Semantic Web 0 (0) 1
IOS Press

Recursion in SPARQL

Juan Reutter [a], Adrián Soto [a,*] and Domagoj Vrgoč [b]

[a] *Departamentio de Ciencia de la Computación, Pontificia Universidad Católica de Chile and IMFD Chile, Chile*
E-mails: jreutter@ing.puc.cl, assoto@uc.cl
[b] *Departamentio de Ingeniería Matemática y Computacional, Pontificia Universidad Católica de Chile and IMFD Chile, Chile*
E-mail: dvrgoc@ing.puc.cl

Abstract. The need for recursive queries in the Semantic Web setting is becoming more and more apparent with the emergence of datasets where different pieces of information are connected by complicated patterns. This was acknowledged by the W3C committee by the inclusion of property paths in the SPARQL standard. However, as more data becomes available, it is becoming clear that property paths alone are not enough to capture all recursive queries that the users are interested in. 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 it as a plug-in on top of existing systems and test its performance on several real world datasets.

Keywords: SPARQL, Recursive Queries, Property Paths

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 [1]. After the initial proposal of SPARQL, and with more data becoming available in the RDF format, users found use cases that asked for more complex querying features that allow 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 [2], 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. [3–9]), and that a more powerful form of recursion needs to be added to SPARQL to address this issue. Consequently, this realization has brought forward a good number of extensions of property paths that offer more expressive recursive functionalities (see e.g. [3, 7, 10–14]), 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 [15]. They use the PROV standard [16] 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 [17]. 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.

*Corresponding author. E-mail: assoto@uc.cl.

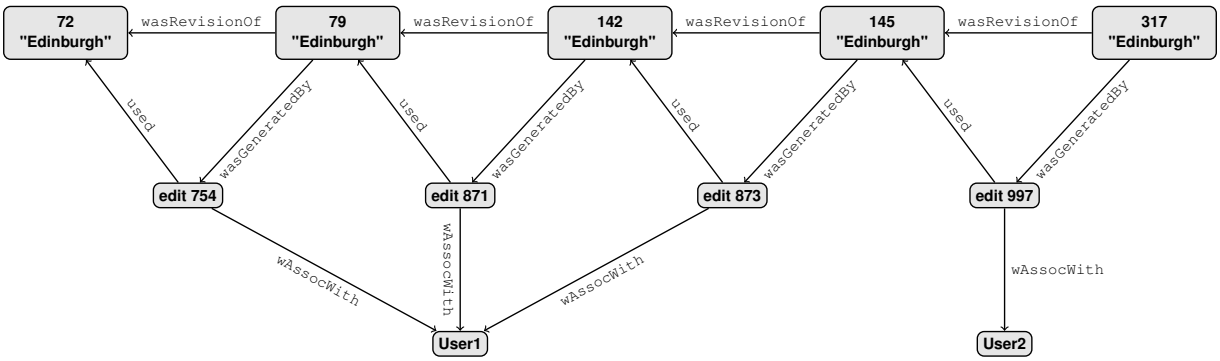


Fig. 1. RDF database of Wikipedia traces. The abbreviation `wAssocWith` is used instead of `wasAssociatedWith` and the `prov:` prefix is omitted from all the properties in this graph.

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 the graph of 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 any version control system (note that the PROV traces of Wikipedia articles are the same as 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 [4, 12, 13], nor by using standard SPARQL functionalities (as provenance traces can contain links of arbitrary length), the need for a general purpose recursive operator seems like a natural addition to the language.

