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

Valeria Fionda\textsuperscript{1}, Giuseppe Pirr\`o\textsuperscript{2} and Mariano P. Consens\textsuperscript{3}
\textsuperscript{1} DeMaCS, University of Calabria, Italy
\textsuperscript{2} WeST, University of Koblenz-Landau, Germany
\textsuperscript{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 (Alkhatceeb, Baget, and Euzenat 2009; Fionda and Pirr\`o 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; Alkhatceeb, 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 Pirr\`o 2012). Languages like TriAL (Libkin, Reutter, and Vrgoč 2013), TriIQ (Arenas, Gottlob, and Pieris 2014) and NEMODEQ (Rudolph and Krötzsch 2013) are grounded on (extensions of) Datalog. Other expressive languages like SPARQLεR (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 TriIQ, TriAL and NEMODEQ in terms of Datalog\textsuperscript{±} 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 in Section 2.

Example 1 (Extended Property Paths). By considering the graph in Fig. 1, find pairs of nodes\(^1\) (\(n_0, n_j\)) 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:

\[
?x \ (p_5 \& p_{10}) \sim (p_5 \& \& TP(\omega, p_6 \& \& T(\omega < 6))) \ ?y
\]

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 \(?x \rightarrow n_0\). From \(n_0\), the evaluation of the first path \((p_5 \& p_{10})\) enables to reach \(n_4\). 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 \(\omega\) means that the test starts from the object of the previous navigational step, that is, the object of the triple \((n_0, p_5, n_3)\). From \(n_3\), another logical AND of two tests is evaluated. The first one checks the existence of an edge \(p_5\) 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 \(n_4\) 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 \(?x \rightarrow n_0\).

Overall, for \(?x \rightarrow n_0\) we get \(?y \rightarrow n_4\) meaning that from \(n_0\) the navigation of the graph according to the expression enabled to reach \(n_4\).

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, \(?o_0.0_0.0_1.0_1.0\) 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. } FILTER EXISTS
  { ?x p10 ?y. } 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 \(V\) (variables, starting with the ‘?’ symbol) be four pairwise disjoint, countably infinite sets. An RDF triple is a tuple of the form \((s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)\). An RDF graph \(G\) is a set of triples. We indicate by \(\text{nodes}(G) \subseteq U \cup B \cup L\) the set of URIs, blank nodes and literals that appear as subject or object of some triples in \(G\). Moreover, \(T \subseteq U \cup B \cup L\) is the set of terms of \(G\).
\[
\begin{align*}
epp & ::= \text{'}epp\text{'} | \text{'}epp\text{'+'} | \text{'}epp\text{'}| | \text{'}epp\text{'}* | \text{'}epp\text{'}/* | \text{'}epp\text{'}| | \text{'}epp\text{'}* | \text{'}epp\text{'}| | \text{'}epp\text{'}| |
\end{align*}
\]

\[
\begin{align*}
\text{pos} & ::= \text{'s'} | \text{'p'} | \text{'o'}
\end{align*}
\]

Table 1: Syntax of EPPs expressions. 1If omitted is \text{'s'}; 2If omitted is \text{'o'}.

\[
\begin{align*}
\text{R1} & \rightarrow \text{E}[^{\text{epp}}]^{\text{G}} := \{ (u,v) : (v,u) \in \text{E}[^{\text{epp}}]^{G}\} \\
\text{R2} & \rightarrow \text{E}[^{\text{epp}/^\text{epp}}]^{\text{G}} := \{ (u,v) : \exists w : (u,w) \in \text{E}[^{\text{epp}}]^{G} \land (w,v) \in \text{E}[^{\text{epp}}]^{G}\} \\
\text{R3} & \rightarrow \text{E}[^{\text{epp}/^\text{'epp}}]^{\text{G}} := \{ (u,v) : u \in \text{nodes}(G) \} \cup \bigcup_{i=1}^{\infty} \text{E}[^{\text{epp}}]^{G} | \text{epp} = \text{epp} \lor \text{epp} = \text{epp}/_{1}\text{epp} \\
\text{R4} & \rightarrow \text{E}[^{\text{epp}/^\text{'}epp}]^{\text{G}} := \bigcup_{i=1}^{\infty} \text{E}[^{\text{epp}}]^{G} | \text{epp} = \text{epp} \land \text{epp} = \text{epp}/_{1}\text{epp} \\
\text{R5} & \rightarrow \text{E}[^{\text{epp}/^\text{'}epp}]^{\text{G}} := \{ (u,v) : u \in \text{nodes}(G) \} \cup \text{E}[^{\text{epp}}]^{G} \\
\text{R6} & \rightarrow \text{E}[^{\text{epp/}^\text{epp2}}]^{\text{G}} := \{ (u,v) : (v,u) \in \text{E}[^{\text{epp}}]^{G} \} \lor \{ (u,v) : (u,v) \in \text{E}[^{\text{epp2}}]^{G} \} \\
\text{R7} & \rightarrow \text{E}[^{\text{epp/}^\text{'}epp2}]^{\text{G}} := \{ (u,v) : (v,u) \in \text{E}[^{\text{epp}}]^{G} \} \land \{ (u,v) : (u,v) \in \text{E}[^{\text{epp2}}]^{G} \} \\
\text{R8} & \rightarrow \text{E}[^{\text{epp}/^\text{'}epp2}]^{\text{G}} := \{ (u,v) : (v,u) \in \text{E}[^{\text{epp}}]^{G} \} \land \{ (u,v) \notin \text{E}[^{\text{epp2}}]^{G} \} \\
\text{R9} & \rightarrow \text{E}[^{\text{posl} \lor \text{pos2}]}^{\text{G}} := \{ (P_{m}(pos_{1}, t), P_{m}(pos_{2}, t)) \} | \text{triple } t \in \text{G} \land \text{E}_{T}[^{\text{test}}]^{G} \\
\end{align*}
\]

