules shown in Table 5. The translation uses two functions: $\Theta^{\mathcal{P}}(\cdot)$ that handles general nEPP expressions and $\Theta^{\mathbf{t}}(\cdot)$ that handles tests.

The translation algorithm applies the rules starting from the root and proceeding via a pre-order depth-first traversal of the operational tree. In a nutshell, the translation proceeds as follows: rule R_m generates the outermost part of the final SPARQL query; moreover, it projects the variables stored in the attributes root.b and root.e; for sake of presentation we assume that $\alpha, \beta \in \mathcal{V}$ in the input pattern $\mathcal{P}$=$\langle \alpha, \texttt{epp}, \beta \rangle$. Path concatenation is handled via rule **R2** and is semantically dealt with via the join operator (**R2** in Table 3). Each of the two operands of the join is one of the children of the node labeled with / in the operational tree. The join operator is also used to handle path conjunction (**R4**). The difference with path concatenation resides in the usage of the variables; indeed, by looking at Table 3 we note that concatenation makes usage of a (fresh) join variable stored in the attribute m of the concatenation node of the operational tree, while path conjunction is evaluated from the same endpoints for both conjuncts. In the same spirit, we note that path difference (**R5**) is translated by using the $-$ operator in the SPARQL algebra (see Table 3) that syntactically corresponds to the MINUS operator. Path union (**R3**), which uses the union operator from the SPARQL algebra, is translated using its SPARQL syntactic counterpart, that is, UNION. Reverse path (**R1**) is handled by switching, in the propagation phase, *n*'s variables when propagated to its child node $n.child(1)$. Tests are handled by a combination of FILTER and FILTER EXISTS along with UNION to deal with disjunction of tests, join to deal with conjunction of tests and MINUS to deal with negated tests. To given a hint, a nEPP pattern containing a single triple pattern of the for $\langle$?b, u, ?e$\rangle$ where $u \in \mathbf{I}$ in translated via rule **R7** as SELECT ?b ?e WHERE {?b ?p_0_0 ?e.FILTER(?p_0_0=u)} where ?p_0_0 is a variable automatically generated. A nEPP pattern containing a EBC (Extended-BuiltInCall) is translated via rule **R8** by using a FILTER expression applied to the specified EBC. For example, the nEPP pattern $\langle$?b, _s T(isLiteral(_o)) _o, ?e$\rangle$ is translated as

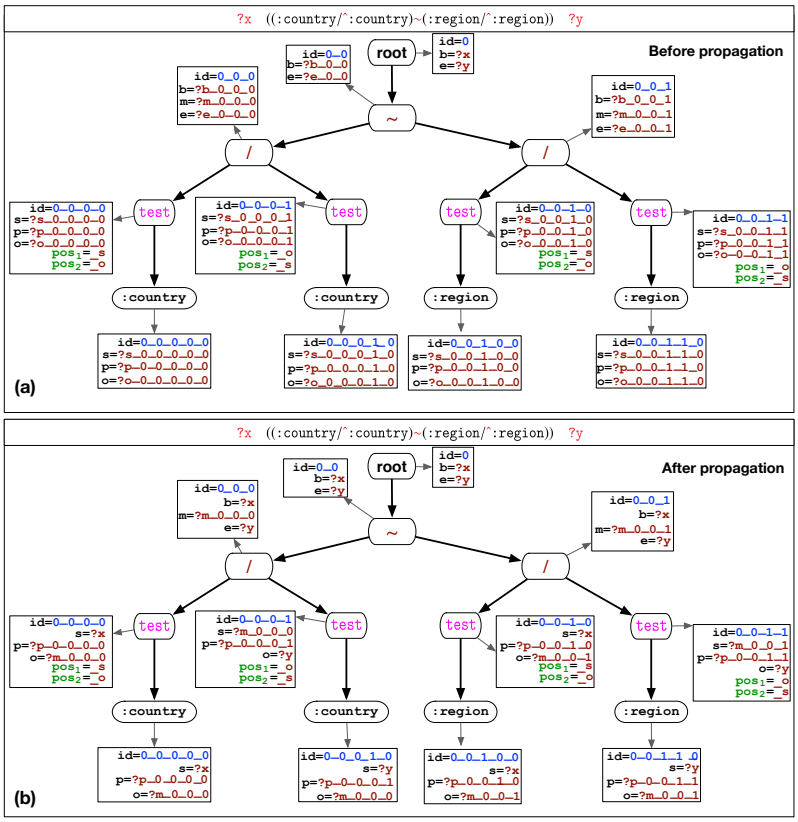


Fig. 11. Operational tree for Example 2 before (a) and after (b) the propagation phase.

`SELECT ?b ?e WHERE {?b ?p_0_0 ?e.FILTER(isLiteral(?e))}` where the parameter of the `isLiteral` BuiltInCall is substituted during the translation with the variable `?e`. Nested nEPPs are handled via rule **R9** and are basically existential tests; test whether the nested nEPP has a solution (see also rule **R12** in Table 3).

Example 18. (**Translating nEPPs into SPARQL**). Consider the nEPP pattern in Example 2. The corresponding operational tree is reported in Fig. 11 (a). The operational tree obtained after the application of the procedure `Propagate` is shown in Fig. 11 (b). As an example, by looking at the *operational node* with `id`=0_0_0 and labeled with / in Fig. 11 (a) and (b) we can see that `Propagate` updated the values of the attributes `s` and `o` of its children 0_0_0_0 and 0_0_0_1 with values in the attributes `b` and `e` of 0_0_0. Applying the translation rules to the operational tree in Fig. 11 (b) means starting from `root` (node 0) and triggering rule $\mathbf{R_m}$ (see Table 5), which generates the outermost part of the final SPARQL translation: $\Theta^{\mathcal{P}}(0)$=SELECT ?x ?y WHERE{ $\Theta^{\mathcal{P}}(0\_0^{\sim})$ }. Then, the node with `id`=0_0 and labeled with $\sim$ is visited; this triggers **R5**: $\Theta^{\mathcal{P}}(0\_0^{\sim})$= '{'$\Theta^{\mathcal{P}}(0\_0\_0^{/})$'}' MINUS '{'$\Theta^{\mathcal{P}}(0\_0\_1^{/})$'}'. The translation uses `MINUS` to reflect the semantics of EPPs dealing with path difference while `test` (e.g., 0_0_0_0) is reflected via the `FILTER` operator. Visiting the node 0_0_0 triggers **R2**. The translation continues with:

$$\Theta^{\mathcal{P}}(0\_0\_0^{/}) = \Theta^{\mathcal{P}}(0\_0\_0\_0^{\texttt{test}})\Theta^{\mathcal{P}}(0\_0\_0\_1^{\texttt{test}});$$
$$\Theta^{\mathcal{P}}(0\_0\_0\_0^{\texttt{test}}) = \Theta^{\mathbf{t}}(0\_0\_0\_0\_0^{:\texttt{country}});$$
$$\Theta^{\mathbf{t}}(0\_0\_0\_0\_0^{:\texttt{country}}) = \texttt{?x ?p\_0\_0\_0\_0 ?m\_0\_0\_0.}$$
$$\texttt{FILTER(?p\_0\_0\_0\_0 = :country).}$$

The translation continues until no more nodes of the operational tree have to be visited and gives:

```
SELECT ?x ?y WHERE {
{?x ?p_0_0_0_0 ?m_0_0_0.
 ?y ?p_0_0_0_1 ?m_0_0_0.
FILTER(?p_0_0_0_0=:country)
FILTER(?p_0_0_0_1=:country)}
MINUS
{?x ?p_0_0_1_0 ?m_0_0_1.
?y ?p_0_0_1_1 ?m_0_0_1.
FILTER(?p_0_0_1_0=:region)
FILTER(?p_0_0_1_1=:region)} }
```

Table 6
Languages and translations into SPARQL for plain RDF.

Navigational Core	Extended Processor	Fragment	SPARQL Fragment
$p \in \mathbf{I}$	No	R1 in Fig. 5	$S^{\{\bowtie\}}$
nPP	No	R1-R5 in Fig. 5	$S^{\{\bowtie,\cup,\text{FILTER}\}}$
nEPP	No	R1-R2, R5-R9, R11-R16 in Table 3	$S^{\{\bowtie,\cup,-,\text{FILTER}\}}$
PP	No	Fig. 5	$S^{\{\bowtie,\cup,\text{FILTER},\texttt{ALP1}\}}$
EPP	Yes	Table 3	$S^{\{\bowtie,\cup,-,\text{FILTER},\texttt{EALP1}\}}$

Discussion about the Translation

Conciseness. EPPs enable to write navigational queries in a more succinct way as compared to SPARQL queries using triple patterns and/or union of graph patterns. Given a nEPPs expression containing a number of fragments (e.g., concatenation, union, predicates) it is interesting to note that its corresponding translation in SPARQL is always more verbose. Consider for instance the nEPPs pattern `?x p1/p2 ?y`; here, concatenation avoids to explicitly deal with intermediate variables besides the expression endpoint. Its translation in SPARQL, that is, `?x p1 ?a. ?a p2 ?y` includes 2 instances of the new variable `?a`. Generally, the number of variables necessary to translate a nEPPs into an equivalent SPARQL query is a function of the size of its operational tree. Not only the elimination of intermediate variables increases the succinctness of the expression, but it also eliminates causes of errors when writing queries as one has to check the consistency of variable names.

Uniqueness of the translation. The translation algorithm implements one of infinitely many possible (correct) mappings from a nEPP expression `epp` into a SPARQL query $Q_{\texttt{epp}}$. Each node in the operational tree of `epp` has one equivalent fragment in Table 5. It is easy to see that by adding an arbitrary number of "`&& true`" fragments to FILTERs (in Table 5) we can make versions of the translation that produce infinitely many other (correct) translations.

Benefits. EPPs coupled with the translation procedure bring a significant practical advantage as compared to other navigational extensions of SPARQL (e.g., nSPARQL, cpSPARQL). On one hand, nEPPs can be evaluated over existing SPARQL processors. On the other hand, the machinery presented in this paper could potentially extend the SPARQL standard in an elegant and non-intrusive way; one would need to use our translation algorithm instead of that currently used by the SPARQL standard.

4.2. SPARQL and Navigational Queries

By analyzing the translation algorithm presented in the previous section and the translation rules reported in Table 5, it is possible to identify the precise SPARQL fragment that can express nEPPs.

Lemma 19. nEPPs can be expressed in the SPARQL fragment $S^{\{\bowtie,\cup,-,\text{FILTER}\}}$, which uses `AND`, `UNION`, `MINUS`, `FILTER` and `SELECT`.

In the remainder of this section we analyze for different navigational cores, the SPARQL fragment necessary for its rewriting. The results of the analysis are reported in Table 6. The table shows in the first column (**Navigational Core**) the navigational core, that is, the type of expression allowed in the predicate position of triple patterns; it can be a predicate p, a non-recursive property path (nPP), a property path (PP), a non-recursive EPP (nEPP), and an EPP. The second column (**Exended Processor**) indicates whether the evaluation requires changes into the SPARQL processors. The third and fourth column represents the SPARQL fragment needed for the rewriting. The simplest case (row 1) does not use regular-expression-like extensions and thus no rewriting is needed. The second and fourth rows consider non-recursive and recursive property paths, respectively. These cases are handled, as per W3C specification, via a rewriting into SPARQL and the `ALP1` procedure, respectively. The third and last rows concern nEPPs and full EPPs, respectively. While the former can be translated into SPARQL queries (as shown in the previous section) and evaluated on existing processors, the latter requires the usage of the `EALP1` procedure shown in Fig. 8, currently not available in existing processors.

The most interesting result that emerges from the table is the fragment $S^{\{\bowtie,\cup,\text{FILTER}\}}$ of the current SPARQL standard is already expressive enough to capture nEPPs. Hence, the current W3C standard could readily benefit from the more expressive language of nEPPs without any impact on current SPARQL pro-

cessors. In the following proposition we also mention an even stronger result that can be derived if we drop set-semantics path repetitions in EPPs (R9' in Table 3).

Proposition 20. The full EPPs language can be incorporated in SPARQL using the `ALP1` procedure with the only difference that for the evaluation of the pattern at line 6 in `ALP2` (see Fig. 5) the translation discussed in Section 4 has to be used instead of the translation currently used by the standard.

5. Query-Based Reasoning on Existing SPARQL Processors

The aim of this section is twofold. First, we study the support that EPPs can give to querying under entailment regimes (Section 5.1) with particular emphasis on how to support the entailment regime *on existing SPARQL processors* (Section 5.2). Second, we provide novel results about the expressiveness of the SPARQL standard in terms of query-based reasoning when considering different navigational cores (Section 6.2).

5.1. Capturing the Entailment Regime

In this paper we focus our attention on the ρdf fragment [19, 24]. This fragment considers a subset of RDFS vocabulary consisting in the following elements: `rdfs:domain`, `rdfs:range`, `rdfs:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf` that we denote with `dom`, `range`, `type`, `sc`, and `sp`, respectively. Authors [19] showed that the ρdf semantics is equivalent to that of full RDFS when one focuses on this fragment. Note that ρdf does not consider datatypes that would allow to obtain inconsistent graphs. When considering SPARQL under the ρdf entailment regime, not only the explicit triples in the RDF graph G have to be taken into account but also triples that can be derived from G by the inference rules shown in Table 7. The application of each inference rule enables to obtain a sequence of graphs $G_1, G_2, G_3, ...G_k$ with $G_{i+1} \setminus G_i \neq \varnothing \; \forall i \in [1, ...k-1]$. When $G_{k+1} \setminus G_k = \varnothing$, that is, when the graph is unchanged, the application of the rules stops. The graph G_k is called the closure of G indicated by $cl(G)$.

Definition 21. **(SPARQL and query-based reasoning).** Given a SPARQL pattern $\mathcal{P}$ and an RDF graph G, the evaluation of $\mathcal{P}$ over G under the ρdf semantics is denoted by $[\![\mathcal{P}]\!]^{\rho\text{df}}_G$, while $[\![\mathcal{P}]\!]_{cl(G)}$ denotes the evaluation of $\mathcal{P}$ over the closure of G.

The intended meaning of two semantics differs with respect to the data graph on which the evaluation is performed. In particular, $[\![\mathcal{P}]\!]^{\rho\text{df}}_G$ means that $\mathcal{P}$ is evaluated on the original data graph G, and the results provided should include those generated by considering the ρdf rules. On the other hand, $[\![\mathcal{P}]\!]_{cl(G)}$ means that $\mathcal{P}$ is evaluated on the materialization of the closure of G obtained by applying the ρdf rules. Of course, we expect $[\![\mathcal{P}]\!]^{\rho\text{df}}_G = [\![\mathcal{P}]\!]_{cl(G)}$ to hold.

Most of existing SPARQL processors handle (variants of) ρdf reasoning in the following way: first, compute and materialize the finite polynomial closure of the graph G and then perform query answering on the closure via RDF *simple entailment* regime [25]. It is interesting to point out that materializing all data by computing the closure $cl(G)$ may cause a waste of space in case most of $cl(G)$ is never really used for query answering, apart from the cost of computation and maintenance after updates. Having a mechanism to support entailment regimes while avoiding the computation of $cl(G)$ beforehand can bring a major advantage. Our goal is to study *query-based reasoning*, that is, the possibility to *rewrite* a query into another query that captures ρdf inferences.