One possible reason why recursion was not previously considered as an integral operator of SPARQL 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 as well, and indeed [18] 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 [18] to function over a more widely used fragment of SPARQL and study how this operator can be implemented in an efficient and non-intrusive way on top of existing SPARQL engines. We begin by showing what the general form of recursion looks like and how to evaluate it. We then argue that any implementation of this general form of recursion is unlikely to perform well on real world data, so we restrict it to the so called *linear recursion*, which is well known in the relational context [19, 20]. We then demonstrate that even this restricted class of queries can express most use cases for recursion 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 (version 3.7.0) framework [21] and we implement recursive queries as an add-on to the ARQ SPARQL query engine. We use Jena TDB (version 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 and LMBD using several natural queries over these datasets. We also test our implementation using the GMark Benchmark to compare it against the property paths implementation of Jena and Virtuoso.[1]

Related work. As mentioned previously the most common type of recursion implemented in SPARQL systems are property paths. This is not surprising as property paths are a part of the latest language standard and there are now various systems supporting them either in a full capacity[22, 23], or with some limitations that ensure they can be efficiently evaluated, most notable amongst them being Virtuoso [24]. The systems

[1]The implementation, test data and complete formulation of all the queries can be found at https://alanezz.github.io/RecSPARQL/.

that support full property paths are capable of returning all pairs of nodes connected by a property path specified by the query, while Virtuoso needs a starting point in order to execute the query. We would like to note that recursive queries we introduce are capable of expressing the transitive closure of any binary operator [18] and can thus be used to express property paths and any of their extensions [3, 7, 10, 14]. Regarding attempts to implement a full-fledged recursion as a part of SPARQL, there have been none as far as we are aware. There were some attempts to use SQL recursion to implement property paths [25], or to allow recursion as a programming language construct [26, 27], however none of them view recursion as a part of the language, but as an outside add-on.

Remark. A previous version of this article was presented at the International Semantic Web Conference in 2015 [28]. This extended version features a detailed discussion on possible extensions to the semantics (in order to support, say, `BIND` operations or some form of negation), complete proofs form most results (except those that are simple applications of other theorems already in the literature), improved experiments and a thorough comparison of the runtime of our approach with respect to property paths. Looking forward, we also include a section where we discuss the advantages, disadvantages and possible uses of our language and approach.

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. We also use $\mathcal{G}$ and $\mathcal{D}$ to denote the sets of all RDF graphs and datasets, correspondingly.

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. Unions of datasets without default graphs is defined in the same way, i.e., as if the default graph was empty.

SPARQL Syntax and Semantics. SPARQL is the standard pattern-matching language for querying RDF datasets. We will use de usual syntax based on graph patterns and the usual semantics based on mappings. Both topics can be found in [2].

We recall the usage of the GRAPH operator: let g be an IRI or a variable and P be a pattern. The expression (GRAPH g P) allows us to determine which graph from the dataset we will be matching the pattern P to. 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.

The fragment of SPARQL graph patterns, as well as its generalisation to SELECT queries, has drawn most of the attention in the Semantic Web community. However, as the results of such queries need not be RDF graphs, we shall use the CONSTRUCT operator in order to obtain 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*[3], 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 $i \geqslant 0$, that is called a *dataset clause*[4], and P is a graph pattern.

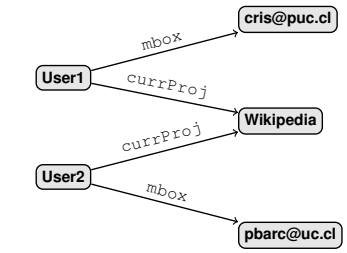
The idea behind the CONSTRUCT operator is that the mappings matched to the pattern P over the graphs $u_1, \ldots, u_n$ are used to construct an RDF graph according to the template H. Since all the patterns in the template are triples we will end up with an RDF graph as desired. Before giving the formal semantics of the operators we illustrate how they work by means of an example.

Example 2.1. Let G and G_1 be the graphs in Figure 1 and Figure 2(a), respectively. Suppose we want to query both graphs to obtain a new graph where each

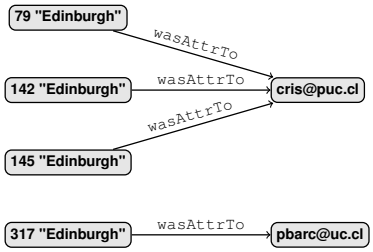
[2] For clarity of presentation we do not include literals nor blank nodes in our definitions.

