Extended Property Paths: Writing More SPARQL Queries in a Succinct Way*

Valeria Fionda[1], Giuseppe Pirrò[2] and Mariano P. Consens[3]
[1] DeMaCS, University of Calabria, Italy
[2] WeST, University of Koblenz-Landau, Germany
[3] MIE, University of Toronto, Canada

Abstract

We introduce Extended Property Paths (EPPs), a significant enhancement of SPARQL property paths. EPPs allow to capture in a succinct way a larger class of *navigational queries* than property paths. We present the syntax and formal semantics of EPPs and introduce two different evaluation strategies. The first is based on an algorithm implemented in a custom query processor. The second strategy leverages a translation algorithm of EPPs into SPARQL queries that can be executed on existing SPARQL processors. We compare the two evaluation strategies on real data to highlight their pros and cons.

1 Introduction

Motivated by the spread of graph-like data, the area of graph databases has received renewed attention. SPARQL, the W3C standard query language for data in RDF, has recently been enhanced with *property paths* (Harris and Seaborne 2013) to support graph navigation capabilities. The regular-expression-like syntax of property paths enables to write SPARQL navigational queries in a more *succinct* way and extend the matching of triple patterns to arbitrary length paths.

However, property paths offer limited navigation capabilities; queries like "*Find my exclusive friends*", that is, my friends that are not friend of any of my friends, cannot be expressed. To enhance the expressive power of property paths and enable to write more navigational queries in a succinct way, we introduce Extended Property Paths (EPPs). In designing EPPs we identified a core of new features and investigated how to make available such features; indeed, one can devise a custom query processor like in (Alkhateeb, Baget, and Euzenat 2009; Fionda and Pirrò 2013) and/or leverage existing (SPARQL) processors. The goal of this paper is to discuss the design and implementation of EPPs.

Related Work. Graph query languages have been deeply studied (Wood 2012). It emerged that for certain classes of languages, like Conjunctive Regular Path Queries (CRPQs) (Barceló et al. 2012; Alkhateeb, Baget, and Euzenat 2009), the evaluation problem can become too expensive, making the class unattractive for practical purposes. Hence, restricted classes of languages like acyclic CRPQs and Nested Regular Expressions (NREs) (Pérez, Arenas, and Gutierrez 2010) have been proposed. NREs are at the core of navigational languages for RDF such as nSPARQL (Pérez, Arenas, and Gutierrez 2010) and NautiLOD (Fionda, Gutierrez, and Pirrò 2012). Languages like `TriAL` (Libkin, Reutter, and Vrgoč 2013), `TriQ` (Arenas, Gottlob, and Pieris 2014) and NEMODEQ (Rudolph and Krötzsch 2013) are grounded on (extensions of) Datalog. Other expressive languages like SPARQLeR (Kochut and Janik 2007) and *extended* CRPQs (Barceló et al. 2012) focus on discovering paths.

In terms of expressiveness, Barceló et al. (Barceló, Pérez, and Reutter 2012) showed that NREs *cannot express* queries involving, for instance, path conjunction. SPARQL property paths are even less expressive than NREs because of the lack of nesting and tests also within (arbitrary length) paths. `TriAL` can capture queries not captured by NREs and nSPARQL. However, NREs/nSPARQL queries can be evaluated in linear time. Arenas et al. (Arenas, Gottlob, and Pieris 2014) studied the relationships between `TriQ`, `TriAL` and NEMODEQ in terms of Datalog$^{\pm}$ programs (Cali et al. 2010). Fletcher et al. (Fletcher et al. 2011) compared the expressiveness of navigational languages by considering different sets of features. We compare EPPs with closely-related languages in Section 3.3.

EPPs extend the expressive power of SPARQL property paths and NREs-based languages with new features such as *path conjunction*, *path negation* and *powerful types of tests*. These features are partially available in the W3C standard language XPath 2.0 used to query tree-like data (Berglund et al. 2010). From a concrete point of view, with EPPs we faced two main challenges: *(i)* how to make EPPs a conservative extension of SPARQL property paths; *(ii)* how to make EPPs readily available in existing SPARQL processors.

1.1 EPPs by Example

We now give an overview of EPPs via a concrete example. The syntax and semantics of EPPs will be discussed in detail

*V. Fionda's work was supported by the European Commission, the European Social Fund and the Calabria region. G. Pirrò was partially supported by the EU Framework Programme for Research and Innovation under grant agreement no. 611242 (Sense4us).

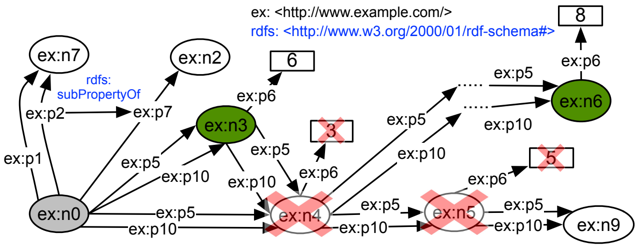

Figure 1: An example of RDF graph.

in Section 2.

Example 1 (Extended Property Paths). By considering the graph in Fig. 1, find pairs of nodes[1] (n_i, n_j) simultaneously connected via two paths: one labeled as p_5 and the other labeled as p_{10}; moreover, from n_j an edge labeled as p_6 that leads to nodes having value <6 must not exists.

NREs-based languages (and PPs) *cannot* express such request due to the lack of path conjunction/negation. With EPPs it can be expressed as:

$$\mathtt{?x}\ ((\mathrm{p}_5\&\mathrm{p}_{10})\sim(\mathrm{p}_5\&\&\mathtt{TP}(\_\mathrm{o}, \mathrm{p}_6\&\&\mathtt{T}(\_\mathrm{o}<6))))\ \mathtt{?y}$$

The path conjunction & enables to check that both paths p_5 and p_{10} connect n_i with n_j, while path negation $\sim$ enables to discard from the set of nodes satisfying the first path those that also satisfy the second one. Note also that SPARQL property paths and NREs-based languages *lack tests like* TP, to check the existence of a path (via p_6) to nodes having value <6.

