Recursion in SPARQL

Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč

PUC Chile and Center for Semantic Web Research
jreutter@ing.puc.cl, assoto@uc.cl, dvrgoc@ing.puc.cl

Abstract. In this paper we propose a general purpose recursion operator to be added to SPARQL, formalize its syntax and develop algorithms for evaluating it in practical scenarios. We also show how to implement recursion as a plug-in on top of existing systems and test its performance on several real world datasets.

1 Introduction

The Resource Description Framework (RDF) has emerged as the standard for describing Semantic Web data and SPARQL as the main language for querying RDF. After the initial proposal of SPARQL, and with more data becoming available in the RDF format, users found use cases that required exploring the structure of the data in more detail. In particular queries that are inherently recursive, such as traversing paths of arbitrary length, have lately been in demand. This was acknowledged by the W3C committee with the inclusion of property paths in the latest SPARQL 1.1. standard [12], allowing queries to navigate paths connecting two objects in an RDF graph.

However, in terms of expressive power, several authors have noted that property paths fall short when trying to express a number of important properties related to navigating RDF documents (cf. [6, 7, 22]), and that a more powerful form of recursion needs to be added to SPARQL to address this issue. As a result various extensions of property paths have been proposed (see e.g. [4, 14, 17, 22]), but to the best of our knowledge no attempt to add a general recursion operator to the language has been made.

To illustrate the need for such an operator we consider the case of tracking provenance of Wikipedia articles presented by Missier and Chen in [19]. They use the PROV standard [24] to store information about how a certain article was edited, whom was it edited by and what this change resulted in. Although they store the data in a graph database, all PROV data is easily representable as RDF using the PROV-O ontology [27]. The most common type of information in this RDF graph tells us when an article A_1 is a revision of an article A_2. This fact is represented by adding a triple of the form (A_1, `prov:wasRevisionOf`, A_2) to the database. These revisions are associated to user's edits with the predicate `prov:wasGeneratedBy` and the edits can specify that they used a particular article with a `prov:used` link. Finally, there is a triple (E, `prov:wasAssociatedWith`, U) if the edit E was made by the user U. A snapshot of the data, showing provenance of articles about Edinburgh, is depicted in Figure 1.

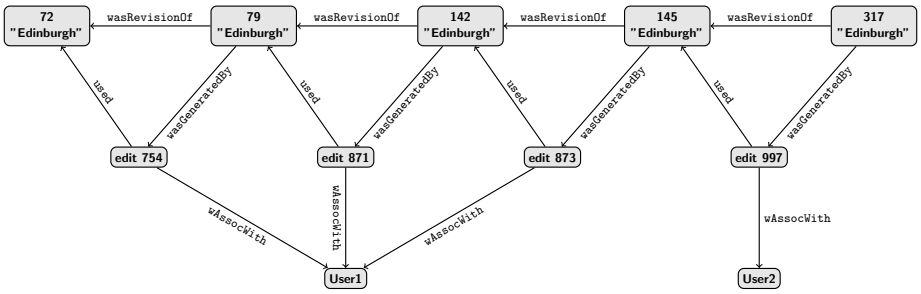


Fig. 1. RDF database of Wikipedia traces. The abbreviation `wAssocWith` is used instead of `wasAssociatedWith` and the `prov:`prefix is omitted from all the properties.

A natural query to ask in this context is the history of revisions that were made by the same user: that is all pairs of articles (A, A') such that A is linked to A' by a path of `wasRevisionOf` links and where all of the revisions along the way were made by the same user. For instance, in Figure 1 we have that the article 145 "Edinburgh" is a revision of the article 72 "Edinburgh" and all the intermediate edits were made by User1. Such queries abound in version control systems (for instance when tracking program development in svn or Git) and can be used to detect which user introduced errors or bugs, when the data is reliable, or to find the latest stable version of the data. Since these queries can not be expressed with property paths [6, 17], nor by using standard SPARQL functionalities (as provenance traces can contain links of arbitrary length), a general purpose recursion operator seems like a natural addition to the language.

One reason why recursion in SPARQL was not considered previously could be the fact that in order to compute recursive queries we need to apply the query to the result of a previous computation. However, typical SPARQL queries do not have this capability as their inputs are RDF graphs but their outputs are mappings. This hinders the possibility of a fixed point recursion as the result of a SPARQL query cannot be subsequently queried. One can avoid this by using CONSTRUCT queries, which output RDF graphs, and indeed [15] has proposed a way of defining a fixed point like extension for SPARQL based on this idea.

In this paper we extend the recursion operator of [15] to function over a more widely used fragment of SPARQL and study how this operator can be implemented in an efficient way on top of existing SPARQL engines. We begin by showing what the general form of recursion looks like and how to evaluate it. After arguing why full fledged recursion is unlikely to perform well on real world data, we consider a restriction called *linear recursion*, which is widely used in the relational context [1, 10], and show that it can express almost any use case found in practice. Next, we develop an elegant algorithm for evaluating this class of recursive queries and show how it can be implemented on top of an existing SPARQL system. For our implementation we use Apache Jena framework [13] and we implement recursive queries as an add-on to the ARQ SPARQL query

engine. We use Jena TDB version 2.12.1, which allows us not to worry about queries whose intermediate results do not fit into main memory, thus resulting in a highly reliable system. Lastly, we test how this implementation performs on YAGO, LMBD and PROV records of Wikipedia revision history.[1]

Related work. The most common recursive functionality available for SPARQL are property paths. These are either implemented fully [11, 13], or with some limitations that ensure they can be efficiently evaluated [26]. Several extensions of property paths have also been considered by the research community [3, 4, 14, 22] and although some of them can simulate certain recursive tasks, they still fail to express arbitrary recursive queries. There were also some attempts to allow recursion as a programming language construct [5,20], however they do not view recursion as a part of the language, but as an outside add-on. Regarding attempts to implement a full-fledged recursion as a part of SPARQL, both [25] and [15] propose a syntax of the recursion operator similar to the one used here, however, neither of the two describes specific algorithms for its execution, nor do they analyse its performance, but instead focus on expressive power.