Similarly to nSPARQL [11], cpSPARQL [21] and others approaches (e.g., [26, 27]), we identified for each inference rule in the fragment considered, ρdf in our case), an EPP expression encoding it. The translation rules are shown in Table 8. Whenever one wants to adopt the ρdf entailment regime, it is enough to rewrite the input pattern according to these translation rules. The result of the evaluation of the rewritten pattern on G is the same as the result that would be obtained by first computing the closure $cl(G)$ and then evaluating the pattern before the rewriting.

Lemma 22. (**ρdf and SPARQL**). Given a triple pattern $(\alpha, \mathtt{p}, \beta)$ with $\alpha, \beta \in \mathbf{I} \cup \mathcal{V}$ and $\mathtt{p} \in \mathbf{I}$, then for every graph G we have that $[\![(\alpha, \mathtt{p}, \beta)]\!]^{\rho\text{df}}_G = [\![((\alpha, \Phi(\mathtt{p}), \beta))]\!]_G = [\![(\alpha, \mathtt{p}, \beta)]\!]_{cl(G)}$.

Sketch. The proof follows from the fact that rules in Table 8 encode the reasoning rules shown in Table 7. This is immediate to see for rules R1-R4. R5 is composed by the union of three expression, each capturing one of the three possible ways (shown in Fig. 12 (a)-(c)) to derive a `type` in RDFS and corresponding to rules Subclass (a), Domain and Range in Table 7. The first sub-expression in R5 captures the rule in Fig. 12 (a); a new `type` can be derived by finding triples of the form $(z, \mathtt{type}, x)$ and possibly (via *) traveling up

Table 7

The ρdf rule system. Capital letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{Y}$, stand for meta-variables to be replaced by actual terms in $\mathcal{UL}$.

1. **Subclass**:
   (a) $\frac{(\mathcal{A},\texttt{sc},\mathcal{B})\ \ (\mathcal{X},\texttt{type},\mathcal{A})}{(\mathcal{X},\texttt{type},\mathcal{B})}$ (b) $\frac{(\mathcal{A},\texttt{sc},\mathcal{B})\ \ (\mathcal{B},\texttt{sc},\mathcal{C})}{(\mathcal{A},\texttt{sc},\mathcal{C})}$
2. **Subproperty**:
   (a) $\frac{(\mathcal{A},\texttt{sp},\mathcal{B})\ \ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\mathcal{B},\mathcal{Y})}$ (b) $\frac{(\mathcal{A},\texttt{sp},\mathcal{B})\ \ (\mathcal{B},\texttt{sp},\mathcal{C})}{(\mathcal{A},\texttt{sp},\mathcal{C})}$
3. **Domain:**
   $\frac{(\mathcal{A},\texttt{dom},\mathcal{B})\ \ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\texttt{type},\mathcal{B})}$
4. **Range:**
   $\frac{(\mathcal{A},\texttt{range},\mathcal{B})\ \ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{Y},\texttt{type},\mathcal{B})}$

Table 8

Enconding of ρdf inference rules via EPPs.

Rule	ρdf	**Translation** ($\Phi(\cdot)$)
R1	sc	Φ(sc)= sc$^+$
R2	sp	Φ(sp)= sp$^+$
R3	dom	Φ(dom)= dom
R4	range	Φ(range)= range
R5	type	Φ(type)= type/sc* | (T(true)_p/sp* /dom/sc*) | (_o T(true)_p/ sp* /range/sc* _o)
R6	$p \notin$ {sc, sp, dom, range, type}	$\Phi(p)$= (TP(_p, sp*/sp&&T(_o=p)) || T(_p=p))

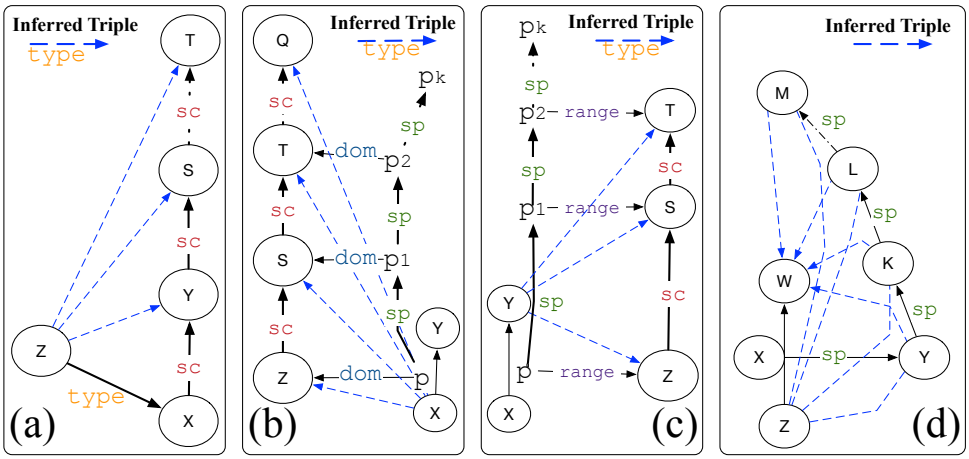


Fig. 12. RDFS inference rules. R5 in Table 8 captures rules (a)-(c) while R6 in Table 8 capture rule (d).

(via sc) the super-classes of x, which is the type of z. The second sub-expression captures the rule depicted in Fig. 12 (b); a new type can be derived by navigating from the subject x to the predicate p and all its possible super-properties (via $*$) and then by finding the dom (i.e., a class) of such predicates, and all possible super-classes (via $*$). A similar reasoning applies for the third sub-expression in R5, which captures the inference rule shown in Fig. 12 (c). As for rule R6 in Table 8, it captures the rule in Fig. 12 (d) corresponding to the rule Subproperty (a) in Table 7. We can notice that the EPP encoding this inference rule includes the union (via ||) of two tests. The second test just checks for triples where p is the predicate; the first performs an existential test (i.e., it uses the nested EPP construct TP) composed by a conjunction (via &&) of two tests,

the first moves to the predicate position of a triple and travels up the property hierarchy (via *) while the second checks that the object of the triple reached is actually p. □

We observe that our translation rules are indeed a translation into the language of EPPs of the NRE expressions that have been shown to capture all the RDFS inferences in Perez et al. [11] (Lemma 5.2). Lemma 22 shows that for an arbitrary pattern there exists a rewriting allowing to capture ρdf inferences. Moreover, it is easy to see that the rewriting can be constructed by using the translation rules in Table 8 in linear time in the size of the pattern. However, in this case (and similarly to nSPARQL and PSPARQL) one would need to use an EPPs processor to capture the inferred triples. This clearly hinders the usage of this machinery in existing processors. Therefore, the research question that we face now is how to support *query-based reasoning on existing processors*.

5.2. Query-based Reasoning on Existing Processors

The idea behind our approach, follows from the observation that closure operators appearing in Table 8 only involve *single predicates* i.e., $\texttt{sc}^+$, $\texttt{sc}^*$, $\texttt{sp}^+$, $\texttt{sp}^*$. Such type of expressions are *property paths* that (taken alone) can be evaluated via the `ALP1` procedure defined in the W3C standard, and implemented in existing processors. Therefore, we need to rewrite the EPPs in Table 8 into SPARQL queries where recursive property paths with single predicates are used. We apply a small variation to the translation algorithm presented in Section 4; the variation consists in *leaving untouched* (single) predicates involving the closure operator (*) used in Table 8. We refer to this variant of the translation algorithm as $\mathcal{A}^t_\rho(\cdot)$.

Lemma 23. Given a triple pattern $\mathcal{P}=\langle\alpha,\texttt{p},\beta\rangle$ with $\alpha,\beta\in\mathbf{I}\cup\mathcal{V}$ and $\texttt{p}\in\mathbf{I}$, $[\![\langle\alpha,\texttt{p},\beta\rangle]\!]_{cl(G)}=[\![\mathcal{A}^t_\rho\langle\alpha,\Phi(\texttt{p}),\beta\rangle]\!]_G$

Proof. The result follows from Lemma 22 which shows that the EPPs rewriting of the ρdf inference rules (via $\Phi(\texttt{p})$) allows to infer the triples in the fragment, and the nEPPs to SPARQL translation (needed in the $\mathcal{A}^t_\rho(\cdot)$ part). □

The above result tells us that an algorithm to perform query-based reasoning works in three steps: (i) apply the translation function $\Phi(\cdot)$; (ii) apply the translation $\mathcal{A}^t_\rho(\cdot)$ over the result of step (i); (iii) evaluate the SPARQL query resulting from (ii) on existing SPARQL processors.

6. Expressiveness Analysis

The aim of this section is to provide novel results about the expressive power of EPPs as compared to PPs (Section 6.1) and the expressiveness of the SPARQL standard in terms of ρdf reasoning when considering different navigational cores (Section 6.2).

6.1. Expressive Power of Extended Property Paths vs. Property Paths

We now investigate the expressiveness of EPPs as compared to PPs. We use the evaluation function $[\![\cdot]\!]_G$ to denote either the evaluation of a PP `elt` (Fig. 5) or EPP `epp` (Table 3). The semantics of the evaluation will be clear from the context. In the next theorem we prove that the language of EPPs is strictly more expressive than PPs[9]. By using the graph in Fig. 13, we will show that there exists an EPP pattern, which is able to distinguish between the node `:b` and the nodes `:c` and `:d`. The same does not hold for PPs; indeed, any PP pattern that provides `:b` as an answer will provide at least an additional answer (either `:c` or `:d`). The rationale behind this result is that PP patterns are not able to tell apart the conjunction of two predicates from the two predicates alone.

Theorem 24. There exists an EPP pattern $\langle\alpha,\texttt{epp},\beta\rangle$ that cannot be expressed as a PP pattern $\langle\alpha,\texttt{elt},\beta\rangle$.

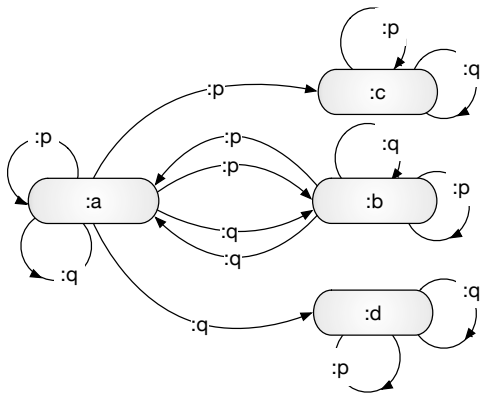


Fig. 13. Graph used to prove Theorem 24.

[9] Even if such result could be obtained by adapting standard results about NREs, we provide, for the sake of completeness, a complete constructive proof.

Proof. Consider the EPP pattern $\pi_1 = \langle ?b, (:p \,\&\, :q)^*, ?e\rangle$ and the graph G in Fig. 13. Let $\Pi_{self}=\{\{?b\rightarrow x, ?e\rightarrow x \mid x \in \{:a, :b, :d, :c\}\}\}$. We have that $[\![\pi_1]\!]_G = \Pi_{self} \cup \{\{?b\rightarrow :a, ?e\rightarrow :b\}, \{?b\rightarrow :b, ?e\rightarrow :a\}\}$. It is immediate to see that the multiset Π_{self} is obtained by evaluating the base step of the closure $(:p \,\&\, :q)^*$ while the other mappings derive from the evaluation of $(:p \,\&\, :q)$ (step 1 of the closure). Moreover, no other mappings can be obtained by evaluating further closure steps. We claim that for every PP pattern $\pi_2=\langle ?b, elt, ?e\rangle$ the following property holds: either $[\![\pi_2]\!]_G=\varnothing$ or $[\![\pi_2]\!]_G \supset [\![\pi_1]\!]_G$. The proof of the theorem relies on the following claim.

Claim 25. Consider the graph G in Fig. 13. For every PP pattern $\langle ?b, elt, ?e\rangle$ we have that either $[\![\langle ?b, elt, ?e\rangle]\!]_G=\varnothing$ or $[\![\langle ?b, elt, ?e\rangle]\!]_G \sqsupseteq \Pi_{self}$.

Proof. We proceed by induction on the construction of the PP expression elt. We start with the base cases:

c1. If elt is of the form $elt = u \in \mathbf{I}$ then: (i) if $u=:p$ or $u=:q$ then $[\![\langle ?b, u, ?e\rangle]\!]_G \sqsupset \Pi_{self}$ because of the self-loops at each node ; (ii) otherwise $[\![\langle ?b, u, ?e\rangle]\!]_G = \varnothing$.

c2. If elt is $elt=!(u_1|...|u_n)$ or $elt=!(\hat{}u_1|...|\hat{}u_n)$ then: (i) if $:p \notin \{u_1, \ldots, u_n\}$ or $:q \notin \{u_1, \ldots, u_n\}$ then $[\![\langle ?b, elt, ?e\rangle]\!]_G \sqsupset \Pi_{self}$ because of the self-loops present at each node; (ii) otherwise $[\![\langle ?b, elt, ?e\rangle]\!]_G = \varnothing$.

c3. If elt is of the form $elt=!(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n)$ then $[\![\langle ?b, !(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n)?e\rangle]\!]_G= [\![\langle ?b, !(u_1|...|u_j), ?e\rangle]\!]_G \cup [\![\langle ?b, !(\hat{}u_{j+1}|...|\hat{}u_n), ?e\rangle]\!]_G$. Hence, the claim holds because of point c2 above.

Let elt_1, elt_2 be PP expressions; assume that it holds that either: (i) $[\![\langle ?b, elt_i, ?e\rangle]\!]_G=\varnothing$ or (ii) $[\![\langle ?b, elt_i, ?e\rangle]\!]_G \sqsupseteq \Pi_{self}$ for $i \in \{1, 2\}$. We now proceed with the inductive step and consider the other types of PP expressions.

c4. If elt is of the form $elt = elt_1 \mid elt_2$ then $[\![\langle ?b, elt_1 \mid elt_2, ?e\rangle]\!]_G=[\![\langle ?b, elt_1, ?e\rangle]\!]_G \cup [\![\langle ?b, elt_2, ?e\rangle]\!]_G$ and the claim follows from the properties of the algebra.

c5. If elt is of the form $elt = elt_1/elt_2$ then $[\![\langle ?b, elt_1/elt_2, ?e\rangle]\!]_G=[\![\langle ?b, elt_1, ?m\rangle]\!]_G \bowtie [\![\langle ?m, elt_2, ?e\rangle]\!]_G$ and the claim follows from the properties of the algebra.

c6. If elt is of the form $elt = (elt_1)^*$ then $[\![\langle ?b, (elt_1)^*, ?e\rangle]\!]_G \sqsupseteq \Pi_{self}$ as a consequence of the evaluation of the base step of the Kleene operator.

c7. If elt is of the form $elt = \hat{}(elt_1)$ then $[\![\langle ?b, \hat{}(elt_1), ?e\rangle]\!]_G=[\![\langle ?e, elt_1, ?b\rangle]\!]_G$ and the claim follows from the properties of the algebra.◁

By relying on Claim 25, the proof of the theorem continues by induction on the construction of the PP expression elt. We start with the base cases:

t1. If elt is of the form $elt=u \in \mathbf{I}$ then:

t1.1. if $elt=:p$ then $[\![\langle ?b, :p, ?e\rangle]\!]_G= \Pi_{self} \cup \{\{?b\rightarrow :a, ?e\rightarrow :c\}, \{?b\rightarrow :a, ?e\rightarrow :b\}, \{?b\rightarrow :b, ?e\rightarrow :a\}\}$. The answer is a superset of $[\![\pi_1]\!]_G$ because of the mapping $\{?b\rightarrow :a, ?e\rightarrow :c\}$.

t1.2. if $elt=:q$ then $[\![\langle ?b, :q, ?e\rangle]\!]_G= \Pi_{self} \cup \{\{?b\rightarrow :a, ?e\rightarrow :d\}, \{?b\rightarrow :a, ?e\rightarrow :b\}, \{?b\rightarrow :b, ?e\rightarrow :a\}\} \sqsupset [\![\pi_1]\!]_G$. The answer is a superset of $[\![\pi_1]\!]_G$ because of the mapping $\{?b\rightarrow :a, ?e\rightarrow :d\}$.

t1.3. otherwise, we have that $[\![\langle ?b, u, ?e\rangle]\!]_G = \varnothing$ for all $u \notin \{:p, :q\}$.