The evaluation of the EPP expression with the algorithm described in Section 3 starts from all the *bindings* of the variable ?x; we discuss the case ?x$\rightarrow\mathrm{n}_0$. From n_0, the evaluation of the first path $(\mathrm{p}_5\&\mathrm{p}_{10})$ enables to reach n_3. The second path (negated via $\sim$) is evaluated again from n_0 and consists in the logical AND (via &&) of two tests. The first test checks for the existence of an edge p_5, which leads to n_3. The second test (i.e., TP) includes: *(i)* the position in the RDF triple from which the test starts; *(ii)* a path. The position symbol $\_\mathrm{o}$ means that the test starts from the object of the previous navigational step, that is, the object of the triple $(\mathrm{n}_0,\ \mathrm{p}_5,\ \mathrm{n}_3)$. From n_3, another logical AND of two tests is evaluated. The first one checks the existence of an edge p_6 and enables to reach the node 6. The second AND, which starts from the object of the previous step (i.e., 6), checks that the value is <6; in this case the test fails and n_3 *is not* included in the results of the negated path. Conversely, n_4 satisfies the test, it is included in the results of the negated path and thus, it is not a valid binding for ?y when considering ?x$\rightarrow\mathrm{n}_0$.

Overall, for ?x$\rightarrow$ n_0 we get ?y$\rightarrow$ n_3 meaning that from n_0 the navigation of the graph according to the expression enabled to reach n_3.

We have also devised a translation procedure (see Section 3.2) of EPPs into SPARQL. The translation of Example 1 is shown in Fig. 2; here, ?o_0_0_1_0_1_0 is a variable automatically generated by the translation. The advantage of the translation is that the query in Fig. 2 can be executed on *existing SPARQL processors*. From the syntactic point of view, the advantage of using EPPs to write navigational queries instead of writing them directly into pure SPARQL is that the same request can be expressed more succinctly and without the need to deal with intermediate variables.

```
SELECT DISTINCT ?x ?y WHERE {
   { ?x p5 ?y. ?x p10 ?y. }
   MINUS
   { ?x p5 ?y.
     FILTER EXISTS
     { ?y p6 ?o_0_0_1_0_1_0.
       FILTER( ?o_0_0_1_0_1_0 < 6 ) }}}
```

Figure 2: Translation into SPARQL of Example 1.

1.2 Contributions and Organization

The contributions of this paper are the following: *(i)* EPPs a language more expressive than NREs-based languages and SPARQL property paths; *(ii)* a concise syntax and a formal semantics for EPPs *(iii)* an algorithm for the evaluation of EPPs and an implementation in a custom query processor; *(iv)* a formalization and implementation of a translation procedure of EPPs into SPARQL; *(v)* a comparison between the custom query processor and Jena ARQ (for EPPs translated into SPARQL).

EPPs can be seen as a language per se, which goes beyond NREs-based languages and SPARQL property paths. Moreover, EPPs serve also the purpose of capturing a larger fragment of pure SPARQL *navigational queries* in a *succinct* way. To the best of our knowledge, EPPs are the first proposal covering both aspects.

The remainder of the paper is organized as follows. Section 2 presents EPPs; syntax and semantics. Section 3 introduces an evaluation algorithm for EPPs and the translation of EPPs into SPARQL. The implementation and evaluation of EPPs are discussed in Section 4. We conclude in Section 5.

2 Extended Property Paths

Let **U** (URIs), **B** (blank nodes), **L** (literals), and $\mathcal{V}$ (variables, starting withe the '?' symbol) be four pairwise disjoint, countably infinite sets. An *RDF triple* is a tuple of the form $\langle s, p, o\rangle \in (\mathbf{U}\cup\mathbf{B})\times\mathbf{U}\times(\mathbf{U}\cup\mathbf{B}\cup\mathbf{L})$. An RDF graph $\mathcal{G}$ is a set of triples. We indicate by $nodes(\mathcal{G}) \subseteq \mathbf{U}\cup\mathbf{B}\cup\mathbf{L}$ the set of URIs, blank nodes and literals that appear as subject or object of some triples in $\mathcal{G}$. Moreover, $\mathcal{T} \subseteq \mathbf{U}\cup\mathbf{B}\cup\mathbf{L}$ is the set of terms of $\mathcal{G}$.

2.1 Extended Property Paths Syntax

The syntax of EPPs is reported in Table 1.

Commonalities with SPARQL property paths (PPs)
The first line of the syntax covers the syntax of PPs (Harris and Seaborne 2013) (section 9.1); moreover, (Negated)PropertySets like '!(u_1 ||...^u_n)' that are

[1]The prefix ex: is omitted for sake of space.

epp	::=	‘^’ epp | epp ‘+’ | epp ‘?’ | epp‘*’ | epp ‘/’ epp | epp ‘|’ epp | ‘(’ epp ‘)’ | [pos][1] test [pos][2] | epp ‘&’ epp | epp ‘~’ epp
test	::=	‘!’ test | test ‘&&’ test | test ‘||’ test | ‘(’ test ‘)’ | base
base	::=	uri | ‘TP(’pos ‘,’ epp ‘)’ | ‘T(’EExp‘)’
EBuiltInCall	::=	BuiltInCall | pos
pos	::=	‘_s’ | ‘_p’ | ‘_o’

Table 1: Syntax of EPPs expressions. [1]If omitted is _s; [2]If omitted is _o.