Table 2: Formal semantics of EPPs expressions.

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 '!(u1 |...|un)' that are 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 (\text{epp1} \& \text{epp2}), path negation (\text{epp1} \sim \text{epp2}) and different types of tests (\text{test} within) a path, also by specifying the starting and ending positions (\text{pos}). EPPs enable to test from each of the subject, predicate and object positions in RDF triples, mapped in the syntax to \text{'s'}, \text{'p'} and \text{'o'}, respectively. Positions do not need to be always specified; by default a test starts from the subject (\text{'s'}) and ends on the object (\text{'o'}) of a triple.

Tests (\text{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., \text{TP}(\text{pos}, \text{epp}), which corresponds to the evaluation of \text{epp} starting from a position \text{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 \text{epp}. A base test (production \text{base}) can be of type \text{T}, which is a SPARQL boolean expression; here, \text{EExp} (not reported here for sake of space) extends the production [110] in the SPARQL grammar where BuiltInCall is substituted with BuiltInCall, which enables to use in EPPs tests available in SPARQL as built-in conditions also augmented with positions (\text{pos}). Built-in conditions are constructed using elements of the set \text{U}L and constants, logical connectives (\sim, \&\&, \lor), (in)equality symbol(s) (=, <, >, \leq, \geq), unary (e.g., \text{isURI}) and binary (e.g., \text{STRSTARTS}) functions.

2.2 Extended Property Paths Semantics

Wendig's (shown in Table 2) for the interpretation of an EPP expression \text{epp} on a graph \text{G} uses two functions: (i) \text{E}[^{\text{epp}}]^{G} defined as a binary relation (u,v) such that u and v are nodes in \text{G} and v is reachable from u via a path in \text{G} satisfying \text{epp}; and (ii) \text{E}_{T}[^{\text{test}}]^{G} defined as a boolean function, which evaluates true if, and only if, the triple \text{t} satisfies the test \text{test}. The semantics also uses the position mapping function \text{P}_{m} defined as follows:

Definition 2 (Position mapping function). Let \text{t}=(\text{x,y,z}) be a triple pattern, \{\text{x,y,z}\} \subseteq \text{T} \cup \text{V}. The position mapping function \text{P}_{m}(\text{pos}, \text{t}) is defined as: (i) \text{P}_{m}(\text{pos}, \text{t}) = \text{x}, (ii) \text{P}_{m}(\text{pos}, \text{t}) = \text{y} (iii) \text{P}_{m}(\text{pos}, \text{t}) = \text{z}.

\text{P}_{m} selects one among the subject, predicate and object of a triple on the basis of the value of \text{pos}. Consider the triple (\text{u1}, \text{p1}, \text{u2}) and the test \text{T}_{(p1=p1)}; this instantiates \text{P}_{m} as \text{P}_{m}(\text{pos}, (\text{u1}, \text{p1}, \text{u2}))=\text{p1}, which checks \text{p1}=\text{p1} returning true while testing \text{T}_{(\text{u1}=\text{u3})} gives false.