In what follows we will focus our attention on the extra answers only that makes $[\![elt]\!]_G$ a superset of $[\![\pi_1]\!]_G$.

t2. If elt is of the form $elt=!(u_1|...|u_n)$ then:

t2.1. if $:q \notin \{u_1, ..., u_n\}$ then $[\![\langle ?b, !(u_1|...|u_n), ?e\rangle]\!]_G \sqsupseteq [\![\langle ?b, :q, ?e\rangle]\!]_G$. Then the result follows from point t1.2.

t2.2. if $:p \notin \{u_1, ..., u_n\}$ then $[\![\langle ?b, !(u_1|...|u_n), ?e\rangle]\!]_G \sqsupseteq [\![\langle ?b, :p, ?e\rangle]\!]_G$. Then the result follows from point t1.1.

t2.3. if $:p, :q \in \{u_1, ..., u_n\}$ then $[\![\langle ?b, !(u_1|...|u_n), ?e\rangle]\!]_G=\varnothing$.

t3. If elt is of the form $elt=!(\hat{}u_1|...|\hat{}u_n)$ then:

t3.1. if $:q \notin \{u_1, ..., u_n\}$ then $[\![\langle ?b, !(\hat{}u_1|...|\hat{}u_n), ?e\rangle]\!]_G \sqsupseteq [\![\langle ?b, \hat{}:q, ?e\rangle]\!]_G = \Pi_{self} \cup \{\{?b\rightarrow :d, ?e\rightarrow :a\}, \{?b\rightarrow :a, ?e\rightarrow :b\}, \{?b\rightarrow :b, ?e\rightarrow :a\}\}$. The answer is a superset of $[\![\pi_1]\!]_G$ because of the mapping $\{?b\rightarrow :d, ?e\rightarrow :a\}$.

t3.2. if $:p \notin \{u_1, ..., u_n\}$ then $[\![\langle ?b, !(\hat{}u_1|...|\hat{}u_n), ?e\rangle]\!]_G \sqsupseteq [\![\langle ?b, \hat{}:p, ?e\rangle]\!]_G = \Pi_{self} \cup \{\{?b\rightarrow :c, ?e\rightarrow :a\}, \{?b\rightarrow :a, ?e\rightarrow :b\}, \{?b\rightarrow :b, ?e\rightarrow :a\}\}$. The answer is a superset of $[\![\pi_1]\!]_G$ because of the mapping $\{?b\rightarrow :c, ?e\rightarrow :a\}$.

t3.3. if $:p, :q \in \{u_1, ..., u_n\}$ then $[\![\langle ?b, !(\hat{}u_1|...|\hat{}u_n), ?e\rangle]\!]_G=\varnothing$.

t4. If elt is of the form $elt=!(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n)$ then:

t4.1. if $:\mathtt{p} \notin \{u_1|...|u_j\}$ then $[\![\langle ?\mathtt{b}, !(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n), ?\mathtt{e}\rangle]\!]_G \supseteq [\![\langle ?\mathtt{b}, :\mathtt{p}, ?\mathtt{e}\rangle]\!]_G$. Then the result follows from point t1.1.

t4.2. if $:\mathtt{p} \notin \{u_{j+1}|...|u_n\}$ then $[\![\langle ?\mathtt{b}, !(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n), ?\mathtt{e}\rangle]\!]_G \supseteq [\![\langle ?\mathtt{b}, \hat{}:\mathtt{p}, ?\mathtt{e}\rangle]\!]_G$. Then the result follows from point t3.2.

t4.3. if $:\mathtt{q} \notin \{u_1|...|u_j\}$ then $[\![\langle ?\mathtt{b}, !(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n), ?\mathtt{e}\rangle]\!]_G \supseteq [\![\langle ?\mathtt{b}, :\mathtt{q}, ?\mathtt{e}\rangle]\!]_G$. Then the result follows from point t1.2.

t4.4. if $:\mathtt{q} \notin \{u_{j+1}|...|u_n\}$ then $[\![\langle ?\mathtt{b}, !(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n), ?\mathtt{e}\rangle]\!]_G \supseteq [\![\langle ?\mathtt{b}, \hat{}:\mathtt{q}, ?\mathtt{e}\rangle]\!]_G$. Then the result follows from point t3.1.

t4.5. otherwise we have that $[\![\langle ?\mathtt{b}, !(u_1|...|u_j|\hat{}u_{j+1}|...|\hat{}u_n), ?\mathtt{e}\rangle]\!]_G = \varnothing$.

Let $\mathtt{elt}_1$ and $\mathtt{elt}_2$ be two PP expressions and assume that it holds that either: (i) $[\![\langle ?\mathtt{b}, \mathtt{elt}_i, ?\mathtt{e}\rangle]\!]_G = \varnothing$ or: (ii) $[\![\langle ?\mathtt{b}, \mathtt{elt}_i, ?\mathtt{e}\rangle]\!]_G \supseteq [\![\pi_1]\!]_G$ for $i \in \{1,2\}$. We now proceed with the inductive step and consider the other types of PP expressions.

t5. If $\mathtt{elt}$ is of the form $\mathtt{elt}=\mathtt{elt}_1 \mid \mathtt{elt}_2$ then $[\![\langle ?\mathtt{b}, (\mathtt{elt}_1|\mathtt{elt}_2), ?\mathtt{e}\rangle]\!]_G = [\![\langle ?\mathtt{b}, \mathtt{elt}_1, ?\mathtt{e}\rangle]\!]_G \cup [\![\langle ?\mathtt{b}, \mathtt{elt}_2, ?\mathtt{e}\rangle]\!]_G$. Hence, if either $\mathtt{elt}_1$ or $\mathtt{elt}_2$ are not empty, the presence of at least an additional answer follows from the properties of the algebra.

t6. If $\mathtt{elt}$ is of the form $\mathtt{elt}=\mathtt{elt}_1/\mathtt{elt}_2$ then $[\![\langle ?\mathtt{b}, (\mathtt{elt}_1/\mathtt{elt}_2), ?\mathtt{e}\rangle]\!]_G = [\![\langle ?\mathtt{b}, \mathtt{elt}_1, ?\mathtt{m}\rangle]\!]_G \bowtie [\![\langle ?\mathtt{m}, \mathtt{elt}_2, ?\mathtt{e}\rangle]\!]_G$. Hence, if both $\mathtt{elt}_1$ and $\mathtt{elt}_2$ are not empty, the presence of at least an additional answer follows from the properties of the algebra.

t7. If $\mathtt{elt}$ is of the form $\mathtt{elt}=(\mathtt{elt}_1)^*$ then $[\![\langle ?\mathtt{b}, (\mathtt{elt}_1)^*, ?\mathtt{e}\rangle]\!]_G$ the result follows from the monotonicity of transitive closure.

t8. If $\mathtt{elt}$ is of the form $\mathtt{elt}=\hat{}(\mathtt{elt}_1)$ then $[\![\langle ?\mathtt{b}, \hat{}(\mathtt{elt}_1), ?\mathtt{e}\rangle]\!]_G = [\![\langle ?\mathtt{e}, \mathtt{elt}_1, ?\mathtt{b}\rangle]\!]_G$.
If $[\![\langle ?\mathtt{e}, \mathtt{elt}_1, ?\mathtt{b}\rangle]\!]_G = \varnothing$ then also $[\![\langle ?\mathtt{b}, \hat{}\mathtt{elt}_1, ?\mathtt{e}\rangle]\!]_G = \varnothing$. Otherwise, by inductive hypothesis we have that:
(i) $[\![\langle ?\mathtt{e}, \mathtt{elt}_1, ?\mathtt{b}\rangle]\!]_G \supseteq \Pi_{self}$ and thus $[\![\langle ?\mathtt{b}, \hat{}(\mathtt{elt}_1), ?\mathtt{e}\rangle]\!]_G \supseteq \Pi_{self}$;
(ii) $\{\{?\mathtt{e}\to:\mathtt{a}, ?\mathtt{b}\to:\mathtt{b}\}, \{?\mathtt{e}\to:\mathtt{b}, ?\mathtt{b}\to:\mathtt{a}\}\} \sqsubset [\![\langle ?\mathtt{e}, \mathtt{elt}_1, ?\mathtt{b}\rangle]\!]_G$ and thus $\{\{?\mathtt{b}\to:\mathtt{a}, ?\mathtt{e}\to:\mathtt{b}\}, \{?\mathtt{b}\to:\mathtt{b}, ?\mathtt{e}\to:\mathtt{a}\}\} \sqsubset [\![\langle ?\mathtt{b}, \hat{}(\mathtt{elt}_1), ?\mathtt{e}\rangle]\!]_G$.
To conclude, note that there exists a mapping $\{?\mathtt{e}\to x, ?\mathtt{b}\to y\} \in [\![\langle ?\mathtt{e}, \mathtt{elt}_1, ?\mathtt{b}\rangle]\!]_G$ with $x, y \in \{:\mathtt{a}, :\mathtt{b}, :\mathtt{d}, :\mathtt{c}\}$, $x \neq y$, such that if $x{=}a$ implies $y{\neq}b$ and if $x{=}b$ implies $y{\neq}a$. Thus, the mapping $\{?\mathtt{b}\to y, ?\mathtt{e}\to x\} \in [\![\langle ?\mathtt{b}, \hat{}(\mathtt{elt}_1), ?\mathtt{e}\rangle]\!]_G$ and $\{?\mathtt{b}\to y, ?\mathtt{e}\to x\} \notin [\![\pi_1]\!]_G$.

□

To continue our expressiveness analysis, we now show that using EPPs as navigational core in SPARQL increases the expressive power of the language.

Theorem 26. There exists a $S^{\{\bowtie,\cup,\text{EPP}\}}$ query that cannot be expressed as a $S^{\{\bowtie,\cup,\text{PP}\}}$ query.

Proof. Consider the following $S^{\{\bowtie,\cup,\text{EPP},\}}$ query: Q_e=`SELECT ?b ?e WHERE {?b (:p &:q)* ?e.}` and the graph G in Figure 13. Let us indicate by $\pi = \langle ?\mathtt{b}, (:\mathtt{p}\ \&:\mathtt{q})^*, ?\mathtt{e}\rangle$ the EPP pattern in Q_e. By evaluating Q_e over G we obtain the set of mappings $\{\{?\mathtt{b}\to x, ?\mathtt{e}\to x \mid x \in \{:\mathtt{a}, :\mathtt{b}, :\mathtt{d}, :\mathtt{c}\}\}\} \cup \{\{?\mathtt{b}\to:\mathtt{a}, ?\mathtt{e}\to:\mathtt{b}\}, \{?\mathtt{b}\to:\mathtt{b}, ?\mathtt{e}\to:\mathtt{a}\}\}$ (in the following we will indicate as $\Pi_{self} = \{\{?\mathtt{b}\to x, ?\mathtt{e}\to x \mid x \in \{:\mathtt{a}, :\mathtt{b}, :\mathtt{c}, :\mathtt{d}\}\}$ the set of self-loop mappings). We will show that the query Q_e cannot be expressed by any $S^{\{\bowtie,\cup,\text{PP}\}}$ query $\bar{Q}$ of the form `SELECT ?b ?e WHERE`$\{\mathcal{P}\}$, where $\mathcal{P}$ is a pattern as defined in Section 2. We claim that for every pattern $\mathcal{P}$ (in the fragment $S^{\{\bowtie,\cup,\text{PP}\}}$) the following property holds: either $[\![\mathcal{P}]\!]_G = \varnothing$ or $[\![\mathcal{P}]\!]_G \supseteq [\![\pi]\!]_G$. Note that if $[\![\mathcal{P}]\!]_G \supseteq [\![\pi]\!]_G$ holds it also holds that $[\![\mathcal{P}]\!]_G \supseteq \Pi_{self}$. We prove the theorem by structural induction on the construction of the pattern $\mathcal{P}$ built by using the constructs in the fragment $S^{\{\bowtie,\cup,\text{PP}\}}$. We start with the base case:

Base: If $\mathcal{P} = \langle ?\mathtt{b}, \mathtt{elt}, ?\mathtt{e}\rangle$ is a single property path pattern then in virtue of Theorem 24 we have that either $[\![\langle ?\mathtt{b}, \mathtt{elt}, ?\mathtt{e}\rangle]\!]_G = \varnothing$ or $[\![\langle ?\mathtt{b}, \mathtt{elt}, ?\mathtt{e}\rangle]\!]_G \supseteq [\![\pi]\!]_G$.

Inductive step: Consider now the case of $\mathcal{P}$ containing two patterns $\mathcal{P}_1$ and $\mathcal{P}_2$ such that either $[\![\mathcal{P}_i]\!]_G = \varnothing$ or $[\![\mathcal{P}_i]\!]_G \supseteq [\![\pi]\!]_G \supseteq \Pi_{self}$ holds for $i \in \{1,2\}$.

- If $\mathcal{P} = \mathcal{P}_1$ AND $\mathcal{P}_2$ then $[\![\mathcal{P}_1 \text{ AND } \mathcal{P}_2]\!]_G = [\![\mathcal{P}_1]\!]_G \bowtie [\![\mathcal{P}_2]\!]_G$. If at least one of the two evaluations is empty then we can conclude $[\![\mathcal{P}_1 \text{ AND } \mathcal{P}_2]\!]_G = \varnothing$. Otherwise, if the evaluation of both $\mathcal{P}_1$ and $\mathcal{P}_2$ is not empty, than all the additional answers in $[\![\mathcal{P}_1]\!]_G$ and $[\![\mathcal{P}_2]\!]_G$ will not be discarded due to the fact that $[\![\mathcal{P}_1]\!]_G \supseteq \Pi_{self}$, $[\![\mathcal{P}_2]\!]_G \supseteq \Pi_{self}$ and from the properties of the algebra. Thus we can conclude that $[\![\mathcal{P}_1 \text{ AND } \mathcal{P}_2]\!]_G \supseteq [\![\pi]\!]_G$ holds.

Table 9
Languages and their translation into SPARQL for reasoning.