R1	$\mathbf{E}[\![\hat{}\,\mathtt{epp}]\!]^{\mathcal{G}}$	:=	$\left\{(\mathtt{u},\mathtt{v}) : (\mathtt{v},\mathtt{u}) \in \mathbf{E}[\![\mathtt{epp}]\!]^{\mathcal{G}}\right\}$
R2	$\mathbf{E}[\![\mathtt{epp}_1/\mathtt{epp}_2]\!]^{\mathcal{G}}$	:=	$\left\{(\mathtt{u},\mathtt{v}) : \exists \mathtt{w}\ (\mathtt{u},\mathtt{w}) \in \mathbf{E}[\![\mathtt{epp}_1]\!]^{\mathcal{G}} \wedge (\mathtt{w},\mathtt{v}) \in \mathbf{E}[\![\mathtt{epp}_2]\!]^{\mathcal{G}}\right\}$
R3	$\mathbf{E}[\![(\mathtt{epp})^*]\!]^{\mathcal{G}}$	:=	$\{(\mathtt{u},\mathtt{u}) \mid \mathtt{u} \in nodes(\mathcal{G})\} \cup \bigcup_{i=1}^{\infty} \mathbf{E}[\![\mathtt{epp}_i]\!]^{\mathcal{G}} \mid \mathtt{epp}_1 = \mathtt{epp} \wedge \mathtt{epp}_i = \mathtt{epp}_{i-1}/\mathtt{epp}$
R4	$\mathbf{E}[\![(\mathtt{epp})^+]\!]^{\mathcal{G}}$	:=	$\bigcup_{i=1}^{\infty} \mathbf{E}[\![\mathtt{epp}_i]\!]^{\mathcal{G}} \mid \mathtt{epp}_1 = \mathtt{epp} \wedge \mathtt{epp}_i = \mathtt{epp}_{i-1}/\mathtt{epp}$
R5	$\mathbf{E}[\![(\mathtt{epp})?]\!]^{\mathcal{G}}$	:=	$\{(\mathtt{u},\mathtt{u}) \mid \mathtt{u} \in nodes(\mathcal{G})\} \cup \mathbf{E}[\![\mathtt{epp}]\!]^{\mathcal{G}}$
R6	$\mathbf{E}[\![(\mathtt{epp}_1 \mid \mathtt{epp}_2)]\!]^{\mathcal{G}}$	:=	$\left\{(\mathtt{u},\mathtt{v}) : (\mathtt{u},\mathtt{v}) \in \mathbf{E}[\![\mathtt{epp}_1]\!]^{\mathcal{G}} \vee (\mathtt{u},\mathtt{v}) \in \mathbf{E}[\![\mathtt{epp}_2]\!]^{\mathcal{G}}\right\}$
R7	$\mathbf{E}[\![(\mathtt{epp}_1 \& \mathtt{epp}_2)]\!]^{\mathcal{G}}$	:=	$\left\{(\mathtt{u},\mathtt{v}) : (\mathtt{u},\mathtt{v}) \in \mathbf{E}[\![\mathtt{epp}_1]\!]^{\mathcal{G}} \wedge (\mathtt{u},\mathtt{v}) \in \mathbf{E}[\![\mathtt{epp}_2]\!]^{\mathcal{G}}\right\}$
R8	$\mathbf{E}[\![(\mathtt{epp}_1 \sim \mathtt{epp}_2)]\!]^{\mathcal{G}}$	:=	$\left\{(\mathtt{u},\mathtt{v}) : (\mathtt{u},\mathtt{v}) \in \mathbf{E}[\![\mathtt{epp}_1]\!]^{\mathcal{G}} \wedge (\mathtt{u},\mathtt{v}) \notin \mathbf{E}[\![\mathtt{epp}_2]\!]^{\mathcal{G}}\right\}$
R9	$\mathbf{E}[\![\mathtt{pos}_1\ \mathtt{test}\ \mathtt{pos}_2]\!]^{\mathcal{G}}$	:=	$\left\{(P_m(\mathtt{pos}_1,\mathtt{t}), P_m(\mathtt{pos}_2,\mathtt{t}))) \mid \text{triple } \mathtt{t} \in \mathcal{G} \wedge \mathbf{E}_{\mathcal{T}}[\![\mathtt{test}]\!]^{\mathcal{G}}_{\mathtt{t}}\right\}$
R10	$\mathbf{E}_{\mathcal{T}}[\![\mathtt{u}]\!]^{\mathcal{G}}_{\mathtt{t}}$	:=	$P_m(\_p,\mathtt{t}) = \mathtt{u}$
R11	$\mathbf{E}_{\mathcal{T}}[\![\mathtt{T(EExp)}]\!]^{\mathcal{G}}_{\mathtt{t}}$	:=	$\mathtt{EvalSPARQLBuilt\text{-}in(EExp,t)}$
R12	$\mathbf{E}_{\mathcal{T}}[\![\mathtt{TP(pos,epp)}]\!]^{\mathcal{G}}_{\mathtt{t}}$	:=	$\exists v : (P_m(\mathtt{pos},\mathtt{t}), v) \in \mathbf{E}[\![\mathtt{epp}]\!]^{\mathcal{G}}$
R13	$\mathbf{E}_{\mathcal{T}}[\![\mathtt{test}_1 \&\& \mathtt{test}_2]\!]^{\mathcal{G}}_{\mathtt{t}}$	:=	$\mathbf{E}_{\mathcal{T}}[\![\mathtt{test}_1]\!]^{\mathcal{G}}_{\mathtt{t}} \wedge \mathbf{E}_{\mathcal{T}}[\![\mathtt{test}_2]\!]^{\mathcal{G}}_{\mathtt{t}}$
R14	$\mathbf{E}_{\mathcal{T}}[\![\mathtt{test}_1 \Vert \mathtt{test}_2]\!]^{\mathcal{G}}_{\mathtt{t}}$	:=	$\mathbf{E}_{\mathcal{T}}[\![\mathtt{test}_1]\!]^{\mathcal{G}}_{\mathtt{t}} \vee \mathbf{E}_{\mathcal{T}}[\![\mathtt{test}_2]\!]^{\mathcal{G}}_{\mathtt{t}}$
R15	$\mathbf{E}_{\mathcal{T}}[\![!\mathtt{test}]\!]^{\mathcal{G}}_{\mathtt{t}}$	:=	$\neg\mathbf{E}_{\mathcal{T}}[\![\mathtt{test}]\!]^{\mathcal{G}}_{\mathtt{t}}$

Table 2: Formal semantics of EPPs expressions.

a combination of forward/reverse predicates in a (negated) set are expressible in the EPPs syntax via the productions test → base → u. Note that EPPs use the symbol ‘||’ while PPs use ‘|’.