[3] We leave the study of blanks in templates as future work.

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



(a) Graph G_1



(b) Result of the query in Example 2.1

Fig. 2. Graphs used for Example 2.1. The prefixes `foaf:` and `prov:` are omitted.

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:

```
CONSTRUCT {
  ?article prov:wasAttributedTo ?mail
}
FROM NAMED <http://db.puc.cl/mail>
WHERE {
   ?article prov:wasGeneratedBy
   ?comment .
   ?comment prov:wasAssociatedWith
   ?usr .
   GRAPH <http://db.puc.cl/mail>
   {
     ?usr foaf:mbox ?mail
   }
}
```

We call $\mathsf{ans}(\boldsymbol{q}, D)$ the result of evaluating q over D. In this case, it 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`.

3. Adding Recursion to SPARQL

The most basic example of a recursive query in the RDF context is that of reachability: given a resource x, compute all the resources that are reachable from x via a path of arbitrary length. These type of queries, amongst others, motivated the inclusion of property paths into the SPARQL 1.1 standard [2].

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. [3, 6, 10, 29]) and as a result several extensions have been brought forward to combat this problem [3, 7, 11, 12, 14, 30]. 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 over a 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. This is based on the recursive operator that was added to SQL when considering similar challenges. We cannot define this type of operator for SPARQL SELECT queries, since these returns mappings and thus no query can be applied to the result of a previous query, but we can do it for CONSTRUCT queries, since these return RDF graphs. Following [18], we now define the language of *Recursive Queries*. Before proceeding with the formal definition we illustrate the idea behind such queries by means of an example.

Example 3.1. 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 recursive query of the Figure 3.

Let us explain how this query works. The second line specifies that a temporary graph named:

`http://db.ing.puc.cl/temp`

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp AS {
    CONSTRUCT {?x ?u ?y}
    FROM NAMED <http://db.ing.puc.cl/temp>
    WHERE {
    { ?x prov:wasRevisionOf ?y .
      ?x prov:wasGeneratedBy ?w .
      ?w prov:used ?y .
      ?w prov:wasAssociatedWith ?u }
    UNION {
        GRAPH <http://db.ing.puc.cl/temp> { ?x ?u ?z } .
        GRAPH <http://db.ing.puc.cl/temp> { ?z ?u ?y } }
    }
}
SELECT ?x ?y
FROM <http://db.ing.puc.cl/temp>
WHERE ?x ?u ?y
```

Fig. 3. Example of a recursive query.

will 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:

Definition 3.1 (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 3.2 (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 $\mathsf{ans}(\boldsymbol{q}_2, D_{\mathrm{LFP}})$, where D_{LFP} is the least fixed point of the sequence $D_0, D_1, \ldots$ with $D_0 = D$ and

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

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 pseudocode of Algorithm 1 for computing the answers of a recursive query $\boldsymbol{q}$ of the form (1) over a dataset D[5].

Obviously this definition only makes sense as long as such fixed point exists. Unfortunately, we show in the following section that there are queries for which this operator indeed does not have a fixed point. Thus,

[5]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

Algorithm 1 Computing the answer for recursive c-queries of the form (1)

Input: Query Q of the form (1), dataset D
Output: Evaluation $\mathsf{ans}(Q, D)$ of Q over D

1: Set $G_{\text{temp}} = \emptyset$ named after the IRI t
2: **loop**
3: Set $G_{\text{Temp}} = \mathsf{ans}(\boldsymbol{q}_1, D \cup \{\langle t, G_{\text{Temp}}\rangle\})$
4: **if** $\mathsf{ans}(\boldsymbol{q}_1, D \cup \{\langle t, G_{\text{Temp}}\rangle\}) = G_{\text{Temp}}$ **then**
5: **break**
6: **end if**
7: **end loop**
8: **return** $\mathsf{ans}(\boldsymbol{q}_2, D \cup \{\langle t, G_{\text{Temp}}\rangle\})$

we need to restrict the language that can be applied to such inner queries[6]. We also discuss other possibilities to allow us using any operator we want.

3.2. Ensuring fixed point of queries

We know that a fixed point exists for recursive queries as long as they are monotone. This can be obtained from the Knaster-Tarski theorem [31]. A query Q is said to be monotone if for all pair of datasets D_1, D_2 where $D_1 \subseteq D_2$ it holds that $Q(D_1) \subseteq Q(D_2)$. However, what happens if queries are not monotone? Let us illustrate the problem with two of the most typical non-monotonic operators: explicit negation and invention of new values.

Adding negation. The problem with negation is that one can use it to alternate the presence of some triples in each iteration of the recursion, and therefore come up with recursive queries where the fixed point does not exists.

Example 3.2. Consider the following query that contains a NOT EXISTS clause.