Navigational Core	Extended Processor	Reference in the Semantics	SPARQL Fragment
$p \in \mathbf{I}$	No	R1 in Fig. 5	$S^{\{\bowtie,\bigcup,\text{FILTER},\text{PP},\text{ALP1}\}}$
nPP	No	R1-R5 in Fig. 5	$S^{\{\bowtie,\bigcup,\text{FILTER},\text{PP},\text{ALP1}\}}$
nEPP	No	R1-R2, R5-R9, R11-R16 in Table 3	$S^{\{\bowtie,\bigcup,\text{FILTER},\text{PP},\text{ALP1}\}}$
PP	Yes	Fig. 5	$S^{\{\bowtie,\bigcup,\text{FILTER},\text{EPP},\text{EALP1}\}}$
EPP	Yes	Table 3	$S^{\{\bowtie,\bigcup,\text{FILTER},\text{EPP},\text{EALP1}\}}$

– If $\mathcal{P} = \mathcal{P}_1$ UNION $\mathcal{P}_2$ then $[\![\mathcal{P}_1 \text{ UNION } \mathcal{P}_2]\!]_G = [\![\mathcal{P}_1]\!]_G \cup [\![\mathcal{P}_2]\!]_G$. Hence, if the evaluation of both $\mathcal{P}_1$ or $\mathcal{P}_2$ is empty then we can conclude $[\![\mathcal{P}_1 \text{ AND } \mathcal{P}_2]\!]_G = \varnothing$. Otherwise, if the evaluation of either $\mathcal{P}_1$ or $\mathcal{P}_2$ is not empty, the presence of at least an additional answer follows from the properties of the algebra. □

6.2. *Expressiveness of SPARQL for Query-Based Reasoning*

We now study the expressiveness of SPARQL in terms of ρdf reasoning when considering various navigational cores. Table 9 mimics the expressiveness study in Table 6 where the second column describes the language produced to support query-based reasoning as described in Section 5. We can notice that, in general, *supporting reasoning requires a more expressive language in the rewriting*. For the basic case $p \in \mathbf{I}$, the query must be rewritten by applying rule R6 in Table 8; this requires the usage of EPP constructs such as nesting (TP), (conjunction of) tests (T), and closure ($\texttt{sp}^*$). Clearly, the presence of closure requires the usage of ALP1 that was not required in the case of plain RDF (first row in Table 6).

Interestingly, when considering more expressive forms of navigational patterns such as non-recursive property paths (nPP), and non-recursive EPPs (nEPP), the fragment needed to capture ρdf in the translation remains the same. The situation changes when moving to navigational patterns with recursion, that is, PP and EPP. In this case, the current SPARQL standard is not enough expressive to capture query-based ρdf reasoning. To give an intuition, consider the query $\texttt{?s}\ (\texttt{p}_1/\texttt{TP}(\texttt{\_p}, \texttt{p}_2))^*\ \texttt{?e}$ where $\texttt{p}_1, \texttt{p}_2 \in \mathbf{I}$ and $\texttt{?s}, \texttt{?e} \in \mathcal{V}$. It follows from Theorem 26 that this type of recursive queries cannot be evaluated under the simple RDF entailment regime via ALP1 although they can be evaluated via EALP1 defined for EPPs. If this negative result holds for plain RDF clearly it also holds for the ρdf entailment regime. The interesting point is that by observing Table 9 one can note that $S^{\{\bowtie,\bigcup,\text{FILTER},\text{EPP},\text{EALP1}\}}$ is the only close language with respect to ρdf reasoning for the rewriting shown in Table 8. We point out that our rewriting into SPARQL for plain RDF and ρdf require the same expressiveness. Practically speaking substituting the current ALP1 procedure with the EALP1 procedure for EPPs would allow full EPPs support for both the plain and ρdf entailment regime.

We want to mention that there could be other rewritings for reasoning using PPs. For instance, Bischof et al. [27] focus on a fragment of OWL. Authors point out how the support for `owl:symmertricProperty` is not possible by using PPs due to the limited expressiveness of this language. This limitation can be overcome by using EPPs (and EALP1) and other navigational extensions of SPARQL like NREs [16]. We again note that the crucial difference between EPPs and NREs is that the focus of the former is to give a receipt about how to extend the SPARQL standard, and how the standard currently support query-based reasoning, with a more powerful navigational core while NREs/nSPARQL depart from the standard (they have been defined before the introduction of PPs). We leave the study of how EPPs can be coupled with the approach proposed by Bischof et al. [27] as a future work.

7. iEPPs: a SPARQL-independent Language

The aim of this section is to study EPPs as an independent language. The advantage of defining EPPs as a navigational language independent from SPARQL stems from the fact that the SPARQL-based semantics and translation discussed in Section 3.2 and Section 4 only apply to KGs based on RDF while the proposed language can be used to query arbitrary KG. To this end, we give a set-based Semantics in Section 7.1 and present an evaluation algorithm along with a complexity analysis in Section 7.2.

Table 10
Set-based semantics for EPPs.

R1	$\mathbf{E}[\![\hat{}\,\mathtt{epp}]\!]_G$	:=	$\{(u,v) : (v,u) \in \mathbf{E}[\![\mathtt{epp}]\!]_G\}$
R2	$\mathbf{E}[\![\mathtt{epp}_1/\mathtt{epp}_2]\!]_G$	:=	$\{(u,v) : \exists w \text{ s.t. } (u,w) \in \mathbf{E}[\![\mathtt{epp}_1]\!]_G \wedge (w,v) \in \mathbf{E}[\![\mathtt{epp}_2]\!]_G\}$
R3	$\mathbf{E}[\![(\mathtt{epp})^*]\!]_G$	:=	$\{(u,u) \mid u \in \text{nodes}(G)\} \cup \bigcup_{i=1}^{\infty} \mathbf{E}[\![\mathtt{epp}_i]\!]_G \mid \mathtt{epp}_1 = \mathtt{epp} \wedge \mathtt{epp}_i = \mathtt{epp}_{i-1}/\mathtt{epp}$
R4	$\mathbf{E}[\![(\mathtt{epp})^+]\!]_G$	:=	$\bigcup_{i=1}^{\infty} \mathbf{E}[\![\mathtt{epp}_i]\!]_G \mid \mathtt{epp}_1 = \mathtt{epp} \wedge \mathtt{epp}_i = \mathtt{epp}_{i-1}/\mathtt{epp}$
R5	$\mathbf{E}[\![(\mathtt{epp})?]\!]_G$	:=	$\{(u,u) \mid u \in \text{nodes}(G)\} \cup \mathbf{E}[\![\mathtt{epp}]\!]_G$
R6	$\mathbf{E}[\![(\mathtt{epp}_1\|\mathtt{epp}_2)]\!]_G$	:=	$\{(u,v) : (u,v) \in \mathbf{E}[\![\mathtt{epp}_1]\!]_G \vee (u,v) \in \mathbf{E}[\![\mathtt{epp}_2]\!]_G\}$
R7	$\mathbf{E}[\![(\mathtt{epp}_1\&\mathtt{epp}_2)]\!]_G$	:=	$\{(u,v) : (u,v) \in \mathbf{E}[\![\mathtt{epp}_1]\!]_G \wedge (u,v) \in \mathbf{E}[\![\mathtt{epp}_2]\!]_G\}$
R8	$\mathbf{E}[\![(\mathtt{epp}_1 \sim \mathtt{epp}_2)]\!]_G$	:=	$\{(u,v) : (u,v) \in \mathbf{E}[\![\mathtt{epp}_1]\!]_G \wedge (u,v) \notin \mathbf{E}[\![\mathtt{epp}_2]\!]_G\}$
R9	$\mathbf{E}[\![\mathtt{epp}\{l,h\}]\!]_G$	:=	$\bigcup_{i=l}^{h} \mathbf{E}[\![\mathtt{epp}_i]\!]_G \mid \mathtt{epp}_1 = \mathtt{epp} \wedge \mathtt{epp}_i = \mathtt{epp}_{i-1}/\mathtt{epp}$
R10	$\mathbf{E}[\![\mathtt{POS}_1\ \mathtt{test}\ \mathtt{POS}_2]\!]_G$	:=	$\{(\Pi(\mathtt{POS}_1,\mathtt{t}),\Pi(\mathtt{POS}_2,\mathtt{t}))) \mid \mathtt{t} \in G \wedge \mathbf{E}_T[\![\mathtt{test}]\!]^{\mathtt{t}}_G\}$
R11	$\mathbf{E}_T[\![\mathtt{u}]\!]^{\mathtt{t}}_G$	:=	`true` if $\Pi(\mathtt{\_p},\mathtt{t}) = \mathtt{u}$, `false` otherwise
R12	$\mathbf{E}_T[\![\mathtt{T}(\mathtt{EExp})]\!]^{\mathtt{t}}_G$	:=	`Evaluate(EExp,t)`
R13	$\mathbf{E}_T[\![\mathtt{TP}(\mathtt{POS},\mathtt{epp})\ ]\!]^{\mathtt{t}}_G$	:=	`true` if $\exists\, v : (\Pi(\mathtt{POS},\mathtt{t}),v) \in \mathbf{E}[\![\mathtt{epp}]\!]_G$, `false` otherwise
R14	$\mathbf{E}_T[\![\mathtt{test}_1\&\&\mathtt{test}_2\ ]\!]^{\mathtt{t}}_G$	:=	$\mathbf{E}_T[\![\mathtt{test}_1\ ]\!]^{\mathtt{t}}_G \wedge \mathbf{E}_T[\![\mathtt{test}_2\ ]\!]^{\mathtt{t}}_G$
R15	$\mathbf{E}_T[\![\mathtt{test}_1\|\|\mathtt{test}_2\ ]\!]^{\mathtt{t}}_G$	:=	$\mathbf{E}_T[\![\mathtt{test}_1\ ]\!]^{\mathtt{t}}_G \vee \mathbf{E}_T[\![\mathtt{test}_2\ ]\!]^{\mathtt{t}}_G$
R16	$\mathbf{E}_T[\![!\mathtt{test}\ ]\!]^{\mathtt{t}}_G$	:=	$\neg\mathbf{E}_T[\![\mathtt{test}\ ]\!]^{\mathtt{t}}_G$

7.1. Formal Semantics of EPPs based on sets

The semantics of EPPs based on sets for both recursive and non-recursive EPPs is shown in Table 10. It leverages two evaluation functions. The first, $\mathbf{E}[\![\mathtt{epp}]\!]_G$ given an epp expression and a graph G returns the pairs of nodes that are linked by paths conforming to epp. The second $\mathbf{E}_T[\![\mathtt{test}]\!]^{\mathtt{t}}_G$, given a test test, a graph G and a triple $\mathtt{t} \in G$, returns `true` if the triple satisfies the test and `false` otherwise. The semantics follows the same spirit of other navigational languages like NREs [16] although EPPs offer more features (e.g., path conjunction and path difference).

7.2. Evaluation Algorithm

The aim of this section is to study whether the semantics in Table 10 can be implemented in an efficient way. In what follows we show an efficient evaluation algorithm, that has been implemented in a custom query evaluator, and discuss its complexity. The presented evaluation algorithm for iEPPs expressions is similar to those of other navigational languages such as nested regular expressions [16] and NautiLOD [14]. The algorithm starts by invoking EVALUATE, which receives as input a graph G, an expression epp and a node n. If epp is non recursive (i.e., it does not contain the closure operators '+' and '*') then it is given as input to the function BASE, which considers the various forms of syntactic expressions. For recursive expressions the algorithm uses the function CLOSURE. Finally, the boolean function EVALTEST handles the different types of test.

Function EVALUATE$(n, \mathtt{epp}, G)$
Input: node n, expression epp, graph G; **Output:** node set Res.

if $\mathtt{epp} = (\mathtt{epp}_1)^*$ **then**
  return CLOSURE$(n, \mathtt{epp}_1, G, \{\}, 0, *)$
else if $\mathtt{epp} = (\mathtt{epp}_1)^+$ **then**
  return CLOSURE$(n, \mathtt{epp}_1, G, \{\}, 1, *)$
else if $\mathtt{epp} = (\mathtt{epp}_1)\{l,h\}$ **then**
  return CLOSURE$(n, \mathtt{epp}_1, G, \{\}, l, h)$
else
  return BASE$(n, \mathtt{epp}, G)$)

Function CLOSURE$(n, \mathtt{epp}, G, Res, l, h)$
Input: node n, EPPs expression epp, graph G, node set Res, lower bound l, upper bound h; **Output:** node set Res.

$S = \{n\}$
for all $i \in \{1, ..., l\}$ **do**
  $S' = \bigcup_{n \in S}$ EVALUATE$(n, \mathtt{epp}, G)$
  $S = S'$
$i = l + 1$
while $S \neq \varnothing$ AND $(h = * \text{ OR } i <= h)$ **do**
  $S' = \varnothing$
  while $S \neq \varnothing$ **do**
    $n = extractNode(S)$ /* delete the node n from S */
    if $n \notin Res$ **then**
      $Res = Res \cup \{n\}$
      $S' = S' \cup$ EVALUATE$(n, \mathtt{epp}, G)$
  $i = i + 1$
  $S = S'$
return Res

Function BASE$(n, \texttt{epp}, G)$
Input: node n, EPPs expression epp, graph G; **Output:** node set Res.
1: **if** epp $=$ ^epp$_1$ **then**
2: **return** EVALUATE$(n, reverse(\texttt{epp}_1), G)$
3: **if** epp $=$ epp$_1$|epp$_2$ **then**
4: **return** EVALUATE$(n, \texttt{epp}_1, G) \cup$ EVALUATE$(n, \texttt{epp}_2, G)$
5: **if** epp $=$ epp$_1$/epp$_2$ **then**
6: $Res' :=$ EVALUATE$(n, \texttt{epp}_1, G)$
7: $Res = \varnothing$
8: **for all** nodes $n' \in Res'$ **do**
9: $Res = Res \cup$ EVALUATE$(n', \texttt{epp}_2, G)$
10: **return** Res
11: **if** epp $=$ epp$_1$&epp$_2$ **then**
12: **return** EVALUATE$(n, \texttt{epp}_1, G) \cap$ EVALUATE$(n, \texttt{epp}_2, G)$
13: **if** epp $=$ epp$_1 \sim$ epp$_2$ **then**
14: **return** EVALUATE$(n, \texttt{epp}_1, G) \setminus$ EVALUATE$(n, \texttt{epp}_2, G)$
15: **if** epp $=$ epp$_1$? **then**
16: **return** $\{n\} \cup$ EVALUATE$(n, \texttt{epp}_1, G)$
17: **if** epp $=$ POS$_1$ test POS$_2$ **then**
18: $Res = \varnothing$
19: **for all** triple $t \in G$ **do**
20: **if** EVALTEST$(n, \text{POS}_1, \text{POS}_2, t, \texttt{test}, G)$ **then**
21: Res=$Res \cup \{\Pi(\text{POS}_1, t), \Pi(\text{POS}_2, t)\}$ /* $\Pi(\text{POS}_1, t) = n$*/
22: **return** Res

Function EVALTEST$(n, \text{POS}_1, \text{POS}_2, t, \texttt{test}, G)$
Input: node n, position POS$_1$, position POS$_2$, triple t, test, test,graph G;
Output: true if t satisfy test.
1: **if** test $=$ test$_1$&&test$_2$ **then**
2: **return** EVALTEST$(n, \text{POS}_1, \text{POS}_2, t, \texttt{test}_1, G) \wedge$
$\wedge$EVALTEST$(n, \text{POS}_1, \text{POS}_2, t, \texttt{test}_2, G)$
3: **if** test $=$ test$_1$||test$_2$ **then**
4: **return** EVALTEST$(n, \text{POS}_1, \text{POS}_2, t, \texttt{test}_1, G) \vee$
$\vee$EVALTEST$(n, \text{POS}_1, \text{POS}_2, t, \texttt{test}_2, G)$
5: **if** test $=$!test$_1$ **then**
6: **return** $\neg$EVALTEST$(n, \text{POS}_1, \text{POS}_2, t, \texttt{test}_1, G)$
7: **if** test $=$ u **then**
8: **return** $\Pi(\text{POS}_1, t) = n \wedge \Pi(\_\text{p}, t) = \texttt{u}$
9: **if** test $=$ TP(POS, epp) **then**
10: **return** $\Pi(\text{POS}_1, t) = n \wedge$ EVALUATE$(\Pi(\text{POS}, t), \texttt{epp}, G) \neq \varnothing$
11: **if** test $=$ T(EExp) **then**
12: **return** $\Pi(\text{POS}_1, t) = n \wedge$ EvalSPARQLBuilt-in(EExp, t)

The result of the evaluation of an iEPP expression epp from a node n is a set of nodes n_r where nodes n_r are reachable from n via paths satisfying epp. To study the complexity of the evaluation algorithm we introduce the decision problem EVAL**EPPs**, which takes as input an EPP expression e, a pair of nodes (s, r) and a graph G and asks whether $(s, r) \in [\![e]\!]_G$.