New features introduced by Extended Property Paths
The following *additional* features are introduced: path conjunction ($\mathtt{epp}_1$&$\mathtt{epp}_2$), path negation ($\mathtt{epp}_1$~$\mathtt{epp}_2$) and different types of tests (test) *within* a path, also by specifying the *starting* and *ending* positions (pos). EPPs enable to test from each of the subject, predicate and object positions in RDF triples, mapped in the syntax to _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 a triple.

Tests (test) can be of different types and can be combined by using the logical operators AND (&&), OR (||) and NOT (!). A test can be a simple check for the existence of a URI or a *nested* EPP, i.e., TP(pos, epp), which corresponds to the evaluation of epp starting from a position pos (of the last triple traversed) and whose evaluation returns true if, and only if, there exists at least one node that can be reached via epp. A base test (production base) can be of type T, which is a SPARQL boolean expression; here, EExp (not reported here for sake of space) extends the production [110] in the SPARQL grammar[2] where BuiltInCall[3] is substituted with EBuiltInCall, 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 $\mathbf{U} \cup \mathbf{L}$ and constants, logical connectives (¬, ∧, ∨), (in)equality symbol(s) (=,<,>,≤,≥), unary (e.g., isURI,) and binary (e.g., STRSTARTS) functions.

[2]http://www.w3.org/TR/sparql11-query/\#rExpression
[3]http://www.w3.org/TR/sparql11-query/\#rBuiltInCall

2.2 Extended Property Paths Semantics

The semantics (shown in Table 2) for the interpretation of an EPP expression epp on a graph $\mathcal{G}$ uses two functions: *(i)* $\mathbf{E}[\![\mathtt{epp}]\!]^{\mathcal{G}}$ defined as a binary relation $(\mathtt{u},\mathtt{v})$ such that u and v are nodes in $\mathcal{G}$ and v is reachable from u via a path in $\mathcal{G}$ satisfying epp; and *(ii)* $\mathbf{E}_{\mathcal{T}}[\![\mathtt{test}]\!]^{\mathcal{G}}_{\mathtt{t}}$ defined as a boolean function, which evaluates true if, and only if, the triple t satisfies the test test. The semantics also uses the position mapping function P_m defined as follows:

Definition 2 (**Position mapping function**). Let t=$\langle \mathtt{x},\mathtt{y},\mathtt{z}\rangle$ be a triple pattern, $\{\mathtt{x},\mathtt{y},\mathtt{z}\} \subseteq \mathcal{T} \cup \mathcal{V}$. The position mapping function $P_m(\mathtt{pos}, t)$ is defined as: *(i)* $P_m(\_s, t) = \mathtt{x}$, *(ii)* $P_m(\_p, t) = \mathtt{y}$ *(iii)* $P_m(\_o, t) = \mathtt{z}$.

P_m selects one among the subject, predicate and object of a triple on the basis of the value of pos. Consider the triple $\langle \mathtt{u}_1, \mathtt{p}_1, \mathtt{u}_2\rangle$ and the test T(_p=$\mathtt{p}_1$); this instantiates P_m as $P_m(\_p, \langle \mathtt{u}_1, \mathtt{p}_1, \mathtt{u}_2\rangle)$=$\mathtt{p}_1$, which checks $\mathtt{p}_1$=$\mathtt{p}_1$ returning true while testing T(_o=$\mathtt{u}_3$) gives false.

3 Algorithms and Complexity

This section presents two strategies for the evaluation of EPPs expressions. The first via an ad-hoc algorithm; the second one via a translation into SPARQL queries that can be executed on existing processors.

Function EVALUATE$(n, \texttt{epp}, \mathcal{G})$

Input: node n, expression $\texttt{epp}$, graph $\mathcal{G}$; **Output:** node set Res.

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

Function CLOSURE$(n, \texttt{epp}, \mathcal{G}, Res, l)$

Input: node n, EPPs expression $\texttt{epp}$, graph $\mathcal{G}$, node set Res, lower bound l;
Output: node set Res.

if $l = 1$ **then**
  $S = $ EVALUATE$(n, \texttt{epp}, \mathcal{G})$
else
  $S = \{n\}$
while $S \neq \emptyset$ **do**
  $n = extractNode(S)$ /* extract one node */
  if $n \notin Res$ **then**
    $Res = Res \cup \{n\}$
    $S = S \cup$ EVALUATE$(n, \texttt{epp}, \mathcal{G})$
return Res

Function BASE$(n, \texttt{epp}, \mathcal{G})$

Input: node n, EPPs expression $\texttt{epp}$, graph $\mathcal{G}$; **Output:** node set Res.