2 Preliminaries

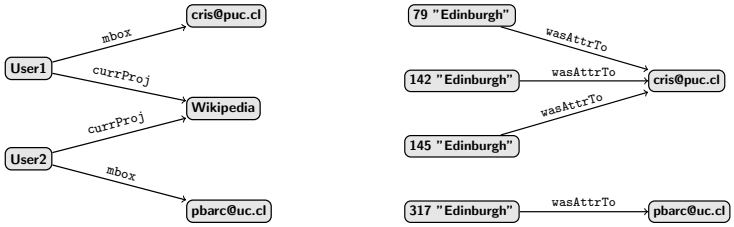
RDF Graphs and Datasets. RDF graphs can be seen as edge-labeled graphs where edge labels can be nodes themselves, and an RDF dataset is a collection of RDF graphs. Formally, let $\mathbf{I}$ be an infinite set of *IRIs*[2]. An *RDF triple* is a tuple (s, p, o) from $\mathbf{I} \times \mathbf{I} \times \mathbf{I}$, where s is called the *subject*, p the *predicate*, and o the *object*. An *RDF graph* is a finite set of RDF triples, and an *RDF dataset* is a set $\{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$, where $G_0, \ldots, G_n$ are RDF graphs and $u_1, \ldots, u_n$ are distinct IRIs. The graph G_0 is called the *default graph*, and $G_1, \ldots, G_n$ are called *named graphs* with *names* $u_1, \ldots, u_n$, respectively. For a dataset D and IRI u we define $\mathsf{gr}_D(u) = G$ if $\langle u, G\rangle \in D$ and $\mathsf{gr}_D(u) = \emptyset$ otherwise. Given two datasets D and D' with default graphs G_0 and G'_0, we define the union $D \cup D'$ as the dataset with the default graph $G_0 \cup G'_0$ and $\mathsf{gr}_{D\cup D'}(u) = \mathsf{gr}_D(u) \cup \mathsf{gr}_{D'}(u)$ for any IRI u. Union of datasets without default graphs is defined in the same way, i.e., as if the default graph was empty.

SPARQL Syntax. We assume the familiarity with syntax and semantics of SPARQL 1.1 query language. However, we do recall two particular features that will be used: the GRAPH operator and the CONSTRUCT result form.

We assume all variables come from an infinite set $\mathbf{V} = \{?x, ?y, \ldots\}$ of *variables*. The official syntax for SPARQL 1.1 queries considers several operators such as OPTIONAL, UNION, FILTER, GRAPH and concatenation via the point symbol (.) to construct what is known as *graph patterns*. Users then use a result form such as SELECT or CONSTRUCT to form either result sets or RDF graphs from the matchings of a graph pattern. We assume that readers are familiar

[1] The implementation, test data and complete formulation of used queries can be found in the online appendix available at http://web.ing.puc.cl/~jreutter/Recsparql.html.

[2] For clarity of presentation we do not include literals or blank nodes in our definitions.



(a) Graph G_1 (b) $\mathsf{ans}(q, D)$ in Example 1

Fig. 2. Graphs used for Example 1. The prefixes `foaf:` and `prov:` are omitted.

with graph patterns, we just note the syntax of the GRAPH operator: if P is a graph pattern and $g \in \mathbf{I} \cup \mathbf{V}$ then (GRAPH g P) is a graph pattern, called a GRAPH-*pattern*. The expression (GRAPH g P) allows us to determine which graph from the dataset we will be matching the pattern P to. For instance if we use an IRI in place of g the pattern will be matched against the named graph with the corresponding name (if such a graph exists in the dataset), and in the case that g is a variable, P will be matched against all the graphs in the dataset.

Although SELECT queries over graph patterns seem to be the most popular use of SPARQL, as the results of such queries are not RDF graphs, we will use the CONSTRUCT operator as a base for recursion. A SPARQL CONSTRUCT query, or *c-query* for short, is an expression

$$\mathsf{CONSTRUCT}\ H\ DS\ \mathsf{WHERE}\ P,$$

where H is a set of triples from $(\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V})$, called a *template*; DS is a set of expressions of the form $\mathsf{FROM\ NAMED}\ u_1, \ldots, \mathsf{FROM\ NAMED}\ u_n$, with each $u_i \in \mathbf{I}$ and $i \geq 0$, called a *dataset clause*[3]; and P is a graph pattern.

The idea behind the CONSTRUCT operator is that the mappings matched to the pattern P are used to construct an RDF graph according to the template H. Since all the patterns in the template are triples we will end up with an RDF graph as desired.

Example 1. Let G and G_1 be the graphs in Figure 1 and Figure 2(a), respectively. We want to query both graphs to obtain a new graph where each article is linked to the email of a user who modified it. Assuming we have a dataset with default graph G and that the IRI identifying G_1 is `http://db.ing.puc.cl/mail`, this would be achieved by the following SPARQL CONSTRUCT query q:

```
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX prov: <http://xmlns.com/foaf/0.1>
CONSTRUCT {?article prov:wasAttributedTo ?mail}
FROM NAMED <http://db.ing.puc.cl/mail>
WHERE {
   ?article prov:wasGeneratedBy ?comment .
   ?comment prov:wasAssociatedWith ?usr .
   GRAPH <http://db.ing.puc.cl/mail> {?usr foaf:mbox ?mail}}
```

[3] For readability we assume the default graph as given.