Theorem 27. The EVAL**EPPs** problem can be solved in time $O(|G| \cdot |\texttt{epp}|) + c_{\texttt{EExp}}$, where $c_{\texttt{EExp}}$ is the cost of evaluating built-in conditions.

Proof. We assume G to be stored by its adjacency list. In particular, for each q $\in$ terms(G), a Hashtable is maintained where the set of keys is the set of predicates p such that there exists a triple in G having as subject q and as predicate p, and the set of values are lists of objects o reachable by traversing p-predicates from q. We assume that given q and a predicate p the set of nodes reachable can be accessed in time $O(1)$. An additional Hashtable is used for inverse navigation, that is, for navigation starting on the object and ending on the subject. Both structures use space $O(|G|)$. Let $|\texttt{epp}|$ be the size of the iEPP expression epp.

The function EVALUATE is recursively called on each sub-expression of the epp in input; if such sub-expressions are not recursive (i.e., do not contain '*', '+'), EVALUATE is invoked at most $O(|\texttt{epp}|)$ times. The base cases (lines 17-21 of function BASE) require to consider at most all the edges for all the nodes; this can be done in time $O(|G|)$. If epp is recursive, the function CLOSURE is executed at most $O(nodes(G))$ times; the procedure EVALUATE is invoked for each node in the worst case. When evaluating a subexpression from a node we use *memoization* to store its result (i.e., the set of reachable nodes) thus avoiding to recompute the same expression from the same node multiple times. Memoization guarantees that the total time required by CLOSURE is $O(|\texttt{epp}| \cdot |G|)$. As for *nested* expressions, memoization enables to mark nodes of the graph satisfying a given subexpression. Path conjunction and difference, corresponding to intersection and difference of sets of nodes respectively (line 12 and 14 of BASE), can be computed in time $O(|G|)$ by using a (prefect) hash function as the graph is known beforehand. As for tests, their cost is constant for *logical operators* and simple URI checking. The complexity is parametric wrt the cost of other SPARQL-based built-in conditions EExp ($c_{\texttt{EExp}}$). Finally, observe that with memoization the space complexity is $O(|\texttt{epp}| \cdot nodes(G)^2)$. □

8. Experimental Evaluation

This section reports on an experimental evaluation meant to investigate different aspects of the EPPs language discussed in the previous sections. Section 8.1 investigates the overhead of our translation algorithm, presented in Section 4, and compares it with that of translations routinely performed by existing SPARQL processors. Then, in Section 8.2 we discuss performance in terms of query evaluation time. We compare the running time of a custom processor implementing the evaluation algorithm for iEPPs, presented in Section 7, with the running time of a SPARQL processor. Finally, in Section 8.3 we discuss the impact of using query-based reasoning, presented in Section 5, both in terms of running time and number of results.

All the experiments have been performed on an Intel i5 machine with 8GBs RAM. Results are the average

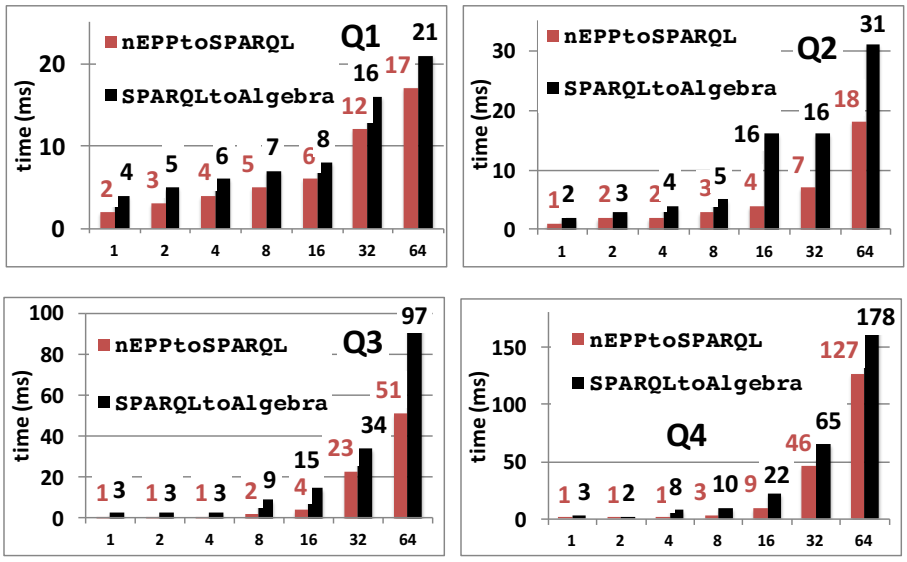


Fig. 14. Time (ms) of the translation overhead of Jena ARQ (`SPARQLtoAlgebra`) and nEPPs (`nEPPtoSPARQL`) vs number of path steps.

of 5 runs (queries were ran in a random order each time) after the top and bottom outliers are removed.

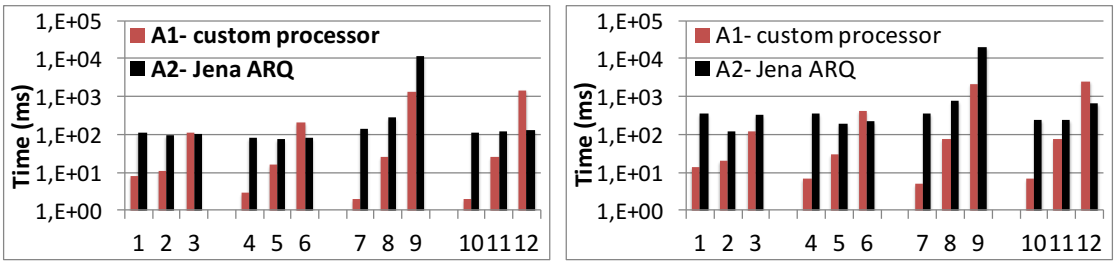
8.1. Translation Performance

Our primary objective is to make practical the immediate adoption of EPPs as a query language for KGs. This objective is fulfilled by using our translation from nEPPs to SPARQL as front end to any existing SPARQL processor. To investigate the performance of the translation algorithm presented in Section 4, we show that our nEPPs to SPARQL translation performs comparably to the existing translations routinely performed by SPARQL processors.

We compared our translation algorithm with the *SPARQL syntax to SPARQL algebra* (referred to as `SPARQLtoAlgebra`) translation performed by ARQ[10]. We used 28 queries generated in two steps. We started with three base expressions (Q1-Q3) plus a fourth one combining them (Q4). Q4 includes all the nEPPs constructs; concatenation, path conjunction, path difference, path test, and logical tests with all the logical operators. Second, we generate increasingly longer expressions Q_i^k by concatenating $Q_i^{(k-1)}/Q_i^{(k-1)}$, up to k=6. The resulting Q_i^6 fragments involve the concatenation of 64 path steps. The running times of the `nEPPtoSPARQL` and `SPARQLtoAlgebra` translations, for each query, are shown in Figure 14. Our translation performs similarly (slightly faster) than ARQ's existing initial phase, and this behavior shows a consistent trend in two dimensions (Q_i^k expressions use more EPP constructs for increasing i, and become exponentially longer for increasing k). To give a sense of the length of the expressions, we observe that Q_4^6 is a 19K characters long nEPPs expression (with an operational tree containing over one thousand nodes), while the Q_4^6 SPARQL translation is 133K characters long after filter elimination (the original translation is ∼239K characters).

While this suggests that the cost of our approach could be up to twice the cost of a direct nEPPs to algebra translation, keep in mind that we are comparing initial phases of query processing and these are typically much faster than subsequent phases. As an example, in Jena ARQ the `SPARQLtoAlgebra` translation is followed by an algebra to algebra optimization phase [28]. The remaining pre-processing phases (particularly those using dataset statistics) can be far more expensive than this initial phase. To give another example, if we consider Virtuoso, we observe that the initial SPARQL to SQL translation phase is followed by a more expensive cost based SQL optimization phase. Hence, the impact of our translation on the running time is negligible as compared to the total running time and other kinds of translations routinely performed by SPARQL processors.

[10]http://jena.apache.org/documentation/query/algebra.html



(a) Simple Entailment on FOAF for epp and $\mathcal{S}_{\texttt{epp}}$ queries.

(b) ρdf Entailment on FOAF for $\texttt{epp}^{\rho}$ and $\mathcal{S}^{\rho}_{\texttt{epp}}$ queries

Fig. 15. Query time for simple and ρdf-entailment comparing iEPPs and Jena ARQ.

8.2. Custom Processor vs. SPARQL Translation

We now discuss the performance, in terms of running time, of the custom EPPs processor implementing the iEPPs evaluation algorithm discussed in Section 7.2 against the translation-based approach described in Section 4. This experiment gives insights about the pros and cons of evaluating EPPs into existing SPARQL processors as compared to the usage of a custom query processor.

We considered a portion of FOAF obtained from the BTC2012 dataset[11] by traversing from the URI of T. Berners-Lee (TBL) `foaf:knows` predicates up to distance 4. This dataset including ∼4M triples has been loaded in memory by using Jena ARQ. We created 4 groups $Q_i, i \in \{1,...,4\}$ of similar nEPP expressions each with 3 queries; this gives a queryset $Q = \bigcup_{i=1}^{4} Q_i$ of a total of 12 queries. For each $\texttt{epp} \in Q$ we generated the corresponding SPARQL query $\mathcal{S}_{\texttt{epp}}$ via the translation algorithm. To investigate the performance also when including the query-based reasoning capabilities discussed in Section 5, we translated each epp into another query $\texttt{epp}^{\rho}$ and each $\mathcal{S}_{\texttt{epp}}$ into another query $\mathcal{S}^{\rho}_{\texttt{epp}}$. At this point, the original query epp and its reasoning-aware variant $\texttt{epp}^{\rho}$ are evaluated via the custom processor while the translated $\mathcal{S}_{\texttt{epp}}$ query and its reasoning-aware variant $\mathcal{S}^{\rho}_{\texttt{epp}}$ via Jena ARQ.

Fig. 15(a) shows the comparison when executing the queries without considering reasoning capabilities (i.e., under *simple entailment*). Fig. 15(b) shows results when considering the ρdf *entailment regime*.

For Q_1, which contains queries asking for friends of TBL at distance 1, 2 and 3, the custom processor performs better than Jena at distance 1 and 2; at distance 3 times are comparable. Q_2 additionally considers a test based on *nesting*. Again, the custom processor performs better at distance 1 and 2; at distance 3 it shows a higher running time. In Q_3, which considers *path difference* (e.g., *exclusive friends* at various distances) the custom processor performs consistently better. Finally, in Q_4 that includes conjunction (to ask for *mutual* friends at various distances) the custom processor performs better at distance 1 and 2 and obtains a higher running time at distance 3. These experiments suggest that for real-world data and natural queries (e.g., mutual friends) working with SPARQL-translated nEPPs and using existing processors (Jena in this case) is a bit less efficient than using the custom query processor. Note that the custom processor works in memory similarly to nSPARQL and other SPARQL navigational extensions. This clearly limits the applicability of these approaches on real-world graphs that typically do not fit into main memory and underlines the advantage to adopt our rewriting approach into SPARQL queries that can be evaluated on existing SPARQL processors capable of handling large graphs.

The huge advantage of using nEPPs is that navigational queries can be written in a succinct way. Anecdotally, while the nEPPs asking for mutual friends (simple entailment) at distance 3 contains ∼200 characters, the SPARQL query (obtained from the translation) contains ∼700 characters; moreover, writing navigational queries directly in SPARQL requires to deal with a large number of variables that need to be consistently joined. The situation is even worse when considering path repetitions that can be easily captured by the nEPPs syntax but require the union of several queries when using SPARQL. The number of results ranges from ∼50 to ∼8000 for the simple entailment and from ∼150 to ∼14500 for the ρdf entailment, respectively.

8.3. Query-Based Reasoning

We now move to a larger scale evaluation of the query-based reasoning approach described in Sec-

[11] http://km.aifb.kit.edu/projects/btc-2012

tion 5. The overall goal is to investigate the overhead of executing queries via the query-based reasoning approach as compared to plain RDF. Moreover, we also investigate the number of results returned.

Among the ρdf inference rules (see Table 5) we considered the two most interesting, that is, R5 that allows to derive new `rdf:type` information and R6 that allows the derivation of generic (sub)properties. Deriving new `rdf:type` information is particularly useful in efficient query processing via type-aware graph transformations [29]. The other rules in Table 5 either derive schema information (e.g., R3-R4) or can be captured via PPs (e.g., R1). For simple RDF, each query was executed as it is. Under the ρdf entailment, each query was first rewritten as described in Section 5.2. The prototypical EPP expression has the form `entity prop ?y`, where `prop`$\in$ {`rdf:type`, `dbo:genre`, `dbo:location`, `yago:hasLocation`}. For instance, the EPP `dbp:Tracy_Mann rdf:type ?y` retrieves asserted RDF types for the entity Tracy Mann. When rewriting this query we could also get inferred RDF types. We tested the performance of the query-based reasoning approach featured by EPPs on existing SPARQL processors (both local and remote) as shown in Table 11.

Table 11
Datasets used for the evaluation of query-based reasoning.