1: **if** $\texttt{epp} = \texttt{epp}_1|\texttt{epp}_2$ **then**
2:   **return** EVALUATE$(n, \texttt{epp}_1, \mathcal{G}) \cup$ EVALUATE$(n, \texttt{epp}_2, \mathcal{G})$
3: **if** $\texttt{epp} = \texttt{epp}_1/\texttt{epp}_2$ **then**
4:   $Res' :=$ EVALUATE$(n, \texttt{epp}_1, \mathcal{G})$
5:   $Res = \{\}$
6:   **for all** nodes $n' \in Res'$ **do**
7:     $Res = Res \cup$ EVALUATE$(n', \texttt{epp}_2, \mathcal{G})$
8:   **return** Res
9: **if** $\texttt{epp} = \texttt{epp}_1\&\texttt{epp}_2$ **then**
10:   **return** EVALUATE$(n, \texttt{epp}_1, \mathcal{G}) \cap$ EVALUATE$(n, \texttt{epp}_2, \mathcal{G})$
11: **if** $\texttt{epp} = \texttt{epp}_1 \sim \texttt{epp}_2$ **then**
12:   **return** EVALUATE$(n, \texttt{epp}_1, \mathcal{G}) \setminus$ EVALUATE$(n, \texttt{epp}_2, \mathcal{G})$
13: **if** $\texttt{epp} = \texttt{epp}_1?$ **then**
14:   **return** $\{n\} \cup$ EVALUATE$(n, \texttt{epp}_1, \mathcal{G})$
15: **if** $\texttt{epp} = \texttt{pos}_1\texttt{test}\texttt{pos}_2$ **then**
16:   $Res = \{\}$
17:   **for all** triple $t \in \mathcal{G}$ s.t. $P_m(\texttt{pos}_1, t) = n$ **do**
18:     **if** EVALTEST$(t, \texttt{test}, \mathcal{G})$ **then**
19:       $Res = Res \cup \{P_m(\texttt{pos}_1, t), P_m(\texttt{pos}_2, t)\}$
20:   **return** Res

Function EVALTEST$(t, \texttt{test}, \mathcal{G})$

Input: triple t, graph $\mathcal{G}$; **Output:** $\texttt{true}$ if t satisfy $\texttt{test}$.

if $\texttt{test} = \texttt{test}_1\&\&\texttt{test}_2$ **then**
  return EVALTEST$(t, \texttt{test}_1, \mathcal{G}) \wedge$ EVALTEST$(t, \texttt{test}_2, \mathcal{G})$
if $\texttt{test} = \texttt{test}_1||\texttt{test}_2$ **then**
  return EVALTEST$(t, \texttt{test}_1, \mathcal{G}) \vee$ EVALTEST$(t, \texttt{test}_2, \mathcal{G})$
if $\texttt{test} = !\texttt{test}_1$ **then**
  return $\neg$EVALTEST$(t, \texttt{test}_1, \mathcal{G})$
if $\texttt{test} = \texttt{u}$ **then**
  return $P_m(\_\texttt{p}, t) = \texttt{u}$
if $\texttt{test} = \texttt{TP}(\texttt{pos}, \texttt{epp})$ **then**
  return EVALUATE$(P_m(\texttt{pos}, t), \texttt{epp}, \mathcal{G}) \neq \emptyset$
if $\texttt{test} = \texttt{T}(\texttt{EExp})$ **then**
  return $\texttt{EvalSPARQLBuilt-in}(\texttt{EExp}, t)$

Figure 3: EPPs evaluation algorithm.

3.1 Recursion-based Algorithm

The algorithm for the evaluation of an EPP expression starts by invoking EVALUATE (Fig. 2), which receives as input a graph $\mathcal{G}$, an expression $\texttt{epp}$ and a node n. If $\texttt{epp}$ 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 $\texttt{test}$.

The result of the evaluation of an EPP expression $\texttt{epp}$ from a node n is a set of pairs of nodes (n,n_r) where nodes n_r are reachable from n via paths satisfying $\texttt{epp}$.

We now discuss the complexity of the algorithm. We assume $\mathcal{G}$ to be stored by its adjacency list. In particular, for each $\texttt{t} \in \mathcal{T}$, a Hashtable is maintained where the set of keys is the set of predicates $\texttt{p}$ such that there exists a triple in $\mathcal{G}$ having as subject $\texttt{t}$ and as predicate $\texttt{p}$, and the set of values are lists of objects $\texttt{o}$ reachable by traversing $\texttt{p}$-predicates from $\texttt{t}$. We assume that given $\texttt{t}$ and a predicate $\texttt{p}$ 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 $O(|\mathcal{G}|)$. Let $|\texttt{epp}|$ be the size of the EPP expression $\texttt{epp}$.

Theorem 3 Given the EPP expression $\texttt{epp}$, an RDF graph $\mathcal{G}$ and a node $n \in \mathcal{G}$ the evaluation of $\mathrm{E}[\![\texttt{epp}]\!]^{\mathcal{G}}$ can be performed in time $O(|\mathcal{G}| \cdot |\texttt{epp}|) + c_{\texttt{EExp}}$.

Proof: [SKETCH] The function EVALUATE is recursively called on each sub-expression of the $\texttt{epp}$ 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 15-19 of function BASE) require to consider at most all the edges for all the nodes; this can be done in time $O(|\mathcal{G}|)$. If $\texttt{epp}$ is recursive, the function CLOSURE is executed at most $O(nodes(\mathcal{G}))$ 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 $O(|\texttt{epp}| \cdot |\mathcal{G}|)$. As for *nested* expressions, memoization enables to mark nodes of the graph satisfying a given subexpression.