The result $\mathsf{ans}(\boldsymbol{q}, D)$ of evaluating q over D is depicted in Figure 2(b). The construct FROM NAMED is used to specify that the dataset needs to include the graph G_1 associated with the IRI `http://db.ing.puc.cl/mail`.

SPARQL Semantics. The semantics of graph patterns is defined in terms of *mappings* [12]; that is, partial functions from variables $\mathbf{V}$ to IRIs $\mathbf{I}$. Given a dataset D, and a graph G amongst the graphs of D, we denote the *evaluation* of a graph pattern P over D with respect to G as $[\![P]\!]^D_G$. The evaluation $[\![P]\!]^D$ of a pattern P over a dataset D with default graph G_0 is $[\![P]\!]^D_{G_0}$. The full definition of $[\![P]\!]^D$ and $[\![P]\!]^D_G$ can be found in the SPARQL standard, here we just note the semantics of GRAPH-patterns, for which we need some notation.

The *domain* $\mathsf{dom}(\mu)$ of a mapping μ is the set of variables on which μ is defined. Two mappings μ_1 and μ_2 are *compatible* (written as $\mu_1 \sim \mu_2$) if $\mu_1(?x) = \mu_2(?x)$ for all variables $?x$ in $\mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2)$. 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 the variables in $\mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1)$. Given two sets of mappings M_1 and M_2, the *join* and *union* between M_1 and M_2 are defined respectively as follows:

$$M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1,\ \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\},$$
$$M_1 \cup M_2 = \{\mu \mid \mu \in M_1 \text{ or } \mu \in M_2\}$$

Let us now define the semantics of GRAPH-patterns. Consider a GRAPH-pattern $P = (\mathsf{GRAPH}\ g\ P')$. Then

$$[\![P]\!]^D_G = \begin{cases} [\![P']\!]^D_{\mathsf{gr}_D(g)} & \text{if } g \in \mathbf{I} \\ \bigcup_{u \in \mathbf{I}} \left([\![P']\!]^D_{\mathsf{gr}_D(u)} \bowtie \{\mu_{g \mapsto u}\} \right) & \text{if } g \in \mathbf{V} \end{cases}$$

where $\mu_{g \mapsto u}$ is the mapping with domain $\{g\}$ and where $\mu_{g \mapsto u}(g) = u$.

Next we recall the semantics of SPARQL queries. Let $\boldsymbol{q}$ be a SPARQL query and D a dataset. The answer $\mathsf{ans}(\boldsymbol{q}, D)$ of q over D depends on the form of q:

- If q is a SELECT query, then $\mathsf{ans}(\boldsymbol{q}, D)$ is the answer to q as defined in the SPARQL standard [12].
- If q is a c-query $\boldsymbol{q} = \mathsf{CONSTRUCT}\ H\ DS\ \ \mathsf{WHERE}\ P$, then let $u_1, \ldots, u_n$ be the IRIs in DS and $G_1, \ldots, G_n$ the graphs associated to these IRIs; and consider the dataset $D' = D \cup \{\langle u_1, G_1 \rangle, \ldots, \langle u_n, G_n \rangle\}$. We define:

$$\mathsf{ans}(\boldsymbol{q}, D) = \{\mu(t) \mid \mu \in [\![P]\!]^{D'}, t \text{ is a triple in } H \text{ and } \mu \text{ is defined on } vars(t)\}.$$

3 Adding Recursion to SPARQL

The most basic example of a recursive query in RDF is reachability: given a resource x, compute all the resources that are reachable from x via a path of arbitrary length. These queries, amongst others, motivated the inclusion of property paths into the recent SPARQL 1.1 standard [12]. However, as several authors subsequently pointed out, property paths fall short when trying to express queries

that involve more complex ways of navigating RDF documents (cf. [4, 7, 8, 22]) and as a result several extensions have been brought forward to combat this problem [2, 14, 17, 22]. Almost all of these extensions are also based on the idea of computing paths between nodes in a recursive way, and thus share a number of practical problems with property paths. Most importantly, these queries need to be implemented using algorithms that are not standard in SPARQL databases, as they are based on automata-theoretic techniques, or clever ways of doing Breadth-first search on the graph structure of RDF documents.

3.1 A Fixed Point Based Recursive Operator

We have decided to implement a different approach: a much more widespread recursive operator that allows us compute the fixed point of a wide range of SPARQL queries. Before proceeding with the formal definition we illustrate the idea behind such queries by means of an example.

Example 2. Recall graph G from Figure 1. In the Introduction we made a case for the need of a query that could compute all pairs of articles (A, A') such that A is linked to A' by a path of `wasRevisionOf` links and where all of the revisions along the way were made by the same user. We can compute this with the following recursive query.