Dataset	Triples	Availability
LinkedMDB [12]	6M	local SPARQL endpoint
Yago[13]	400M	local SPARQL endpoint
DBpedia	412M	remote SPARQL endpoint[14]
LDCache	22B	remote SPARQL endpoint[15]

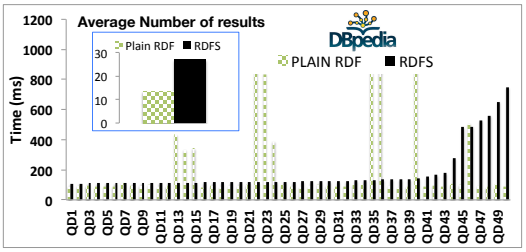
DBpedia is a large dataset with limited RDFS usage, Yago/LDCache makes extensive usage of RDFS predicates while LinkedMDB does not use RDFS. LinkedMDB and Yago have been loaded into a BlazeGraph[16] instance while DBpedia and LDCache have been accessed via their Virtuoso[17] SPARQL endpoints. Figs. 16 (a)-(c) report the running times on the RDFS rule R5 on 50 different queries that count the number of results by randomly picking 50 entities in DBpedia, Yago and LinkedMDB, respectively. Results are available in Appendix A. We observe that the overhead of the translation is reasonable and there are a few exceptions (in DBpedia) where plain RDF query execution takes more time. As expected, there is some variation in DBpedia while the overhead is larger in Yago. Note that query answering under entailment regime in some cases, takes less time; this can be explained by the fact that it requires the usage of the `ALP1` procedure that may perform better than the standard evaluation technique in some cases. To show that even without additional inference the overhead is minimal, we tested R5 also on LinkedMDB (that does not have schema).

The advantage of using the entailment regime is evident when looking at the average number of results, reported in the inner boxes in Figs. 16 (a)-(c). As an example, on DBpedia it increases from 13 to 27. The average ratio in terms of time for all queries is 1.7, 11.3, and 1.7 for DBpedia, Yago and LinkedMDB, respectively. As expected, the larger the ratio the larger the number of results. Figs. 16 (d)-(f) further investigate the benefit of query-based reasoning.

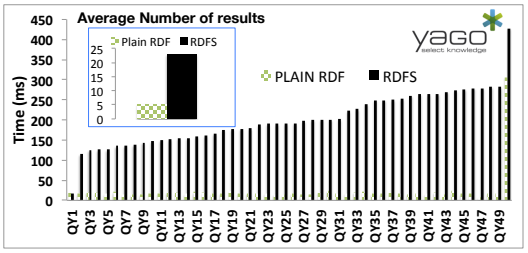
We created 150 additional queries for R6; 100 for DBpedia by considering two properties, that is, `dbpo:genre` and `dbpo:location`, and 50 for LDCache by picking the property `yago:hasLocation` (note that we used a property from Yago's schema since it is contained in LDCache). By looking at R5 in Table 8 it can be noted that the translation of an EPP under the entailment regime requires the union of three queries; hence, the resulting EPP is translated into a SPARQL query using (three) `UNION`. On the other hand, the translation of an EPP to capture R6 requires a single query that will be translated in SPARQL using `FILTER` (to capture tests).

In other words, queries using R5 are more involved than those using R6. R6 has less overhead than R5; this also reflects on the average ratio of time that now is 1.15 for `dbpo:genre`, 1.44 for `dbpo:location` and 1.05 for `yago:hasLocation`. By looking at the average number of results (reported in the inner boxes) it can be observed that plain RDF did not provide any result while our query-based reasoning approach allowed to get results. To be more specific, in DBpedia results have been obtained not via the property `dbpo:genre`, but via the more general property `dbpo:literaryGenre`. This allowed to discover, for instance, that `Night Surf` (one seed entity) is a post-apocalyptic short story. In LDCache, while the query returned zero results when using `yago:hasLocation`, it returned results via the more general property `yago:placedIn` (via R6).

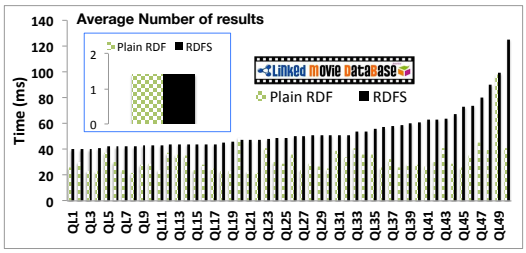
[12] http://linkedmdb.org
[13] www.mpi-inf.mpg.de/yago
[14] http://dbpedia.org/snorql
[15] http://lod.openlinksw.com/sparql
[16] https://www.blazegraph.com/download
[17] http://virtuoso.openlinksw.com



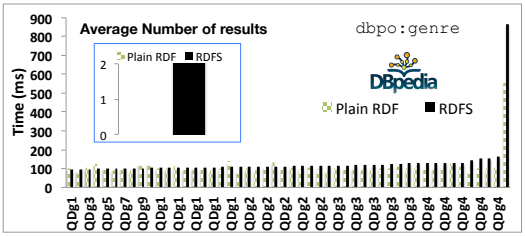
(a) DBPedia (remote endpoint) results for R5 in Table 8



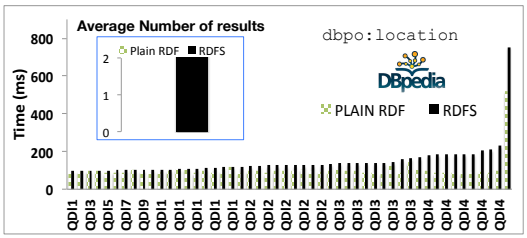
(b) Yago (local endpoint) results for R5 in Table 8



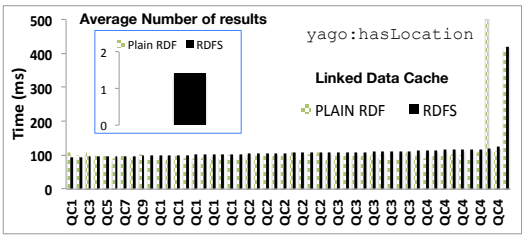
(c) LinkedMDB (local endpoint) results for R5 in Table 8



(d) DBPedia (remote endpoint) results for R6 (`dbo:genre`).



(e) DBpedia (remote endpoint) results for R6 (`dbo:location`).



(f) LDCache (remote endpoint) results for R6.

Fig. 16. Query time for simple and ρdf entailment over different datasets. Average number of results reported in the inner boxes.

Comparison with Closure Computation. An additional advantage of the query-based approach is that it can benefit from space optimization if one would work with the *transitive reduction*[18] of a graph [30] that removes edges derivable from ρdf-reasoning. In contrast, if one wants to precompute the closure (the currently used approach) one would need to materialize the full closure of the RDF graph under consideration, which would require cubic space in the worst case [25]. This become prohibitive for large KGs like DBPedia, Yago and many other. Indeed, we did measure in a local copy of (a subset of) Yago the space and the time of the closure. Starting from 400M triples the closure doubled the number of triples (giving 853M triples) and took 3.5h of computation.

[18]Tools like Slib[19] can compute the reduction of RDF graphs.

9. Related Work

The idea of graph query languages is to use (variants of) regular expressions to express (in a compact way) navigational patterns (e.g., [13, 31–33]). Angles and Gutierrez [34], and Wood [35] provide surveys on the topic while Barceló provides a detailed overview of research in the field [36] while Angles et al. [7] describe a recent proposal. Our goal with EPPs is to *extend* the navigational core of SPARQL (i.e., PPs) and make the extension readily available into existing SPARQL processors.

9.1. SPARQL Navigational Extensions

Proposals to extend SPARQL with navigational features have been around from some time. Notable examples are PSPARQL [21] and nSPARQL [16] that

tackled this problem even before the standardization of property paths (PPs) as SPARQL navigational core. From the practical point of view, the need for RDF navigational languages is witnessed by projects like Apache Marmotta[20] that incorporates a simple navigational language that borrows ideas from XPath. Since our main goal is to extend the navigational core of SPARQL we focus on the comparison between EPPs and other SPARQL navigational extensions. We compare EPPs with PPs, cpSPARQL [21], rec-SPARQL [15], RDFPath [37], nSPARQL-NREs [16], and star-free Nested Regular Expressions (sfNREs) that extends NREs with negation [38]. Table 12 summarizes the results of the comparison; we considered the following language features: path conjunction (&), path difference ($\sim$), negation of tests (!), nesting (TP), tests over nodes (T), usage of positions (POS), path repetitions ({l,h}), entailment regime, and closure operator (*). Additionally, we consider how expressions in each of the languages are evaluated, the support for reasoning (we focus on RDFS and in particular the ρdf fragment [19]) and the support for query-based reasoning (QBR); finally, we also report whether the language is implemented.

RDFPath is more focused on specific types of queries (i.e., shortest paths) and their efficient implementation in MapReduce and it is the language having less features than all the other languages considered. Path conjunction/difference are *natively* supported only by EPPs and sfNREs while nSPARQL, cpSPARQL and rec-SPARQL require the usage of the SPARQL algebra (i.e., . for conjunction). Nevertheless, this does not allow to use path conjunction inside the closure operator where the number path conjunction evaluations is apriori not bound. As a side note, we also mention that queries that resort to the SPARQL algebra for conjunction are more verbose. Finally, nSPARQL, cpSPARQL and rec-SPARQL do not support path difference. Test negation (!) is only supported by PPs (e.g., via negated property sets) and EPPs; *nesting* is supported by all languages but PPs and rec-SPARQL. However, only EPPs allow to test *node values* in a nested expression (see Example 6). Node tests are supported in limited form by cpSPARQL; EPPs allow logical combination of tests representing nesting and tests representing (in)equalities of node values. As a matter of fact, none of these extensions can express the *Italian exclusive friends* query mentioned in the Introduction. EPPs support path repetitions; this feature (called curly brace form) is in the agenda of the SPARQL working group[21]. rec-SPARQL also supports repetitions of more verbose queries since the motivation behind rec-SPARQL is not to provide a concise syntax. Nevertheless, rec-SPARQL requires an ad-hoc query processor.

A crucial difference between EPPs and related research is that we tackle the problem of extending the SPARQL language in the least intrusive way. We show that there exists a precise fragment of SPARQL that is expressive enough to capture non recursive EPPs (nEPPs), that is, EPPs that do not use closure operators (i.e., * and $^+$). Therefore, following the same line of the SPARQL standard where non-recursive PPs are translated into SPARQL queries, we devised a translation from (concise) nEPPs into (more verbose) SPARQL queries. The advantage of this approach wrt previous navigational extensions of SPARQL (e.g., [16, 21, 39]) that require the usage of ad-hoc query processors is that nEPPs can be evaluated on existing SPARQL processors.

Reasoning is *not supported* by PPs, sfNREs, RDFPath, and rec-SPARQL. Along the same line of NREs (and nSPARQL) and cpSPARQL, we focus on how EPPs can support SPARQL queries with embedded reasoning capabilities [28]. We focus on the ρdf fragment [19], which captures the main semantic functionalities of RDFS. We show that certain classes of SPARQL queries can be rewritten into queries that capture ρdf semantic functionalities, and thus can be evaluated on existing SPARQL processors. This is again a significant advantage as compared to previous attempts (e.g., nSPARQL [16]) that require ad-hoc processors.

Another difference with related proposals concerns the implementation of the language. To foster the adoption of EPPs and show its feasibility, we make EPPs available to users and developers in different forms: (i) as an implementation independent from SPARQL; (ii) as a front-end to SPARQL endpoints (for nEPPs) and (iii) as an extension to the Jena library. Further information along with pointers to the source code is available on the EPPs's website[22].

Finally, our study includes two novel expressiveness aspects. The first concerns the expressive power of the current SPARQL standard in terms of naviga-

[20] http://marmotta.apache.org

[21] http://www.w3.org/2009/sparql/wiki/Future_Work_Items

[22] http://extendedpps.wordpress.com

Table 12
Comparison of EPPs with other navigational extensions of SPARQL.

Lang	Features (Native Support)								Eval	Reasoning	QBR	Impl
	&	∼	TP	!	T	POS	{l,h}	*				
EPPs	X	X	X	X	X	X	X	X	SPARQL + EALP1	X	X	X
PPs				limited				limited	SPARQL +ALP1			X
cpSPARQL					X	X			Ad-hoc	X		X
rec-SPARQL									Ad-hoc			X
RDFPath								limited	Ad-hoc			X
NREs			X			X			Ad-hoc	X		
sfNREs	X	X	X	X		X			SPARQL			

tional features (see Section 6). We show that the language of EPPs is more expressive than SPARQL PPs; as a by-product we show that using EPPs as navigational core in SPARQL increases the expressive power of the whole SPARQL language. The second aspect concerns the expressiveness of SPARQL also in terms of query-based reasoning capabilities when considering the ρdf entailment regime (see Section 5). We show that our translation allows to evaluate queries enhanced with reasoning capabilities on existing SPARQL processors. We also show that EPPs is the only closed language in this respect and that in general, rewriting a query to capture the entailment regime requires a more expressive language in the rewriting.

9.2. Other Navigational Languages

Besides SPARQL navigational extensions there exist other graph languages like GraphQL [40] the Facebook query language. However, this language departs from the SPARQL standard and it is not clear how reasoning is supported. We also mention logic-based languages like TriAL [41], TriQ [42], GxPAth [43], and NEMODEQ [44]. Since these languages depart from the SPARQL standard, query evaluation cannot be done on existing SPARQL processors. On the contrary, our primary focus is on *extending* the current navigational core of the SPARQL standard by keeping compatibility and allowing query evaluation on existing SPARQL processors also under the ρdf entailment regime. Indeed, none of the above proposals has focused on the expressiveness of the current SPARQL standard in terms of navigational features. Ditto for the support of the ρdf entailment regime on *existing SPARQL processors*. We also mention work on graphs with data (e.g., [43]). This line of research: (i) does not adopt the RDF standard data model; (ii) does not consider SPARQL, which is the focus of this paper; (iii) does not deal with entailment regimes. Our work is also related to: *(i)* Ontology Based Data Access [26], where a (conjunctive) query is rewritten into a (set of) queries that fully incorporate the schema information. In this case the schema is treated separately and is needed in the rewriting; *(ii)* approaches that rewrite queries to capture entailment regimes like Bischof et al. [27]; *(iii)* approaches independent from SPARQL such as Stefanoni et al. [45] that study conjunctive and navigational queries over OWL 2 EL. Another recent line of research studied the problem of introducing recursion into SPARQL [15]. Our approach has different objectives. We focus on EPPs, a more expressive language than PPs; we provide a precise account of those fragments that can be executed on existing SPARQL processors and those that cannot, with or without considering the (ρdf) entailment regime. Hence, our study is more focused on expressiveness wrt SPARQL. Moreover, our approach is readily available and has been experimentally evaluated. The comparison with navigational languages for the Web of data (e.g., [14, 46–49]) is orthogonal to our goal. We also want to mention recent research that studied problems related to SPARQL property paths, including containment and subsumption [50]. We performed a similar study for EPPs. Results range from undecidability for the full EPPs to 2-EXPTIME for the positive queries [51].

9.3. CONSTRUCT Query Forms

Reutters et. al [15] proposed to enhance the expressive power of SPARQL via the introduction of recursions in a similar way to SQL. The idea is to alternate CONSTRUCT queries (that materialize in a graph the portion of data needed in each recursive call) and SELECT queries to project only parts of interest. This approach, *which is currently not available in standard SPARQL implementations* could be used to materialize the portion of the graph needed to cap-

ture RDFS inferences. Both data materialization and changes required to SPARQL processors (to support recursion) go against the idea of EPPs that provide expressive SPARQL navigational queries (also under the ρdf entailment regime) with no materialization and no changes to existing SPARQL processors.