Path conjunction and negation, corresponding to intersection and difference of set of nodes respectively (line 10 and 12 of BASE), can be computed in time $O(|\mathcal{G}|)$ 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 $\texttt{EExp}$ ($c_{\texttt{EExp}}$). Finally, observe that with memoization the space complexity is $O(|\texttt{epp}| \cdot nodes(\mathcal{G})^2)$. $\square$

3.2 Translation into SPARQL

The W3C spec. (Harris and Seaborne 2013) *informally* mentions the fact that non-recursive property paths can be evaluated via a *translation* into equivalent SPARQL algebraic forms. However, no formal proof of the correctness and completeness of such translation is provided. Recursive property paths are handled in the standard via auxiliary functions called ALP (Harris and Seaborne 2013). With EPPs

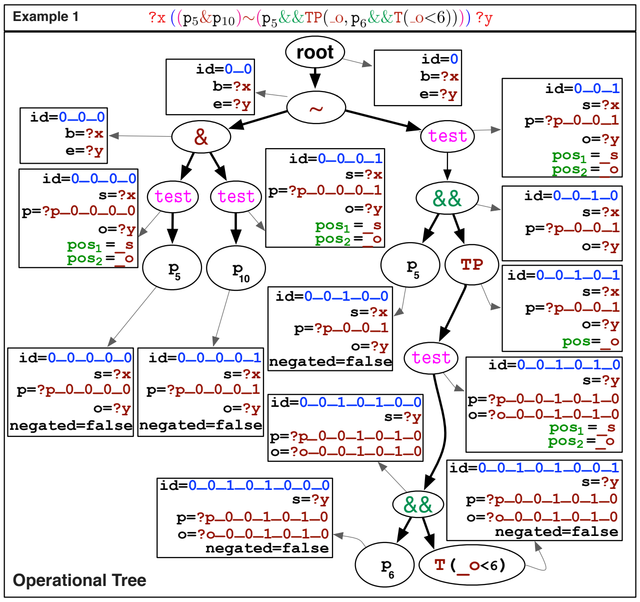

Figure 4: *Operational tree* for Ex. 1 after propagation.

(that are meant to extend property paths) we adopted a similar approach. In what follows, we give an overview of how *non-recursive* EPPs (NREPPs) are translated into SPARQL queries. For recursive EPPs we modified the ALP function defined in the W3C standard where subexpressions in recursive calls to ALP are evaluated via the SPARQL translation.

The `NREPPtoSPARQL` translation

Let $\mathcal{P}$=$(\alpha, \texttt{epp}, \beta)$, $\alpha, \beta \in \mathcal{T} \cup \mathcal{V}$ be a NREPP pattern. The translation algorithm $\mathcal{A}^t$ uses a structure called *operational tree* built from the *parse tree* of $\mathcal{P}$. The operational tree associated to $\mathcal{P}$ is an ordered, labeled, rooted tree with node attributes reflecting the operational structure of $\mathcal{P}$. We now give a high level overview of the three phases of the translation algorithm $\mathcal{A}^t$:

$\mathbf{R_m}$	$\Theta^{\mathcal{P}}(\texttt{root})$	:=‘SELECT DISTINCT’ 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	$\Theta^{\mathcal{P}}(n^{\vert})$	:=‘{’$\Theta^{\mathcal{P}}(n.child(1))$‘} 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	$\Theta^{\mathcal{P}}(n^{\texttt{test}})$	:= $\Theta^{t}(n.child(1))$
R7	$\Theta^{t}(n^{u})$	:=$\Gamma(n)$ ‘FILTER(’n.p[‘!’]‘$= u$)’[a]
R8	$\Theta^{t}(n^{\texttt{T(EExp)}})$	:=$\Gamma(n)$ ‘FILTER’[‘!’]‘(’ EExp‘)’[a]
R9	$\Theta^{t}(n^{\texttt{TP}})$	:=$\Gamma(n)$ ‘FILTER’[‘NOT’] ‘EXISTS’{$\Theta^{\mathcal{P}}(n.child(1))$‘}’[a]
R10	$\Theta^{t}(n^{\&\&})$	:=$\Theta^{t}(n.child(1))$ $\Theta^{t}(n.child(2))$
R11	$\Theta^{t}(n^{\vert\vert})$	:=‘{’$\Theta^{t}(n.child(1))$‘} UNION{’$\Theta^{t}(n.child(2))$‘}’

Figure 5: Translation rules. [a]If the node in input has the attribute `negated`=true use ‘!’ (resp., ‘NOT’).

(i) Building of the operational tree

Consider a NREPP pattern $\mathcal{P}$. In the operational tree associated to $\mathcal{P}$, nodes are of two types: *(i) operational nodes* (labeled with $\sim, \&, /, |, \hat{}$) and *(ii) test nodes* (labeled with `test`[4], $||$, $\&\&$, `TP`, u, `T`(`EExp`)). Each node n_i in the operational tree has a single parent and its children are an ordered set. Moreover, n_i has an attribute `id` whose unique value is computed as the *concatenation* of the parent's `id` and n_i's position in the ordered set of children. The operational tree for Example 1 is shown in Fig. 4.

(ii) Propagation of variables and terms[5]

The key point is the propagation of variable names and RDF terms (kept in nodes' attributes). This is done by traversing the operational tree top-down and propagating attributes from each parent node to its children.

(iii) Application of the translation rules

This phase always starts by applying rule $\mathbf{R_m}$ in Fig. 5 on the `root` of the operational tree. This generates the outermost part of the final rewritten query into the SPARQL syntax. The translation proceeds by applying rules at each node of the operational tree visited according to a pre-order depth-first traversal.

For instance, in Fig. 4, after the `root`, the node with `id`=0_0 and labeled with $\sim$ is visited. This causes the triggering of rule R5, which enables to generate another (internal) chunk of the final SPARQL query. Node 0_0 is an operational node representing path negation $\sim$. As it can be noted, our translation procedure uses the SPARQL `MINUS` operator to reflect the semantics of EPPs dealing with path negation. The semantics of nodes representing EPPs `test` (e.g., 0_0_1_0_1) is reflected into SPARQL via the `FILTER` operator.