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp AS {
    CONSTRUCT {?newversion ?user ?oldversion}
    FROM NAMED <http://db.ing.puc.cl/temp>
    WHERE{{
        ?newversion prov:wasRevisionOf ?oldversion .
        ?newversion prov:wasGeneratedBy ?edit .
        ?edit prov:used ?oldversion .
        ?edit prov:wasAssociatedWith ?user}
    UNION{
      GRAPH <http://db.ing.puc.cl/temp>
      {?newversion ?user ?intversion . ?intversion ?user ?oldversion}}}
}
SELECT ?newversion ?oldversion
FROM <http://db.ing.puc.cl/temp>
WHERE {?newversion ?user ?oldversion}
```

Let us explain how this query works. The second line specifies that a temporary graph named `http://db.ing.puc.cl/temp` is to be constructed according to the query below which consists of a UNION of two subpatterns. The first pattern does not use the temporary graph and it simply extracts all triples (A, U, B) such that A was a revision of B and U is the user generating this revision. All these triples should be added to the temporary graph.

Then comes the recursive part: if (A, U, B) and (B, U, C) are triples in the temporary graph, then we also add (A, U, C) to the temporary graph. We continue iterating until a fixed point is reached, and finally we obtain a graph that contains all the triples (A, U, A') such that A is linked to A' via a path of revisions of arbitrary length but always generated by the same user U. Finally, the SELECT query extracts all such pairs of articles from the constructed graph.

As hinted in the example, the following is the syntax for recursive queries. It is based on the recursive operator that is part of SQL.

Definition 1 (Syntax of recursive queries). *A* recursive SPARQL query*, or just recursive query, is either a SPARQL query or an expression of the form*

$$\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2, \tag{1}$$

where t is an IRI from $\mathbf{I}$, $\boldsymbol{q}_1$ is a c-query, and $\boldsymbol{q}_2$ is a recursive query. The set of all recursive queries is denoted rec-*SPARQL.*

Note that in this definition $\boldsymbol{q}_1$ is allowed to use the temporary graph t, which leads to recursive iterations. Furthermore, the query $\boldsymbol{q}_2$ could be recursive itself, which allows us to compose recursive definitions. As usual with this type of queries, semantics is given via a fixed point iteration.

Definition 2 (Semantics of recursive queries). *Let $\boldsymbol{q}$ be a recursive query of the form (1) and D an RDF dataset. If $\boldsymbol{q}$ is a non recursive query then* $\mathsf{ans}(\boldsymbol{q}, D)$ *is defined as usual. Otherwise the* answer $\mathsf{ans}(\boldsymbol{q}, D)$ *is equal to* $\mathsf{ans}(\boldsymbol{q}_2, D_{LFP})$*, where D_{LFP} is the least fixed point of the sequence $D_0, D_1, \ldots$ with $D_0 = D$ and*

$$D_{i+1} = D \cup \{\langle t, \mathsf{ans}(\boldsymbol{q}_1, D_i)\rangle\},\ \textit{for } i \geq 0.$$

In other words, D_1 is the union of D with a temporary graph t that corresponds to the evaluation of q_1 over D, D_2 is the union of D with a temporary graph t that corresponds to the evaluation of q_1 over D_1, and so on until $D_{i+1} = D_i$. Note that the temporary graph is completely rewritten after each iteration. This definition suggests the following pseudocode for computing the answers of a recursive query $\boldsymbol{q}$ of the form (1) over a dataset D[4]:

1. Initialize a temporary RDF graph named after the IRI t as $G_{\text{Temp}} = \emptyset$.
2. While $\mathsf{ans}(\boldsymbol{q}_1, D \cup \{\langle t, G_{\text{Temp}}\rangle\}) \neq G_{\text{Temp}}$ do:
   - Set $G_{\text{Temp}} = \mathsf{ans}(\boldsymbol{q}_1, D \cup \{\langle t, G_{\text{Temp}}\rangle\})$
3. Output $\mathsf{ans}(\boldsymbol{q}_2, D \cup \{\langle t, G_{\text{Temp}}\rangle\})$

Obviously this definition only makes sense as long as such fixed point exists. From the Knaster-Tarski theorem [16] it easily follows that the fixed point exists as long as queries used to define recursion are monotone. For the sake of presentation, here we ensure this condition by disallowing explicit negation (such as NOTEXISTS or MINUS) and optional matching from our c-queries (note that under construct queries, this fragment is known to be equivalent to queries defined by union of well designed graph patterns [15]). It was also shown in [15] that the existence of a fixed point can be guaranteed even when q_1 belongs to a rather technical fragment that does allow a limited form of negation and optional matching that extends beyond the use of unions of well designed patterns.

[4] For readability we assume that t is not a named graph in D. If this is not the case then the pseudocode needs to be modified to meet the definition above

3.2 Complexity Analysis

Recursive queries can use either the SELECT or the CONSTRUCT result form, so there are two decision problems we need to analyze. For SELECT queries we define the problem SelQueryAns, that receives as an input a recursive query Q using the SELECT result form, a tuple $\bar{a}$ of IRIs from $\mathbf{I}$ and a dataset D, and asks whether $\bar{a}$ is in $\mathsf{ans}(Q, D)$. For CONSTRUCT queries the problem ConQueryAns receives a recursive query Q with a CONSTRUCT result form, a triple (s, p, o) over $\mathbf{I} \times \mathbf{I} \times \mathbf{I}$ and a dataset D, and asks whether this triple belongs to $\mathsf{ans}(Q, D)$.

Proposition 1. SelQueryAns *is* PSPACE*-complete and* ConQueryAns *is* NP*-complete. The complexity of* SelQueryAns *drops to* Π_2^p *if one only considers* SELECT *queries given by unions of well-designed graph patterns.*

Thus, at least from the point of view of computational complexity, our class of recursive queries are not more complex than standard select queries [21] or construct queries [15]. We also note that the complexity of similar recursive queries in most data models is typically complete for exponential time; what lowers our complexity is the fact that our temporary graphs are RDF graphs themselves, instead of arbitrary sets of mappings or relations.