10. Concluding Remarks

We introduced EPPs, a significant extension of property paths, the current *navigational core* of SPARQL, the standard query language for querying KGs based on RDF. We underlined several practical advantages of adopting such extension. Our study also offers interesting theoretical observations, among which: (i) we identified a precise fragment of SPARQL that can capture non-recursive EPPs thus providing an indirect analysis of the navigational expressiveness of SPARQL; (iii) we have studied the expressiveness of EPPs as compared to PPs; (iii) we have also studied the expressiveness of SPARQL with respect to the ρdf entailment regime when considering different navigational cores, and identified those that can be supported on existing processors and those that require changes. Overall, we think that the practical and theoretical contributions of our work can help pave the way toward extending the navigational core of SPARQL and incorporate query-based reasoning capabilities. A promising direction of future work is to study how optimization techniques devised for SPARQL property paths [52] can be applied to extended property paths.

References

[1] G.W. Markus Krötzsch, Special Issue on Knowledge Graphs, *Journal of Web Semantics* **37-38** (2016), 53–54.

[2] Google Knowledge Graph: http://www.google.com/insidesearch/features/search/knowledge.html. http://www.google.com/insidesearch/features/search/knowledge.html.

[3] Facebook Graph: https://www.facebook.com/about/graphsearch. https://www.facebook.com/about/graphsearch.

[4] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak and S. Hellmann, DBpedia-A crystallization point for the Web of Data, *Web Semantics: science, services and agents on the world wide web* **7**(3) (2009), 154–165.

[5] F.M. Suchanek, G. Kasneci and G. Weikum, Yago: a core of semantic knowledge, in: *Proceedings of the 16th international conference on World Wide Web*, ACM, 2007, pp. 697–706.

[6] D. Vrandečić and M. Krötzsch, Wikidata: a free collaborative knowledgebase, *Communications of the ACM* **57**(10) (2014), 78–85.

[7] R. Angles, M. Arenas, G.H. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, H. Voigt et al., G-CORE: A Core for Future Graph Query Languages, in: *SIGMOD*, 2018.

[8] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*, Morgan & Claypool, 2011.

[9] W.W.W. Consortium et al., RDF 1.1 concepts and abstract syntax (2014).

[10] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, 2013.

[11] J. Pérez, M. Arenas and C. Gutierrez, Semantics and Complexity of SPARQL., *ACM TODS* **34**(3) (2009).

[12] M. Arenas, S. Conca and J. Pérez, Counting Beyond a Yottabyte, or how SPARQL 1.1 Property Paths will Prevent Adoption of the Standard, in: *Proc. of WWW*, 2012, pp. 629–638.

[13] P. Barceló, L. Libkin, A.W. Lin and P.T. Wood, Expressive Languages for Path Queries over Graph-Structured Data, *ACM TODS* **37**(4) (2012), 31.

[14] V. Fionda, G. Pirrò and C. Gutierrez, NautiLOD: A Formal Language for the Web of Data Graph, *ACM Trans. on the Web* **9**(1) (2015), 1–543.

[15] J.L. Reutter, A. Soto and D. Vrgoc, Recursion in SPARQL, in: *Proc. of ISWC*, 2015, pp. 19–35.

[16] J. Pérez, M. Arenas and C. Gutierrez, nSPARQL: A Navigational Language for RDF., *J. Web Sem.* **8**(4) (2010).

[17] V. Fionda, G. Pirrò and M.P. Consens, Extended Property Paths: Writing More SPARQL Queries in a Succinct Way, in: *Proc. of AAAI*, 2015.

[18] E. Prud'hommeaux, S. Harris and A. Seaborne, SPARQL 1.1 Query Language, Technical Report, W3C, 2013. http://www.w3.org/TR/sparql11-query.

[19] S. Muñoz, J. Pérez and C. Gutierrez, Simple and Efficient Minimal RDFS, *J. Web Sem.* **7**(3) (2009), 220–234.

[20] R. Angles and C. Gutierrez, The multiset semantics of SPARQL patterns, in: *International Semantic Web Conference*, Springer, 2016, pp. 20–36.

[21] F. Alkhateeb, J.-F. Baget and J. Euzenat, Constrained Regular expressions for Answering RDF-path Queries Modulo RDFS, *IJWIS* **10**(1) (2014), 24–50.

[22] K. Losemann and W. Martens, The Complexity of Evaluating Path Expressions in SPARQL, in: *Proc. of PODS*, 2012.

[23] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie and J. Siméon, XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation 14 December 2010, 2010.

[24] E. Franconi, C. Gutierrez, A. Mosca, G. Pirrò and R. Rosati, The Logic of Extensional RDFS, in: *Proc. of ISWC*, 2013, pp. 101–116.

[25] C. Gutierrez, C. Hurtado and A.O. Mendelzon, Foundations of Semantic Web Databases, in: *Proc. of PODS*, 2004, pp. 95–106.

[26] R. Kontchakov, M. Rezk, M. Rodríguez-Muro, G. Xiao and M. Zakharyaschev, Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime, in: *Proc. of ISWC*, 2014.

[27] S. Bischof, M. Krötzsch, A. Polleres and S. Rudolph, Schema-agnostic Query Rewriting in SPARQL 1.1, in: *Proc. of ISWC*, 2014.

[28] B. Glimm, Using SPARQL with RDFS and OWL entailment, in: *Reasoning Web*, 2011, pp. 137–201.

[29] J. Kim, H. Shin, W.-S. Han, S. Hong and H. Chafi, Taming Subgraph Isomorphism for RDF Query Processing, *Proc. of VLDB Endowment* **8**(11) (2015), 1238–1249.
[30] A.V. Aho, M.R. Garey and J.D. Ullman, The Transitive Reduction of a Directed Graph, *SIAM J. Comput.* **1**(2) (1972), 131–137.
[31] D. Calvanese, G. De Giacomo and M. Lenzerini, Conjunctive Query Containment and Answering under Description Logic Constraints, *ACM TOCL* **9**(3) (2008), 22.
[32] M.P. Consens and A.O. Mendelzon, GraphLog: a Visual Formalism for Real Life Recursion, in: *Proc. of PODS*, 1990, pp. 404–416.
[33] A. Mendelzon and P.T. Wood, Finding Regular Simple Paths in Graph Databases., *SIAM J. Comput.* **24**(6) (1995).
[34] R. Angles and C. Gutierrez, Survey of Graph Database Models, *ACM COMPUT SURV* **40**(1) (2008), 1.
[35] P.T. Wood, Query Languages for Graph Databases., *SIGMOD Rec.* (2012).
[36] P. Barceló, Querying Graph Databases, in: *Proc. of PODS*, 2013.
[37] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung and G. Lausen, RDFPath: Path Query Processing on Large RDF Graphs with MapReduce, in: *Proc. of ESWC Workshops*, 2011, pp. 50–64.
[38] X. Zhang and J. Van den Bussche, On the Power of SPARQL in Expressing Navigational Queries, *COMPUT J* **58**(11) (2015), 2841–2851.
[39] H. Zauner, B. Linse, T. Furche and F. Bry, A RPL through RDF: Expressive Navigation in RDF Graphs., in: *Proc. of RR*, 2010.
[40] O. Hartig and J. Pérez, Semantics and Complexity of GraphQL, in: *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2018, pp. 1155–1164.
[41] L. Libkin, J. Reutter and D. Vrgoč, TriaL for RDF: adapting graph query languages for RDF data, in: *Proc. of PODS*, 2013.
[42] M. Arenas, G. Gottlob and A. Pieris, Expressive Languages for Querying the Semantic Web, in: *Proc. of PODS*, 2014.
[43] L. Libkin, W. Martens and D. Vrgoč, Querying graph databases with XPath, in: *Proc. of ICDT*, 2013, pp. 129–140.
[44] S. Rudolph and M. Krötzsch, Flag & check: Data access with monadically defined queries, in: *Proc. of PODS*, 2013, pp. 151–162.
[45] G. Stefanoni, B. Motik, M. Krötzsch and S. Rudolph, The Complexity of Answering Conjunctive and Navigational Queries over OWL 2 EL Knowledge Bases, *JAIR* (2014), 645–705.
[46] M. Acosta and M.-E. Vidal, Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data, in: *Proc. of ISWC*, 2015, pp. 111–127.
[47] O. Hartig and J. Pérez, LDQL: A Query Language for the Web of Linked Data, in: *Proc. of ISWC*, Springer, 2015, pp. 73–91.
[48] O. Hartig and G. Pirrò, A Context-based Semantics for SPARQL Property Paths over the Web, in: *Proc. of ESWC*, 2015, pp. 71–87.
[49] S. Schaffert, C. Bauer, T. Kurz, F. Dorschel, D. Glachs and M. Fernandez, The Linked Media Framework: Integrating and Interlinking Enterprise Media Content and Data, in: *Proc. of I-SEMANTICS*, 2012, pp. 25–32.
[50] E.V. Kostylev, J.L. Reutter, M. Romero and D. Vrgoc, SPARQL with Property Paths, in: *Proc. of ISWC*, 2015, pp. 3–18.
[51] W.C. Melisachew and G. Pirrò, Containment of Expressive SPARQL Navigational Queries, in: *ISWC*, 2016, To appear.
[52] N. Yakovets, P. Godfrey and J. Gryz, Query planning for evaluating SPARQL property paths, in: *Proceedings of the 2016 International Conference on Management of Data*, ACM, 2016, pp. 1875–1889.

Appendix A. Detailed Experiments in Section 8.3

Table 13
DBpedia results for R5 (`rdf:type`)

		Result Count		Time(ms)	
Seed Entity	**QId**	**No reasoning**	ρdf	**No reasoning**	ρdf
dbp:%EC%83%AE	Q1	0	2	86	122
dbp:Texas_(Lasse_Stefanz_album)	Q2	9	45	86	747
dbp:Abul_Qasim_ibn_Mohammed_al-Ghassani	Q3	35	46	95	526
dbp:Emiko_Tsukada	Q4	9	16	94	118
dbp:Airbus_Military_S.A.S.	Q5	1	1	95	124
dbp:Variazh	Q6	26	60	86	561
dbp:Ralph_Golen__2	Q7	10	24	95	166
dbp:Thomas_Richardson_(Middlesbrough)	Q8	9	20	87	111
dbp:Lex_Richardson	Q9	41	59	95	137
dbp:Tracy_Mann	Q10	25	36	105	275
dbp:Montaigut,_Puy-de-D%C3%B4me	Q11	18	38	84	137
dbp:(18651)_1998_FP11	Q12	0	2	86	121
dbp:Sumida_River	Q13	18	38	103	123
dbp:Contact,_Nevada	Q14	22	40	992	136
dbp:1962âĂŞ63_West_Ham_United_F.C._season	Q15	0	1	99	652
dbp:Vecherniy_Bishkek	Q16	20	37	102	482
dbp:Basilides,_Cyrinus,_Nabor_and_Nazarius	Q17	14	25	86	180
dbp:Cyrtolepis	Q18	0	2	95	113
dbp:Yoxford	Q19	19	40	94	108
dbp:Mass_Destruction_(video_game)	Q20	25	65	85	114
dbp:Marian_Kozovy	Q21	3	3	86	111
dbp:Aghuzbon,_Savadkuh	Q22	9	36	84	116
dbp:Eero_Saari	Q23	2	2	85	124
dbp:The_Reason_Why_I'm_Talking_S–t	Q24	9	47	94	110
dbp:Geelong_West_Football_Club	Q25	11	14	115	111
dbp:V1_500m_at_the_2011_Pacific_Games	Q26	0	3	388	119
dbp:Little_Negro_Bu-ci-bu	Q27	16	34	93	122
dbp:NTV-NBC	Q28	0	2	976	145
dbp:Maridi_Airport	Q29	12	18	102	118
dbp:Korokchi	Q30	10	36	94	116
dbp:Haki_St%C3%ABrmilli	Q31	26	39	91	126
dbp:G%C3%B6rel_Crona	Q32	9	23	98	132
dbp:Lord_Lisle	Q33	2	2	92	114
dbp:Category:1841_in_Portugal	Q34	1	3	102	113
dbp:Nhill	Q35	25	46	89	107
dbp:Koeberliniaceae	Q36	10	12	853	118
dbp:Probulov	Q37	24	48	90	137
dbp:Pauline_Pepinsky	Q38	9	16	998	132
dbp:Acorda_Therapeutics	Q39	23	40	836	118
dbp:Armagetron_Advanced	Q40	31	65	435	115
dbp:Didihat	Q41	29	57	334	115
dbp:Brook_Glacier	Q42	13	20	100	120
dbp:Western_Union_(schooner)	Q43	23	44	88	110
dbp:Fearon	Q44	0	2	95	117
dbp:Scaphella_neptunia	Q45	8	11	87	116
dbp:Sebadani_Dam__2	Q46	8	33	87	155
dbp:Derek_Gaudet__5	Q47	10	24	96	112
dbp:John_Orsino	Q48	50	68	99	129
dbp:Holy_orders	Q49	1	6	342	115
dbp:Our_Lady_of_Lourdes_School_(disambiguation)	Q50	1	1	506	487

Table 14
Yago results for R5 (`rdf:type`)

		Result Count		Time(ms)	
Seed Entity	**QId**	**No reasoning**	ρdf	**No reasoning**	ρdf
yago:A_Hard_Road	Q1	9	18	306	428
yago:A_Pizza_Tweety_Pie	Q2	3	19	18	202
yago:A_Word_in_Your_Ear	Q3	3	25	18	179
yago:Aap_Ke_Deewane	Q4	4	19	20	152
yago:Abbo_II_of_Metz	Q5	6	31	18	282
yago:Abdul_Ilah_Khatib	Q6	11	57	18	282
yago:About_Face_(film)	Q7	3	18	18	147
yago:Aerolysin	Q8	1	12	19	126
yago:Affair_in_Trinidad	Q9	9	25	20	201
yago:Agni_Yudham	Q10	4	19	18	150
yago:Agricola_(book)	Q11	3	17	17	178
yago:Ahmed_Hadid_Al_Mukhaini	Q12	7	23	21	159
yago:AIDS_Action_Committee_of_Massachusetts	Q13	3	13	18	115
yago:AIDS_Sutra	Q14	1	13	18	176
yago:Aigen	Q15	1	12	20	125
yago:AÃŕgue_Longue	Q16	3	12	24	177
yago:Aisha_Dee	Q17	5	26	18	264
yago:Al_Jalahma	Q18	2	9	22	135
yago:Alan_Dowding	Q19	12	36	20	227
yago:Alan_Smith_(Welsh_footballer)	Q20	5	27	17	249
yago:Alarilla	Q21	3	19	16	188
yago:Albert_Dubois-Pillet	Q22	8	35	17	222
yago:Albert_Glover	Q23	1	22	17	249
yago:Alec_Soth	Q24	8	42	24	274
yago:Aleksandr_Rymanov	Q25	7	29	16	252
yago:AlÃĺne	Q26	3	12	17	190
yago:Alexandra_Feodorovna_(Charlotte_of_Prussia)	Q27	14	51	16	276
yago:All_About_Anna	Q28	9	25	18	192
yago:All_That_I_Am_(Santana_album)	Q29	8	16	19	155
yago:All_the_Way..._A_Decade_of_Song	Q30	18	26	19	161
yago:Almost_a_Gentleman	Q31	6	21	16	126
yago:Along_the_Way_(TV_series)	Q32	3	25	17	166
yago:Ambush_Bay	Q33	7	22	15	144
yago:Aminabad,_Sindh	Q34	1	14	24	201
yago:Ampang_Park_LRT_station	Q35	0	0	20	18
yago:And_Hell_Will_Follow_Me	Q36	6	15	16	155
yago:Andalusian_horse	Q37	3	12	15	136
yago:Andre_Norton_Award	Q38	5	12	15	138
yago:Andy_Jones_(producer)	Q39	2	23	15	260
yago:Andy_Valmorbida	Q40	2	25	16	279
yago:Anema_(lichen)	Q41	1	15	15	197
yago:Aneta_PospÃ■ÅąilovÃą	Q42	3	29	20	265
yago:Angela_Chalmers	Q43	11	40	20	250
yago:Angola_Fire_Department_(Louisiana)	Q44	1	18	15	191
yago:Anna_Catharina_von_BÃd'rfelt	Q45	3	28	15	270
yago:Annapurna_High_School	Q46	5	21	15	190
yago:Annet,_Isles_of_Scilly	Q47	8	22	15	200
yago:Annette_Sikveland	Q48	7	41	16	264
yago:Aonghas_ÃŞg_of_Islay	Q49	4	32	8	279
yago:Aqualillies	Q50	2	21	21	239