Correctness of the translation

In order to prove the correctness of the translation of NREPPs into SPARQL we have defined a SPARQL based semantics (not reported here for sake of space) where conjunction (&) and negation ($\sim$) are translated into join ($\bowtie$) and difference ($\setminus$) of (multi)sets of solution mappings (Pérez, Arenas, and Gutierrez 2009). Let $[\![\mathcal{P}]\!]^{\mathcal{G}}_{epps}$ denote the SPARQL-based semantics of EPPs, where $\mathcal{P}$ is an NREPP pattern, and $[\![\mathcal{S}]\!]^{W3C}_{\mathcal{G}}$ denote the SPARQL W3C semantics (Harris and Seaborne 2013), where $\mathcal{S}$ is a SPARQL query. The following theorem shows the correctness of the translation.

Theorem 4 The translation algorithm $\mathcal{A}^t$ is correct and runs in polynomial time in the size of the expression to be translated. Moreover, for any RDF graph $\mathcal{G}$ given $\mathcal{P}$ it holds that $[\![\mathcal{P}]\!]^{epps}_{\mathcal{G}}$=$[\![\mathcal{S}]\!]^{W3C}_{\mathcal{G}}$, where $\mathcal{S}$=$\mathcal{A}^t(\mathcal{P})$ is the query produced by translating $\mathcal{P}$.

Proof: [SKETCH] $\mathcal{A}^t$ is polynomial as it requires one scan of the operational tree (whose size is polynomial in the size of the EPP expression). As for the correctness, it is enough to prove that the propagation of variable names and RDF terms in the operational tree is correct. This proof can be

[4]In the syntax it coresponds to $\texttt{pos}_1$ `test` $\texttt{pos}_2$.

[5]Negation (!) for test nodes is propagated top-down when building the tree via the attribute `negated`.

done by structural induction on the depth of the operational tree. Finally, the semantic equivalence can be proved by associating to all types of EPP patterns the corresponding semantics as per SPARQL specification (Harris and Seaborne 2013). □

Complexity of the translated queries. Consider a NREPP `epp` in a pattern $\mathcal{P}$ and its translation into SPARQL $\mathcal{S}_{\texttt{epp}}$. Clearly, the complexity of evaluating $\mathcal{S}_{\texttt{epp}}$ depends on the fragment of SPARQL used in the translation. In particular (see Fig. 5) we make usage of `SELECT`, `UNION`, `MINUS` and `FILTER` and we *do not* use `OPTIONAL`. The complexity of this and other SPARQL fragments has been studied in (Pérez, Arenas, and Gutierrez 2009).

Summary. EPPs can be seen as a language per se and can be evaluated by implementing the algorithm in Fig. 3 in a custom processor, thus being independent from existing SPARQL processors. EPPs can also be seen as a conservative extension of SPARQL property paths and, thanks to the translation algorithm, can be evaluated on existing SPARQL processors. We have implemented both strategies; their pros and cons will be discussed in Section 4.

3.3 Comparison with Related Languages

As the goal of EPPs is to extend the expressive power of SPARQL property paths (PPs) and NREs-based languages, we compared such proposals with EPPs; Table 3 summarizes the (informal) comparison. We consider the following features of EPPs: path conjunction (&), path negation (~), nesting (TP), tests over node values (`test`) and usage of positions (`pos`). As it can be observed, PPs are the least expressive language; they do not support any of the new features of EPPs. This motivated our choice of *extending* PPs as described in Section 2.1. As for NREs, they clearly support nesting but neither other types of tests (e.g., node equality) nor path conjunction/negation as discussed in (Barceló, Pérez, and Reutter 2012).

Language	&	~	TP	`test`	`pos`
EPPs	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
PPs	No	No	No	No	No
NREs	No	No	**No**	No	No
nSPARQL	**Yes**	**Yes**	**Yes**	No	**Yes**

Table 3: Comparison of EPPs with related languages.

As for nSPARQL (based on NREs), it supports path conjunction and negation only via the SPARQL algebra; it also supports nesting and positions. However, it does not allow to test (in)equalities of nodes reached with a nested expression. EPPs support logical combination of tests representing nesting and tests representing (in)equalities as well as *safe-negation*[6]. Note that neither PPs nor NREs nor nSPARQL can express Example 1. nSPARQL supports positions by transforming an RDF graph into another graph where *navigational axes* (similar to those defined in XPath) are made explicit. On the other hand, EPPs do not require any transformation. In the syntax of EPPs, positions enable to perform *edge to node* traversals allowing to reach the node representing the predicate of a triple whence it is possible to traverse e.g. the property hierarchy. Finally, we want to emphasize that EPPs are syntactically compatible with PPs and are the only language, which thanks to the `NREPPtoSPARQL` translation, can be used in existing SPARQL processors.

[6]The first element must be positive path.

4 Implementation and Evaluation

We have implemented both a custom query processor for EPPs and the `NREPPtoSPARQL` translation[7].

Dataset and query set. We used a crawl of the FOAF social network (~500MBs) obtained from the BTC2012[8] by traversing from the URI of T. Berners-Lee (TBL) `foaf:knows` predicates up to distance 4. We call the resulting graph $\mathcal{G}^F$, which has ~4M triples. We created 4 groups $G_i, i \in \{1,..., 4\}$ of EPP expressions each with 3 queries; this gives a queryset Q of 12 queries. The experiments have been performed on an Intel i5 machine with 8GBs RAM. Results are the average of 5 runs after removing lowest and highest values.

Experiment 1: Running time
For each `epp` $\in Q$ we generated the corresponding SPARQL query $\mathcal{S}_{\texttt{epp}}$ via the algorithm described in Section 3.2. We measured the execution time for each `epp` $\in Q$ with *our processor* and the execution time for $\mathcal{S}_{\texttt{epp}}$ in *Jena ARQ*[9]. Fig. 6 shows the running times. The number of results ranges from ~50 to ~8000.