For databases it is also common to study the data complexity of the query answering problem, that is, the same decision problems as above but considering the input query to be fixed. We denote these problems as SelQueryAns(Q) and ConQueryAns(Q), for select and construct queries, respectively. As we see, the problem remains in polynomial time for data complexity, albeit in a higher class than for non recursive queries (see again [21] or [15]).

Proposition 2. *Both the problem* SelQueryAns(Q) *and the problem* ConQueryAns(Q) *are* PTIME*-complete. They remain* PTIME*-hard even for queries without negation or optional matching.*

However, even if theoretically the problems have the same combined complexity as queries without recursion and are polynomial in data complexity, any implementation of the above algorithm is likely to run excessively slow due to a high demand on computational resources (computing the temporary graph over and over again) and would thus not be useful in practice. For this reason, instead of implementing full-fledged recursion, we decided to support a fragment of recursive queries based on what is commonly known as *linear recursive queries* [1, 10]. This restriction is common when implementing recursive operators in other database languages, most notably in SQL [23], but also in graph databases [8], as it offers a wider option of evaluation algorithms while maintaining the ability of expressing almost any recursive query that one could come up with in practice. For instance, as demonstrated in the following section, linear recursion captures all the examples we have considered thus far and it can also define any query that uses property paths. Furthermore, it can be implemented in an efficient way on top of any existing SPARQL engine using a simple and easy to understand algorithm. Next we formally define this fragment.

4 Realistic Recursion in SPARQL

The concept of *linear recursion* has become popular in the industry as a restriction for fixed point operators in relational query languages, because it presents a good tradeoff between the expressive power of recursive operators and their practical applicability. Let Q be the query $\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2$, where t is an IRI from $\mathbf{I}$, $\boldsymbol{q}_1$ is a c-query, and $\boldsymbol{q}_2$ is a recursive query. We say that Q is *linear* if for every dataset D, the answer $\mathsf{ans}(Q, D)$ of the query corresponds to the least fixed point of the sequence given by

$$D_0 = D, \quad D_{-1} = \emptyset,$$
$$D_{i+1} = D_i \cup \{\langle t, \mathsf{ans}(\boldsymbol{q}_1, (D \cup D_i \setminus D_{i-1}))\rangle\}.$$

In other words, a recursive query is linear if, in order to compute the i-th iteration, we only need the original dataset plus the tuples that were added to the temporary graph t in the previous iteration. Considering that the final size of t might be comparable to the original dataset, linear queries save us from evaluating the query several times over an ever increasing dataset.

Most of the recursive extensions proposed for SPARQL are linear: from property paths [12] to nSPARQL [22], SPARQLeR [14] or Trial [17], and even our example. Unfortunately it is undecidable to check if a recursive query is linear (under usual complexity-theoretic assumptions) [9], so one needs to impose syntactic restrictions to enforce this condition. This is what we do next.

4.1 Linear recursive queries

Our queries are made from the union of a graph pattern that does not use the temporary IRI, denoted as $\boldsymbol{p}_{\text{base}}$ and a graph pattern $\boldsymbol{p}_{\text{rec}}$ that does mention the temporary IRI. Formally, a *linear recursive query* is an expression of the form

$$\begin{array}{l}\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\\ \quad \mathsf{CONSTRUCT}\, H\ DS\ \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}\ \mathsf{UNION}\ \boldsymbol{p}_{\text{rec}}\ \}\ \boldsymbol{q}_{\text{out}}\end{array} \tag{2}$$

with H and DS a construct template and dataset clause as usual, with $\boldsymbol{p}_{\text{base}}$ and $\boldsymbol{p}_{\text{rec}}$ graph patterns such that only $\boldsymbol{p}_{\text{rec}}$ is allowed to mention the IRI t and with $\boldsymbol{q}_{\text{out}}$ a linear recursive query. We further require that the recursive part $\boldsymbol{p}_{\text{rec}}$ mentions the temporary IRI only once. In order to describe our algorithm, we shall abuse the notation and speak of $\boldsymbol{q}_{\text{base}}$ to denote the query $\mathsf{CONSTRUCT}\, H\ DS\ \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}$ and $\boldsymbol{q}_{\text{rec}}$ to denote the query $\mathsf{CONSTRUCT}\, H\ DS\ \mathsf{WHERE}\ \boldsymbol{p}_{\text{rec}}$, respectively.

This simple yet powerful syntax resembles the design choices taken in most SQL commercial systems supporting recursion [23] and even graph databases [8].

For example, the query in example 2 is not linear, because the temporary IRI is used twice in the pattern. Nevertheless, it can be restated as the following query that uses one level of nesting:

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp1 AS {
    CONSTRUCT {?newversion ?user ?oldversion}
    FROM NAMED <http://db.ing.puc.cl/temp1>
    WHERE{
        {?newversion prov:wasRevisionOf ?oldversion .
        ?newversion prov:wasGeneratedBy ?edit .
        ?edit prov:used ?oldversion .
        ?edit prov:wasAssociatedWith ?user}
    UNION
        {}}
}
WITH RECURSIVE http://db.ing.puc.cl/temp2 AS {
    CONSTRUCT {?newversion ?user ?oldversion}
    FROM NAMED <http://db.ing.puc.cl/temp1>
    FROM NAMED <http://db.ing.puc.cl/temp2>
    WHERE{
        GRAPH <http://db.ing.puc.cl/temp1> {?newversion ?user ?oldversion}
    UNION{
        GRAPH <http://db.ing.puc.cl/temp1> {?newversion ?user ?intversion}.
        GRAPH <http://db.ing.puc.cl/temp2> {?intversion ?user ?oldversion}}}
}
SELECT ?newversion ?oldversion
FROM <http://db.ing.puc.cl/temp>
WHERE {?newversion ?user ?oldversion}
```