Table 15
LinkedMDB results for R5 (`rdf:type`)

		Result Count		Time(ms)	
Seed Entity	**QId**	**No reasoning**	ρdf	**No reasoning**	ρdf
lmdb-actor:1	Q1	2	2	98	99
lmdb-actor:10	Q2	2	2	28	61
lmdb-actor:10000	Q3	2	2	27	63
lmdb-actor:10001	Q4	2	2	26	57
lmdb-actor:10009	Q5	2	2	29	51
lmdb-actor:1001	Q6	2	2	27	51
lmdb-actor:10010	Q7	2	2	29	60
lmdb-actor:10013	Q8	2	2	26	73
lmdb-actor:10014	Q9	2	2	32	58
lmdb-actor:10016	Q10	2	2	25	51
lmdb-actor:10017	Q11	2	2	36	56
lmdb-actor:10018	Q12	2	2	26	59
lmdb-actor:10023	Q13	2	2	39	51
lmdb-actor:10027	Q14	2	2	29	67
lmdb-actor:10029	Q15	2	2	31	63
lmdb-actor:10030	Q16	2	2	40	90
lmdb-actor:10034	Q17	2	2	41	125
lmdb-actor:10035	Q18	2	2	41	64
lmdb-actor:10038	Q19	2	2	45	80
lmdb-actor:10039	Q20	2	2	30	49
lmdb-film:10504	Q21	1	1	41	54
lmdb-film:10508	Q22	1	1	34	51
lmdb-film:10510	Q23	1	1	36	50
lmdb-film:10894	Q24	1	1	49	47
lmdb-film:10895	Q25	1	1	41	48
lmdb-film:10896	Q26	1	1	37	44
lmdb-film:10897	Q27	1	1	36	54
lmdb-performance:108172	Q28	1	1	28	43
lmdb-performance:108173	Q29	1	1	35	74
lmdb-performance:108174	Q30	1	1	37	42
lmdb-performance:108175	Q31	1	1	36	44
lmdb-performance:108176	Q32	1	1	35	44
lmdb-performance:108177	Q33	1	1	24	44
lmdb-performance:108178	Q34	1	1	28	44
lmdb-mc:1810	Q35	1	1	29	43
lmdb-mc:1811	Q36	1	1	23	41
lmdb-mc:1812	Q37	1	1	29	49
lmdb-mc:1817	Q38	1	1	24	50
lmdb-mc:1819	Q39	1	1	26	40
lmdb-mc:1820	Q40	1	1	30	40
lmdb-mc:1822	Q41	1	1	23	46
lmdb-mc:1823	Q42	1	1	23	40
lmdb-mc:1826	Q43	1	1	30	42
lmdb-mc:1828	Q44	1	1	23	47
lmdb-mc:1830	Q45	1	1	22	47
lmdb-mc:1838	Q46	1	1	23	45
lmdb-producer:10111	Q47	1	1	24	42
lmdb-producer:10112	Q48	1	1	24	43
lmdb-producer:10113	Q49	1	1	25	44
lmdb-producer:10114	Q50	1	1	22	42

Table 16
DBpedia results for R6 on the predicate `dbo:genre`

		Result Count		Time(ms)	
Seed Entity	**QId**	**No reasoning**	ρdf	**No reasoning**	ρdf
dbp:Night_Surf	Q1	0	1	554	864
dbp:The_Last_Man	Q2	0	1	90	142
dbp:Metro_2035	Q3	0	1	92	110
dbp:The_Last_Ship_(novel)	Q4	0	1	114	131
dbp:Taronga	Q5	0	1	99	129
dbp:The_Third_World_War_(novel)	Q6	0	1	99	128
dbp:The_Sending	Q7	0	3	100	100
dbp:Desecration_(novel)	Q8	0	3	100	108
dbp:The_Girl_Who_Owned_a_City	Q9	0	3	99	121
dbp:The_Sword_of_the_Lady	Q10	0	2	98	156
dbp:Shikari_in_Galveston	Q11	0	3	107	113
dbp:Fitzpatrick's_War	Q12	0	2	101	118
dbp:Sykom	Q13	0	3	110	124
dbp:So_This_Is_How_It_Ends	Q14	0	1	139	109
dbp:On_the_Beach_(novel)	Q15	0	1	126	131
dbp:The_Postman	Q16	0	1	101	123
dbp:Swan_Song_(novel)	Q17	0	1	99	106
dbp:The_Children's_Hospital	Q18	0	1	123	98
dbp:Piter_(novel)	Q19	0	1	99	116
dbp:Mutants_in_Orbit	Q20	0	1	107	110
dbp:Zone_One	Q21	0	3	89	100
dbp:Apollyon_(novel)	Q22	0	3	92	120
dbp:Armageddon_(novel)	Q23	0	3	89	118
dbp:Assassins_(LaHaye_novel)	Q24	0	3	98	105
dbp:Glorious_Appearing	Q25	0	3	90	109
dbp:Left_Behind_(novel)	Q26	0	2	97	128
dbp:Nicolae_(novel)	Q27	0	3	90	97
dbp:The_Indwelling	Q28	0	3	95	113
dbp:The_Mark_(novel)	Q29	0	3	127	103
dbp:The_Rapture_(novel)	Q30	0	3	103	115
dbp:The_Remnant_(novel)	Q31	0	3	106	97
dbp:The_100_(novel)	Q32	0	4	89	106
dbp:Caesar's_Column	Q33	0	1	102	99
dbp:Pandemia_(book)	Q34	0	4	89	99
dbp:The_Maze_Runner	Q35	0	3	101	111
dbp:The_Road	Q36	0	1	94	116
dbp:Warm_Bodies	Q37	0	4	116	105
dbp:Blood_Red_Road	Q38	0	1	100	96
dbp:The_Twelve_(novel)	Q39	0	6	111	131
dbp:Dies_the_Fire	Q40	0	3	98	105
dbp:The_Walking_Dead	Q41	0	0	98	127
dbp:The_Passage_(novel)	Q42	0	6	99	155
dbp:Metro_2033_(novel)	Q43	0	1	107	163
dbp:Metro_2034	Q44	0	1	104	105
dbp:Fever_Crumb_Series	Q45	0	3	98	103
dbp:Mutants_of_the_Yucatan	Q46	0	1	98	109
dbp:Road_Hogs	Q47	0	1	96	115
dbp:Brother_in_the_Land	Q48	0	2	90	115
dbp:_Rise_of_the_Governor	Q49	0	2	126	102
dbp:Cannibal_Reign	Q50	0	1	133	110

Table 17
DBpedia results for R6 on the predicate `dbo:location`

		Result Count		Time(ms)	
Seed Entity	**QId**	**No reasoning**	ρdf	**No reasoning**	ρdf
dbp:Bayou_Corne_sinkhole	Q1	0	2	518	752
dbp:Lake_Ophelia_National_Wildlife_Refuge	Q2	0	1	89	186
dbp:Calcasieu_Lake	Q3	0	4	87	210
dbp:Lacassine_National_Wildlife_Refuge	Q4	0	3	86	187
dbp:Sabine_Pass_Lighthouse	Q5	0	2	96	229
dbp:Sabine_National_Wildlife_Refuge	Q6	0	1	97	104
dbp:Grand_Lake_(Louisiana)	Q7	0	1	105	103
dbp:East_Cove_National_Wildlife_Refuge	Q8	0	1	119	118
dbp:Cameron_Prairie_National_Wildlife_Refuge	Q9	0	1	95	125
dbp:Catahoula_National_Wildlife_Refuge	Q10	0	4	95	139
dbp:Sandy_Lake,_Louisiana	Q11	0	1	96	129
dbp:Chicot_State_Park	Q12	0	2	96	138
dbp:Louisiana_State_Arboretum	Q13	0	2	88	140
dbp:Bogue_Chitto_State_Park	Q14	0	2	96	181
dbp:Great_Salt_Plains_State_Park	Q15	0	3	86	184
dbp:Salt_Plains_National_Wildlife_Refuge	Q16	0	1	91	187
dbp:Great_Salt_Plains_Lake	Q17	0	1	86	183
dbp:Tilicho_Lake	Q18	0	2	94	208
dbp:Berney_Ar_railway_station	Q19	0	0	95	127
dbp:St_Nicholas,_Blakeney	Q20	0	1	104	109
dbp:Blakeney_Windmill	Q21	0	1	86	126
dbp:Bracknell_railway_station	Q22	0	2	103	108
dbp:Crowthorne_railway_station	Q23	0	2	96	103
dbp:Martins_Heron_railway_station	Q24	0	2	86	103
dbp:Fort_Cobb_State_Park	Q25	0	3	95	126
dbp:Lake_Ellsworth_(Oklahoma)	Q26	0	4	95	126
dbp:Red_Rock_Canyon_State_Park_(Oklahoma)	Q27	0	3	87	113
dbp:Fort_Cobb_Reservoir	Q28	0	1	95	99
dbp:Caister-on-Sea_railway_station	Q29	0	2	95	98
dbp:Caister_Camp_Halt_railway_station	Q30	0	2	86	110
dbp:Chalk_Farm_tube_station	Q31	0	3	95	134
dbp:Roundhouse_(venue)	Q32	0	3	89	101
dbp:Cockfosters_tube_station	Q33	0	3	95	98
dbp:Trent_Park	Q34	0	1	97	121
dbp:Pelion_Gap	Q35	0	4	93	103
dbp:Rio_Cinema_(Dalston)	Q36	0	2	96	117
dbp:Dalston_Kingsland_railway_station	Q37	0	3	94	130
dbp:Dalston_Junction_railway_station	Q38	0	3	85	101
dbp:Eltead_Woods_railway_station	Q39	0	0	96	112
dbp:Fort_Arbuckle_(Oklahoma)	Q40	0	2	85	157
dbp:Perry_Island_(Queensland)	Q41	0	1	86	119
dbp:Turtle_Head_Island	Q42	0	1	105	127
dbp:Gunnersbury_station	Q43	0	3	86	140
dbp:Kew_Bridge_railway_station	Q44	0	4	86	140
dbp:Harold_Wood_railway_station	Q45	0	3	145	165
dbp:Hatch_End_railway_station	Q46	0	3	105	137
dbp:Green-Works	Q47	0	2	86	99
dbp:St_John_the_Baptist,_Hoxton	Q48	0	2	121	142
dbp:Hoxton_railway_station	Q49	0	3	103	167
dbp:Great_Plains_State_Park	Q50	0	3	85	97

Table 18
LDCache results for R6 on the predicate `yago:hasLocation`

		Result Count		Time(ms)	
Seed Entity	**QId**	**No reasoning**	ρdf	**No reasoning**	ρdf
yago:Aberdeen	Q1	0	2	518	752
yago:A._Reyrolle_&_Company	Q2	0	1	89	186
yago:AD_Torreforta	Q3	0	4	87	210
yago:AFI_Conservatory	Q4	0	3	86	187
yago:ALZ_(steelworks)	Q5	0	2	96	229
yago:APSA_Colombia	Q6	0	1	97	104
yago:ASFA_Soccer_League	Q7	0	1	105	103
yago:ASTM_International	Q8	0	1	119	118
yago:ATP_Challenger_Guangzhou	Q9	0	1	95	125
yago:ATP_Challenger_La_Serena	Q10	0	4	95	139
yago:A_Home_at_the_End_of_the_World_(film)	Q11	0	1	96	129
yago:A_Sharp_Intake_of_Breath	Q12	0	2	96	138
yago:Aabach_(Afte)	Q13	0	2	88	140
yago:Aacay_Organization	Q14	0	2	96	181
yago:Aach,_Baden-WÃijrttemberg	Q15	0	3	86	184
yago:Aachen_Central_Station	Q16	0	1	91	187
yago:Aaniiih_Nakoda_College	Q17	0	1	86	183
yago:Aaronsburg_Historic_District	Q18	0	2	94	208
yago:Aavahelukka_Airfield	Q19	0	0	95	127
yago:Abandoned_Pennsylvania_Turnpike	Q20	0	1	104	109
yago:Abashiri_Quasi-National_Park	Q21	0	1	86	126
yago:Abbeville_Historic_District_(Abbeville,_South_Carolina)	Q22	0	2	103	108
yago:Abel_I._Smith_Burial_Ground	Q23	0	2	96	103
yago:Abel_Iturralde_Province	Q24	0	2	86	103
yago:Abenteuermuseum_(SaarbrÃijcken)	Q25	0	3	95	126
yago:Aberdeen_Historic_District_(Aberdeen,_South_Dakota)	Q26	0	4	95	126
yago:Aberfan_disaster	Q27	0	3	87	113
yago:AbukumaExpress	Q28	0	1	95	99
yago:Academy_of_Korean_Studies	Q29	0	2	95	98
yago:Academy_of_the_Canyons	Q30	0	2	86	110
yago:Accra_Sports_Stadium	Q31	0	3	95	134
yago:Acheron,_Victoria	Q32	0	3	89	101
yago:Acheron_Boys_Home	Q33	0	3	95	98
yago:Achimota_School	Q34	0	1	97	121
yago:Acme,_Washington	Q35	0	4	93	103
yago:Acquaviva_Picena	Q36	0	2	96	117
yago:AdOn_Network	Q37	0	3	94	130
yago:Ada,_Croatia	Q38	0	3	85	101
yago:Adabay_River	Q39	0	0	96	112
yago:Adaganahalli	Q40	0	2	85	157
yago:Adair,_Idaho	Q41	0	1	86	119
yago:Adak,_Alaska	Q42	0	1	105	127
yago:Adak_Airport	Q43	0	3	86	140
yago:Adakanahalli	Q44	0	4	86	140
yago:Adakatahalli	Q45	0	3	145	165
yago:Adalin_River	Q46	0	3	105	137
yago:Adam's_Green	Q47	0	2	86	99
yago:Adam_&_Steve	Q48	0	2	121	142
yago:Adam_Airport	Q49	0	3	103	167
yago:Adam_Orris_House	Q50	0	3	85	97