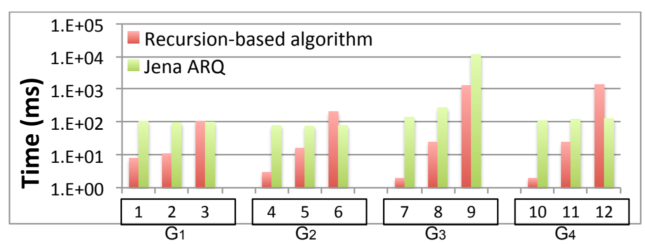

Figure 6: EPPs-custom-processor vs. Jena ARQ.

As it can be observed, for G_1, which contains queries asking for friends of TBL up to distance 3, our evaluator performs better than ARQ at distance 1 and 2; at distance 3 times are comparable. The group G_2 additionally considers a test based on *nesting*. Again, our evaluator performs better at distance 1 and 2; at distance 3 it shows a higher running time. In G_3, which considers *negation* (e.g., *exclusive friends* at various distances) our approach performs consistently better. Finally, in G_4 that includes conjunction (to ask for *mutual* friends at various distances) our approach performs better at distance 1 and 2 and obtains a higher running time at distance 3.

[7]Available at http://extendedpps.wordpress.com
[8]http://km.aifb.kit.edu/projects/btc-2012
[9]http://jena.apache.org

The complexity of the algorithm described in Section 3 is lower than that of Jena ARQ, which implements SPARQL. However, our evaluation suggests that for real-world data and natural queries (e.g., mutual friends) working with the translation of EPPs and using existing processors is still useful. Hence, the advantage of our approach is that navigational queries can be written in a succinct way via the EPPs syntax and execute after their translation via `NREPPtoSPARQL`. Anecdotally, while the EPP asking for mutual friends at distance 3 contains ∼200 characters, the SPARQL query (obtained from the translation) contains ∼700 characters. Moreover, when writing the SPARQL query one has also to deal with a large number of variables that need to be consistently joined.

Experiment 2: Translation overhead
We compared the elapsed times of our `NREPPtoSPARQL` with the `SPARQLtoALGEBRA` translation of Jena ARQ. We used 28 queries generated in two steps. We started with 4 base expressions plus a fifth one combining them. Second, starting from them we generated increasingly longer expressions. We found that our `NREPPtoSPARQL` translation performs comparably (in the order of ms) to the existing `SPARQLtoALGEBRA` translation.

5 Conclusions and Future Work

We introduced Extended Property Paths (EPPs), a significant extension of SPARQL property paths. We have provided an algorithm for their evaluation and also a translation procedure into SPARQL.

EPPs bring some benefits: *(i)* users can leverage their syntax to write more expressive *navigational* queries in a succinct way; *(ii)* EPPs are immediately available into existing SPARQL processors. Our study opens some research questions such as investigating properties of the translation (e.g., minimality) and studying optimization techniques. Also containment of EPPs is an open question.

References

Alkhateeb, F.; Baget, J.-F.; and Euzenat, J. 2009. Extending SPARQL with Regular Expression Patterns (for querying RDF). *J. Web Semantics* 7(2):57–73.

Arenas, M.; Gottlob, G.; and Pieris, A. 2014. Expressive Languages for Querying the Semantic Web. In *Proc. of the 33rd symposium on Principles of Database Systems*, 14–26. ACM.

Barceló, P.; Libkin, L.; Lin, A. W.; and Wood, P. T. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Transactions on Database Systems* 37(4):31.

Barceló, P.; Pérez, J.; and Reutter, J. L. 2012. Relative Expressiveness of Nested Regular Expressions. In *Proc. of the 6th Alberto Mendelzon International Workshop on Fundations of Data Management*, 180–195.

Berglund, A.; Boag, S.; Chamberlin, D.; Fernández, M.; Kay, M.; Robie, J.; and Siméon, J. 2010. XML Path Language (XPath) 2.0, W3C.

Cali, A.; Gottlob, G.; Lukasiewicz, T.; Marnette, B.; and Pieris, A. 2010. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *Proc. of the 25th Annual IEEE Symposium on Logic in Computer Science*, 228–242. IEEE.

Fionda, V., and Pirrò, G. 2013. Querying Graphs with Preferences. In *Proc.of the 22nd ACM international conference on Conference on Information & Knowledge Management*, 929–938. ACM.

Fionda, V.; Gutierrez, C.; and Pirrò, G. 2012. Semantic Navigation on the Web of Data: Specification of Routes, Web Fragments and Actions. In *Proc. of the 21st International Conference on World Wide Web*, 281–290. ACM.

Fletcher, G. H.; Gyssens, M.; Leinders, D.; Van den Bussche, J.; Van Gucht, D.; Vansummeren, S.; and Wu, Y. 2011. Relative Expressive Power of Navigational Querying on Graphs. In *Proc. of the 14th International Conference on Database Theory*, 197–207. ACM.

Harris, S., and Seaborne, A. 2013. SPARQL 1.1 Query Language, W3C.

Kochut, K., and Janik, M. 2007. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *Proc. of the 4th European Semantic Web Conf.*, 145–159.

Libkin, L.; Reutter, J.; and Vrgoč, D. 2013. TriAL for RDF: Adapting Graph Query Languages for RDF Data. In *Proc. of the 32nd Symposium on Principles of Database Systems*, 201–212. ACM.

Pérez, J.; Arenas, M.; and Gutierrez, C. 2009. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems* 34(3).

Pérez, J.; Arenas, M.; and Gutierrez, C. 2010. nSPARQL: A Navigational Language for RDF. *J. Web Semantics* 8(4):255–270.

Rudolph, S., and Krötzsch, M. 2013. Flag & check: Data Access with Monadically Defined Queries. In *Proc. of the 32nd Symposium on Principles of Database Systems*, 151–162. ACM.

Wood, P. T. 2012. Query Languages for Graph Databases. *SIGMOD Rec.* 41(1).