We wrote the union in the first query for clarity, but in general either $\boldsymbol{p}_{\text{base}}$ or $\boldsymbol{p}_{\text{rec}}$ can be empty. The idea of this query is to first dump all meaningful triples from the graph into a new graph `http://db.ing.puc.cl/temp1`, and then use this graph as a basis for computing the required reachability condition, that will be dumped into a second temporary graph `http://db.ing.puc.cl/temp2`[5].

Note that these queries are indeed linear, and thus we can perform the incremental evaluation that we have described above. The separation between base and recursive query also allows us to keep track of changes made in the temporary graph without the need of computing the difference of two graphs. We have decided to implement what is known as *seminaive evaluation*, although several other alternatives have been proposed for the evaluation of these types of queries (see [10] for a good survey). Our algorithm is presented in Algorithm 1.

So what have we gained? By looking at Algorithm 1 one realizes that in each iteration we only evaluate the query over the union of the dataset and the intermediate graph G_{temp}, instead of the previous algorithm where one needed the whole graph being constructed (in this case G_{ans}). Furthermore, $\boldsymbol{q}_{\text{base}}$ is evaluated only once, using $\boldsymbol{q}_{\text{rec}}$ in the rest of the iterations. Considering that the temporary graph may be large, and that no indexing scheme could be available, this often results in a considerable speedup for query computation. As we see next, the computational complexity is also reduced.

Complexity Analysis. We can find some explanation of why linear recursive queries behave better in practice when revisiting the computational complexity of the query answering problem, which shows a reduction in data complexity.

Theorem 1. *If Q is a linear recursive query,* SelQueryAns(Q) *and* ConQueryAns(Q) *are* NLogSpace*-complete.*

[5] One can show that in this case the nesting in this query can be avoided.

Algorithm 1 Computing the answer for linear recursive queries of the form (2)

Input: Query Q of the form (2), dataset D
Output: Evaluation $\mathsf{ans}(Q, D)$ of Q over D

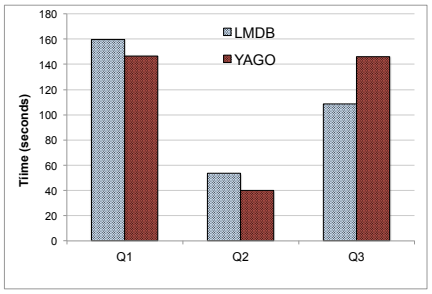
Set $G_{\text{temp}} = \mathsf{ans}(\boldsymbol{q}_{\text{base}}, D)$ and $G_{\text{ans}} = G_{\text{temp}}$
Set size $= |G_{\text{ans}}|$
loop
  Set $G_{\text{temp}} = \mathsf{ans}(\boldsymbol{q}_{\text{rec}}, D \cup \{(t, G_{\text{temp}})\})$
  Set $G_{\text{ans}} = G_{\text{ans}} \cup G_{\text{temp}}$
  if size $= |G_{\text{ans}}|$ **then**
    break
  else
    size $= |G_{\text{ans}}|$
  end if
end loop
return $\mathsf{ans}(\boldsymbol{q}_{\text{out}}, D \cup \{\langle t, G_{\text{ans}}\rangle\})$

5 Experimental Evaluation

Our implementation of linear recursive queries was carried out using the Apache Jena framework [13] as an add-on to the ARQ SPARQL query engine. The version used was Jena TDB 2.12.1 as it allows the user to run queries either in main memory, or using disk storage when needed. As previously mentioned, since the query evaluation algorithms we develop make use of the same operations that already exist in current SPARQL engines, we can use those as a basis for the recursive extension to SPARQL we propose. In fact, as we show by implementing recursion on top of Jena, this capability can be added to an existing engine in an elegant and non-intrusive way[6].

We test our implementation using three different datasets. The first one is Linked Movie Database (LMDB) [18], an RDF dataset containing information about movies and actors[7]. The second dataset we use is a part of the YAGO ontology [28] and consists of all the facts that hold between instances. For the experiments the version from March 2015 was used. The last dataset is based on Missier and Chen's database of Wikipedia traces [19] we described previously. We chose 3 of their datasets, but since they are very small we enlarge them by taking disjoint copies of the same data until it reached the desired size. Since these datasets contain only the traces and nothing else we also added 30% of random unrelated triples to simulate the database containing other pieces of information. We grew 4 different datasets out of the provenance traces, of 50, 100, 150 and 200 Mb of size approximately. We refer to these datasets as PROV1, PROV2, PROV3 and PROV4, respectively[8]. All the experiments were run on a MacBook Air with an Intel Core i5 1.3 GHz processor and 4GB of main memory.

[6] The implementation we use is available at http://web.ing.puc.cl/~jreutter/Recsparql.html.
[7] We use the data dump available at http://queens.db.toronto.edu/~oktie/linkedmdb/.
[8] The datasets are available at http://web.ing.puc.cl/~jreutter/Recsparql.html.



(a) Query times on LMDB and YAGO

dataset \ query	Q1	Q2	Q3
LMDB	37349	1172	14568
YAGO	25404	480	9416

(b) The number of output tuples

Dataset	PROV1	PROV2	PROV3	PROV4
Time(sec)	12.3	22.8	33.8	46.5
No. tup.	220950	441900	667269	883800

(c) Query from Section 4 on PROV datasets

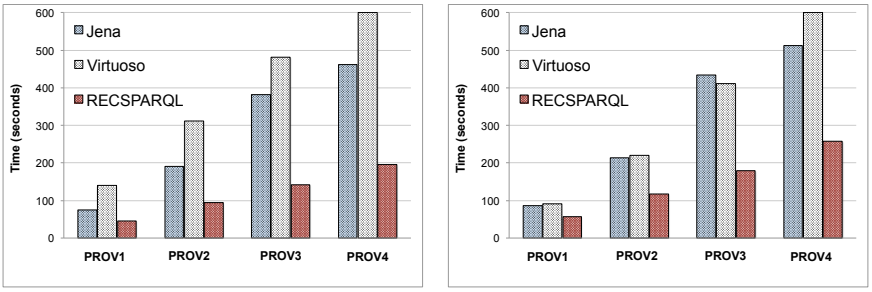
Fig. 3. Running times and the number of output tuples for the three datasets.