```
PREFIX ex: <http://example.org>
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT {?x ?y "a"}
  FROM NAMED <http://db.puc.cl/temp>
  WHERE {
    { ?x ?y ?z }
    GRAPH <http://db.puc.cl/temp> {
      FILTER NOT EXISTS
      { ?x ?y "a" }
    }
  }
}
SELECT *
FROM NAMED <http://db.puc.cl/temp>
WHERE { ?x ?y ?z }
```

Also consider the following instance for the default graph:

```
ex:s1 ex:p1 "b" .
ex:s2 ex:p2 "b" .
```

In the first iteration, the graph <temp> will have the triples:

```
ex:s1 ex:p1 "a" .
ex:s2 ex:p2 "a" .
```

But in the next iteration the graph <temp> will be empty because of the NOT EXISTS clause. The <temp> graph will be alternating between an empty graph and a graph with the triples mentioned above. Thus, the fixed point does not exist for this query.

Similar examples can be obtained with other SPARQL operators that can simulate negation, such as MINUS or even arbitrary OPTIONAL [32, 33]. This is why we disallow the usage of all these operators inside our recursive clauses. Of course, one can simply choose to allow them, at the risk allowing users to write queries that enter an infinite loop.

Creating values. The BIND clause allows us to generate new values that were not in the domain of the database before executing a recursive query. This means that, even if no negation is present, one can use BIND to generate completely new values for the temporal graph t in each iteration. Once again, this implies that a fixed point may not exists, but this time not because there is a loop but because we keep inserting completely new triples to the database.

Example 3.3. Consider the following query that makes use of the BIND clause.

```
PREFIX ex: <http://example.org>
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT {?x ex:number ?b}
  FROM NAMED <http://db.puc.cl/temp>
  WHERE {
```

[6] it should be noted that the recursive SQL operator has the same problem, and indeed the SQL standard restricts which SQL features can appear inside a recursive operator.