\text{2 http://www.w3.org/TR/sparql11-query/#Expression}
\text{3 http://www.w3.org/TR/sparql11-query/#BuiltInCall}
3.1 Recursion-based Algorithm

The algorithm for the evaluation of an EPP expression starts by invoking \texttt{EVALUATE} (Fig. 2), which receives as input a graph \(G\), an expression \(\text{epp}\) and a node \(n\). If \(\text{epp}\) is non recursive (i.e., it does not contain the closure operators \('^*\' and \('^+\)'), then it is given as input to the function \texttt{BASE}, which considers the various forms of syntactic expressions. For recursive expressions the algorithm uses the function \texttt{CLOSURE}. Finally, the boolean function \texttt{EVALTEST} handles the different types of \texttt{test}.

The result of the evaluation of an EPP expression \(\text{epp}\) 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 \(G\) to be stored by its adjacency list. In particular, for each \(t \in \mathcal{T}\), 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 \(t\) and as predicate \(p\), and the set of values is lists of objects \(o\) reachable by traversing \(p\)-predicates from \(t\). We assume that given \(t\) 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 \(|\text{epp}|\) be the size of the EPP expression \text{epp}.

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

**Proof:** [Sketch] The function \texttt{EVALUATE} is recursively called on each sub-expression of the \text{epp} in input; if such sub-expressions are not recursive (i.e., do not contain \('^*\', \'^+\)'), \texttt{EVALUATE} is invoked at most \(O(|\text{epp}|)\) times. The base cases (lines 15-19 of function \texttt{BASE}) require to consider at most all the edges for all the nodes; this can be done in time \(O(|G|)\). If \text{epp} is recursive, the function \texttt{CLOSURE} is executed at most \(O(nodess(G))\) times; the procedure \texttt{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 \texttt{CLOSURE} is \(O(|\text{epp}| \cdot |G|)\). 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 \texttt{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 \textit{logical operators} and simple URI checking. The complexity is parametric wrt the cost of other SPARQL-based built-in conditions \texttt{EExp} (\(c_{\text{EExp}}\)). Finally, observe that with memoization the space complexity is \(O(|\text{epp}| \cdot \text{nodes}(G)^2)\).
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 (that are meant to extend 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 $R_m$ 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 ∼ is visited. This causes the triggering of rule $R_5$, which enables to generate another (internal) chunk of the final SPARQL query. Node $0.0$ is an operational node representing path negation ∼.

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 (¬) are translated into join (∘) and difference (∖) of (multi)sets of solution mappings (Pérez, Arenas, and Gutiérrez 2009). Let $\{\mathcal{P}\}_{\epsilon e p p s}$ denote the SPARQL-based semantics of EPPs, where $\mathcal{P}$ is an NREPP pattern, and $\{\mathcal{S}\}_{\epsilon W 3 C}$ 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.

\[ \mathcal{P}_{\epsilon e p p s} \mathbf{=} \{ \mathcal{S} \}_{\epsilon W 3 C} \]

\[ \mathcal{P}_{\epsilon e p p s} \mathbf{=} \{ \mathcal{S} \}_{\epsilon W 3 C} \]

\[ \mathcal{P}_{\epsilon e p p s} \mathbf{=} \{ \mathcal{S} \}_{\epsilon W 3 C} \]
Theorem 4 The translation algorithm $A^t$ is correct and runs in polynomial time in the size of the expression to be translated. Moreover, for any RDF graph $G$ given $\mathcal{P}$ it holds that $[P]^{epp}_{G} = [S]^{\mathbb{E}}_{G}$, where $S = A^t(\mathcal{P})$ is the query produced by translating $\mathcal{P}$.

Proof: [SKETCH] $A^t$ 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 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 $S_{epp}$. Clearly, the complexity of evaluating $S_{epp}$ 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 ($\sim$), 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).

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. 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 XPaths) 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.

4 Implementation and Evaluation

We have implemented both a custom query processor for EPPs and the NREPPtoSPARQL translation.

Dataset and query set. We used a crawl of the FOAF social network (~500MBs) obtained from the BTC2012 by traversing from the URI of T. Berners-Lee (TBL) foaf:knows predicates up to distance 4. We call the resulting graph $G^F$, 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 $S_{epp}$ 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 $S_{epp}$ in Jena ARQ. Fig. 6 shows the running times. The number of results ranges from ~50 to ~8000.

![Figure 6: EPPs-custom-processor vs. Jena ARQ.](http://extendedpps.wordpress.com)

6 The first element must be positive path.

7 Available at [http://extendedpps.wordpress.com](http://extendedpps.wordpress.com)

8 [http://km.aifb.kit.edu/projects/btc-2012](http://km.aifb.kit.edu/projects/btc-2012)

9 [http://jena.apache.org](http://jena.apache.org)
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.

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 $\sim 200$ characters, the SPARQL query (obtained from the translation) contains $\sim 700$ 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**


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