5.1 Query evaluation

Because of the novelty of our approach it was impossible to compare our times against other implementations, or run standard benchmarks to test the performance of our queries. Furthermore, while our formalism is similar to that of recursive SQL, all of the RDF systems that we checked were either running RDF natively, or running on top of a relational DBMS that did not support recursion as mandated by the SQL standard. OpenLink Virtuoso does have a *transitive closure* operator, but this operator can only compute transitivity when starting in a given IRI. Our queries were more general than this, and thus we could not compare them. For this reason we invented several queries that are very natural over the considered datasets and tested their performance. As all property paths can be expressed by linear recursive queries we will also test our implementation against current SPARQL systems in the following subsection.

We start our round of experiments with movie-related queries over both LMDB and YAGO. Since YAGO also contains information about movies, we have the advantage of being able to test the same queries over different real datasets (only the ontology differs). We use three different queries, all of them similar to that of Example 2. The first query Q1 returns all the actors in the database that have a finite Bacon number[9], meaning that they co-starred in the same movie with Kevin Bacon, or another actor with a finite Bacon number. A similar notion, well known in mathematics, is the Erdős number. Note that Q1 is a property path query. To test recursive capabilities of our implementation we use another two queries, Q2 and Q3, that apply various tests along the paths computing the Bacon number. The query Q2 returns all actors with a finite Bacon number such that all the collaborations were done in movies with the same director. Finally the query Q3 tests if an actor is connected to Kevin Bacon through movies where the director is also an actor (not necessarily in the same movie). The structure of queries Q2 and Q3 is similar to the query from Example 2 and cannot be expressed using property paths either. The results of the evaluation can be found in Figure 3(a). As we can see the running times,

[9] See http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon.



(a) Times for Q_A (b) Times for Q_B

Fig. 4. Evaluation time of Q_A and Q_B in our implementation is comparable to that of Jena or Virtuoso. For PROV4 both queries reported more than 600 seconds in Virtuoso.

although high, are reasonable considering the size of the datasets and the number of output tuples (Figure 3(b)).

The next round of experiments pushes our implementation to compute inherently recursive queries. For this we use the query from Example 2 that finds all pairs of Wikipedia articles whose revision history can be attributed to the same user. As we implement linear recursion, the version of the query presented in Section 4 is used. Figure 3(c) shows the running time of this query on the datasets derived from Wikipedia traces described before; it illustrates that running times are quite low when we take the number of computed tuples into consideration.

5.2 Comparison with Property Paths

Since to the best of our knowledge no SPARQL engine implements general recursive queries, we cannot really compare the performance of our implementation with the existing systems. The only form of recursion mandated by the latest language standard are property paths, so in this section we test how our implementation stacks against popular systems when executing property paths.

Every property path query is easily expressible using linear recursion. However, it is not fair to compare our recursive implementation of property paths to the one in current systems, as they specialize in executing this type of recursive queries, while the recursive operator we introduced is aimed at expressing a wide variety of queries that lie beyond the scope of property paths. For this reason highly efficient systems like Virtuoso will run queries they are optimized for much faster. For instance to run the query Q1 from Subsection 5.1 that computes all actors with a finite Bacon number in LMDB or YAGO Virtuoso takes less than 10 seconds, while our implementation takes much longer. Part of the difference in running times could be attributed to the fact that in this particular case our implementation runs queries on disc, while Virtuoso can perform them in main memory, but the main detractor is the fact that Virtuoso is designed to be efficient at property paths that are given a starting point, while recursive queries are not since they can express more general queries.

To have a somewhat fair comparison we will use property path queries that compute all pairs of IRIs connected by a specified property path. We use the PROV datasets introduced above and in Figure 1 and test for the existence of property paths `wasRevisionOf`* and `(wasGeneratedBy/used)`*. We refer to these queries as Q_A and Q_B.[10] Figure 4 presents the time each of the queries takes on the four PROV datasets of increasing size. We test the recursive implementation of property paths against the one in Jena and Virtuoso. As we can see our implementation is quite competitive with systems that specialize in property paths when we need to compute the entire relation. We can also see that Jena runs faster than Virtuoso in this case and we believe that this is due to the fact that Jena implements property paths in a way that returns all pairs of nodes that are connected by the specified query, while for Virtuoso we need to run the query from every possible starting point.

5.3 Limiting the number of iterations

In practical scenarios users are often interested in running recursive queries only for a predefined number of iterations. For instance, very long paths between nodes are seldom of interest and in a many use cases we will be interested in using property paths only up to depth four or five. For this reason we propose the following syntax to restrict the depth of recursion to a user specified number:

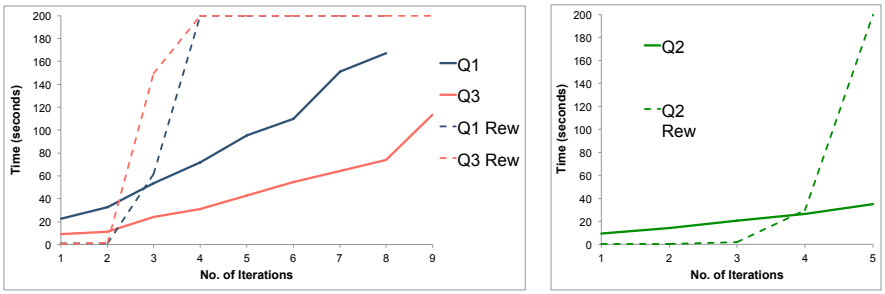
$$\begin{aligned}&\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\\ &\quad \mathsf{CONSTRUCT}\ H\ DS\ \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}\ \mathsf{UNION}\ \boldsymbol{p}_{\text{rec}}\\ &\quad \}\ \mathsf{MAXRECURSION}\ k\ \boldsymbol{q}_{\text{out}}\end{aligned} \tag{3}$$

Here all the keywords are the same as when defining linear recursion, and $k \geq 1$ is a natural number. The semantics of such queries is defined using Algorithm 1, where the loop between steps 4 and 12 is executed precisely $k - 1$ times.