```
    { ?x ex:type ex:person .
      ?x ex:age ?a .
      BIND (?a AS ?b) }
    UNION {
      GRAPH <http://db.puc.cl/temp> {
        ?x ex:number ?aux .
        BIND(?aux + 1 AS ?b)
      }
    }
  }
}
SELECT *
FROM NAMED <http://db.puc.cl/temp>
WHERE { ?x ?y ?z }
```

The base graph stores the age for all the people in the database. In each iteration, we will increase by one all the objects in our graph and then we will store triples with those new values. As we mentioned, in each iteration the query will try to insert new triples into the database.

Besides the BIND operator, we can also simulate the creation of new values by means of blanks in the construct templates, or even with blanks inside queries or subqueries. Once again, systems can allow the usage of these operators at their own risk.

Using stable model semantics. A possible solution to support BIND and Negation is to extend the semantics by borrowing the notion of stable models from logic programs (see e.g. [34]). The idea of a stable model is the following: consider a program P and a set of literals D. P can contain a set of rules, where each rule has literals that are in D and such literals may be negated. A stable model is a set of literals $S \in D$ appearing in the program that has to be equals to a set called **Deductive Closure of** P^S. P^S is a program obtained from deleting the rules with negation in P in a certain way according to the variables in S. We can say that a Stable Model is an answer for the program P. Adding BIND supposes additional complications. The natural analog for BIND operators in programs is a function. But when programs combine variables and functions, the notion of stable models becomes more involved, because one now needs to replace each rule with k variables for $|D|^k$ rules, where $|D|$ is the number of elements in the set D, and compute a relation for each function, storing all the inputs with their respective outputs.

Thus, coming up with a reasonable implementation under these semantics is definitely out of the scope of this paper. However, we do remark that such an extension to the semantics of recursive SPARQL queries is an interesting topic for future work.

Queries we support. We know that negation may lead to infinite loops, and any process that creates new values may also go infinite. So what types of queries we support? In this paper, and for our experiments, the fragment of SPARQL we support are **Group Graph Patterns** without negation (including NOT EXISTS), filters nor blanks, and with further combination through UNION or VALUES clauses (but not with BIND). At this point, we have a few points to discuss.

First, we remark that one could support much more SPARQL queries. For example, we have not dealt with operators such as subqueries, federation or aggregation, and there are various restrictions we can do to BIND clauses that could allow for fixed points. The intention of this paper is to argue in favor of the usage of a recursive operator within SPARQL, but we believe that the question of what exactly should be allowed in inner queries is a question better to be addressed with more involvement from the community. Nevertheless, the fragment we study here is a good fragment capable of expressing a wide range of queries. However, we do note that our fragment is as expressive as Group Graph Patterns with well-designed optionals [3]. This is because for CONSTRUCT queries, the fragment we consider has been shown to contain queries defined by union of well designed graph patterns [18]).

As another option for supporting more SPARQL queries inside the recursion, in Section 4.2 we will show an alternative way for computing a recursive query that limits the number of iterations that the recursive algorithm can perform. Such way of evaluating a recursive query works regardless of the existence of a fixed point, and therefore we can allow for full SPARQL as long as we limit the number of iteration of queries.

3.3. Complexity Analysis

Since recursive queries can use either the SELECT or the CONSTRUCT result form, there are two decision problems we need to analyze. For SELECT queries, we define the problem SELECTQUERYANSWERING, 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 CONSTRUCTQUERYANSWERING receives a recursive query Q us-

ing the 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 is in $\mathsf{ans}(Q, D)$.