It is straightforward to see that every query defined using recursion with predefined number of iterations can be rewritten in SPARQL by explicitly specifying each step of the recursion and joining them using the union operator. The question then is, why is specifying the recursion depth beneficial?

One apparent reason is that it makes queries much easier to write and understand. The second reason we would like to argue for is that, when implemented using Algorithm 1, recursive queries with a predetermined number of steps result in faster query evaluation times than evaluating an equivalent query with lots of joins. The intuitive reason behind this is that computing $\boldsymbol{q}_{\text{base}}$, although expensive initially, acts as a sort of index to iterate upon, resulting in fast evaluation times as the number of iterations increases. On the other hand, for even a moderately complex query using lots of joins, the execution plan will seldom be optimal and will often resort to simply trying all the possible matchings to the variables, thus recomputing the same information several times.

[10] Note that in Virtuoso we need to specify the starting point of a property path. This is done by extracting each node from a unique triple containing it.



(a) Queries Q_1 and Q_3 (b) Query Q_2

Fig. 5. Limiting the number of iterations for Q_1, Q_2 and Q_3 over LMDB. Recursion dominates manually written SPARQL joins when several iterations are required.

We substantiate this claim by running two rounds of experiments on LMDB and YAGO using queries Q1,Q2 and Q3 from Subsection 5.1 and running them for an increasing number of steps. In the first round we evaluate each of the queries using Algorithm 1 and run it for a fixed number of steps until the algorithm saturates. In the second round we use a SPARQL rewriting of a recursive query where the depth of recursion is fixed and evaluate it in Jena.

Figure 5 shows the results over LMDB. The results for YAGO show the same trend, so we do not include them. As we can see, the initial cost is much higher if we are using recursive queries, however as the number of steps increases we can see that they show much better performance and in fact, the queries that use only SPARQL operators time out after a small number of iterations.

6 Conclusion

As illustrated by several use cases, there is a need for recursive functionalities in SPARQL that go beyond the scope of property paths. To tackle this issue we propose a recursive operator to be added to the language and show how it can be implemented efficiently on top of existing SPARQL systems. We concentrated on linear recursive queries which have been well established in SQL practice and cover almost all interesting use cases and show how to implement them as an extension to Jena framework. Our tests show that, although very expressive, these queries run in reasonable time even on a machine with limited computational resources. We also include a command that allows to run recursive queries for a limited number of steps and show that the proposed implementation outperforms equivalent queries specified using only SPARQL operators. We believe all of this to be a good indicator of the usefulness of the recursion operator and why it should be a potential candidate for inclusion in the next SPARQL standard.

Acknowledgements. This work was funded by the Millennium Nucleus Center for Semantic Web Research Grant NC120004. We would like to thank Aidan Hogan, Egor Kostylev, James Cheney and the reviewers for helpful comments.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. A-W, 1995.
2. F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73, 2009.
3. F. Alkhateeb and J. Euzenat. Constrained regular expressions for answering rdf-path queries modulo RDFS. *IJWIS*, 10(1):24–50, 2014.
4. K. Anyanwu and A. Sheth. ρ-Queries: enabling querying for semantic associations on the semantic web. In *WWW*, 2003.
5. M. Atzori. Computing recursive SPARQL queries. In *ICSC*, pages 258–259, 2014.
6. P. Barceló, J. Pérez, and J. L. Reutter. Relative Expressiveness of Nested Regular Expressions. In *AMW*, pages 180–195, 2012.
7. P. Bourhis, M. Krötzsch, and S. Rudolph. How to best nest regular path queries. In *Proceedings of the 27th International Workshop on Description Logics*, 2014.
8. M. Consens and A. Mendelzon. Graphlog: A visual formalism for real life recursion. In *PODS*, pages 404–416, 1990.
9. H. Gaifman, H. Mairson, Y. Sagiv, and M. Y. Vardi. Undecidable optimization problems for database logic programs. *JACM*, 40(3):683–713, 1993.
10. T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.
11. A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling kleene: fast property paths in RDF-3X. In *GRADES*, 2013.
12. S. Harris and A. Seaborne. SPARQL 1.1 query language. *W3C Recommendation*, 21, 2013.
13. The Apache Jena Manual. http://jena.apache.org, 2015.
14. K. J. Kochut and M. Janik. Sparqler: Extended sparql for semantic association discovery. In *The SW: Research and Applications*, pages 145–159. Springer, 2007.
15. E. V. Kostylev, J. L. Reutter, and M. Ugarte. CONSTRUCT queries in SPARQL. In *ICDT*, pages 212–229, 2015.
16. L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.
17. L. Libkin, J. Reutter, and D. Vrgoč. Trial for RDF: adapting graph query languages for RDF data. In *PODS*, pages 201–212. ACM, 2013.
18. Linked movie database. http://linkedmdb.org/.
19. P. Missier and Z. Chen. Extracting prov provenance traces from wikipedia history pages. In *EDBT/ICDT 2013 Workshops*, pages 327–330, 2013.
20. B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI*, 2014.
21. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3), 2009.
22. J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.
23. PostgreSQL. http://www.postgresql.org/docs/current/interactive/queries-with.html.
24. PROV Model Primer. http://www.w3.org/TR/2013/NOTE-prov-primer-20130430/, 2013.
25. M. Shaw, L. F. Detwiler, N. F. Noy, J. F. Brinkley, and D. Suciu. vSPARQL: A view definition language for the semantic web. *J. Biomedical Informatics*, 44, 2011.
26. Open Link Virtuoso. http://virtuoso.openlinksw.com/, 2015.
27. W3C. PROV-O: The PROV Ontology. http://www.w3.org/TR/prov-o/, 2013.
28. YAGO: A High-Quality Knowledge Base. http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/.