Proposition 3.1. The problem SELECTQUERYANSWERING is PSPACE-complete and CONSTRUCTQUERYANSWERING is NP-complete. The complexity of SELECTQUERYANSWERING drops to Π_2^p if one only consider SELECT queries given by unions of well-designed graph patterns.

Proof. It was proved in [18] that the problem CONSTRUCTQUERYANSWERING is NP-complete for non recursive c-queries, and Pérez et al. show in [35] that the problem SELECTQUERYANSWERING if PSPACE-complete for non-recursive SPARQL queries, and Π_2^p for non-recursive SPARQL queries given by unions of well-designed graph patterns. This immediately gives us hardness for all three problems when recursion is allowed.

To see that the upper bound is maintained, note that for each nested query, the temporal graph can have at most $|D|^3$ triples. Since we are computing the least fixed point, this means that in every iteration we add at least one triple, and thus the number of iterations is polynomial. This in turn implies that the answer can be found by composing a polynomial number of NP problems, to construct the temporal graph corresponding to the fixed point, followed by the problem of answering the outer query over this fixed point database, which is in PSPACE for SELECTQUERYANSWERING, in Π_2^p for SELECTQUERYANSWERING assuming queries given by unions of well designed patterns and in NP for CONSTRUCTQUERYANSWERING. First two classes are closed under composition with NP, and the last NP bound can be obtained by just guessing all meaningful queries, triples to be added and witnesses for the outer query at the same time. □

Thus, at least from the point of view of computational complexity, our class of recursive queries are not more complex than standard select queries [35] or construct queries [18]. 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 this problems as SELECTQUERYANSWERING(Q) and CONSTRUCTQUERYANSWERING(Q), for select and result queries, respectively. The following shows that the problem remains in polynomial time for data complexity, albeit in a higher class than for non recursive queries.

Proposition 3.2. SELECTQUERYANSWERING(Q) and CONSTRUCTQUERYANSWERING(Q) are PTIME-complete. They remain PTIME-hard even for queries without negation or optional matching.

Proof. Following the same idea as in the proof of Proposition 3.1, we see that the number of iterations needed to construct the fixed point database is polynomial. But, if queries are fixed, the problem of evaluating SELECT and CONSTRUCT queries is always in NLOGSPACE (see again [35] and [18]). The PTIME upper bound then follows by composing a polynomial number of NLOGSPACE algorithms.

We prove the lower bound by a reduction from the *path systems* problem, which is a well known PTIME-complete problem (c.f. [36]). The problem is as follows. Consider a a set of nodes V and a unary relation $C(x) \subseteq V$ that indicates whether a node is *coloured* or not. Let $R(x, y, z) \subseteq V \times V \times V$ be a relation of reachable elements, and the following rule for colouring additional elements: if there are coloured elements a, b such that a triples (a, b, c) is coloured, then c should also be coloured. Finally consider a *target* relation $T \subseteq V$. The problem of *path systems* is to decide if some element in T is coloured by our rule.

For our reduction we construct a database instance and a (fixed) recursive query according to the instance of *path systems* such that the result of the query is empty if and only if $T \subseteq P$ for the *path system* problem. The construction is as follows.

The database instance contains the information of which vertex is coloured, which vertex is part of the target relation T and the elements of the R relation:

- We define the function u which maps every vertex to a unique URI.
- For each element $v \in C$, we add the triple $(u(v), \texttt{ex:p}, \texttt{"C"})$ to a named graph `gr:C` of the database instance.
- For each element $v \in T$, we add the triple $(u(v), \texttt{ex:p}, \texttt{"T"})$ to a named graph `gr:T` of the database instance.

– For each element $(x, y, z) \in R$ we add the triple $(u(x), u(y), u(z))$ to the default graph of the database instance.

Thus, the recursive query needs to compute all the coloured elements in order to check if the target relation is covered. This can be done in the following way:

```
PREFIX ex: <http://example.org>
PREFIX gr: <http://example.org/graph>
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT { ?z ex:p "C" }
  FROM NAMED <http://db.puc.cl/temp>
  FROM NAMED gr:C
  WHERE {
    { GRAPH gr:C { ?z ex:p "C" } }
    UNION {
      { ?x ?y ?z } .
      GRAPH <http://db.puc.cl/temp> {
        ?x ex:p "C" } .
      GRAPH <http://db.puc.cl/temp> {
        ?y ex:p "C" }
      }
    }
  }
}
ASK
FROM NAMED <http://db.puc.cl/temp>
FROM NAMED gr:T
WHERE {
  GRAPH gr:T {
    ?x ex:p "T"
  } .
  GRAPH <http://db.puc.cl/temp> {
    ?x ex:p "C"
  }
}
```

It is clear that the recursive part of the query is computing all the coloured nodes according to the R relation. Then in the ASK query, its result will be false iff all the nodes of T are reachable. Note that this reduction can be immediately adapted to reflect hardness for queries using CONSTRUCT or SELECT. □

From a practical point of view, and even if theoretically the problems have the same combined complexity as queries without recursion and are polynomial in data complexity, any implementation of the Algorithm 1 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* [19, 20]. This restriction is common when implementing recursive operators in other database languages, most notably in SQL [37], but also in graph databases [29], 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. All of this is defined in the following section.

4. Realistic Recursion in SPARQL

The concept of *linear recursion* has become popular in industry, as a restriction for fixed point operators in relational query languages, because it presents a good trade-off between the expressive power of recursive operators and their practical applicability. Let Q be the query WITH RECURSIVE t 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 $i+1$-th iteration, we only need the original dataset plus the tuples that were added to the temporary graph t in the i-th iteration. Considering that the temporary graph t might be of size comparable to the original dataset, linear queries save us from evaluating the query several times over an ever increasing dataset: instead we only need to take into account what was added in the previous iteration, which is generally much smaller.

Furthermore, most of the recursive extensions proposed for SPARLQ are linear: from property paths [2] to nSPARQL [3], SPARQLeR [14] or Trial [12], as well as our example. Unfortunately it is undecid-

able to check if a recursive query is linear [38] (under usual complexity-theoretic assumptions), so one needs to impose some 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

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

with H and DS a construct template and dataset clauses as usual, with $\boldsymbol{q}_{\text{out}}$ a linear recursive query, with $\boldsymbol{p}_{\text{base}}$ and $\boldsymbol{p}_{\text{rec}}$ group graph patterns, possibly with property paths, and where only $\boldsymbol{p}_{\text{rec}}$ is allowed to mention the IRI t. 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 CONSTRUCT H DS WHERE $\boldsymbol{p}_{\text{base}}$ and $\boldsymbol{q}_{\text{rec}}$ to denote the query CONSTRUCT H DS WHERE $\boldsymbol{p}_{\text{rec}}$, respectively.

For example, the query in example 3.1 is not linear, because the temporary IRI is used twice in the pattern. Nevertheless, it can be restated as the query from the Figure 4 that uses one level of nesting. This simple yet powerful syntax resembles the design choices taken in most SQL commercial systems supporting recursion (see e.g. [37]), and is also present in graph databases [29].

Continuing with the query of Figure 4, we note that the union in the first query can obviously be omitted, and is there only for clarity (our implementation supports queries where either $\boldsymbol{p}_{\text{base}}$ or $\boldsymbol{p}_{\text{rec}}$ is 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`[7].

[7] Interestingly, one can show that in this case the nesting in this query can be avoided, and indeed an equivalent non-nested recursive query is given in the online appendix.

Note that these queries are indeed linear, and thus we can perform the incremental evaluation that we have described at the beginning of the section, where we only recompute the answers of the recursive query over the result of the previous iteration. 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 [20] for a good survey). Our algorithm for query evaluation is presented in Algorithm 2.

Algorithm 2 Computing the answer for linear recursive c-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 $\mathsf{ans}(\boldsymbol{q}_{\text{out}}, D \cup \{\langle t, G_{\text{ans}}\rangle\})$

So what have we gained? By looking at Algorithm 2 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.

4.2. Limiting the depth of the recursion

In practice it could happen that an user may not be interested in having all the answers for a recursive query. Instead, the user could prefer to have only the answers until a certain number of iterations are per-

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp1 AS {
    CONSTRUCT { ?x ?u ?y }
    FROM NAMED <http://db.ing.puc.cl/temp1>
    WHERE{
        { ?x prov:wasRevisionOf ?z .
          ?x prov:wasGeneratedBy ?w .
          ?w prov:used ?z .
          ?w prov:wasAssociatedWith ?u }
    UNION
        {}}
}
WITH RECURSIVE http://db.ing.puc.cl/temp2 AS {
    CONSTRUCT { ?x ?u ?y }
    FROM NAMED <http://db.ing.puc.cl/temp1>
    FROM NAMED <http://db.ing.puc.cl/temp2>
    WHERE
        { GRAPH <http://db.ing.puc.cl/temp1> { ?x ?u ?y } }
    UNION {
        GRAPH <http://db.ing.puc.cl/temp1> { ?x ?u ?z }.
        GRAPH <http://db.ing.puc.cl/temp2> { ?z ?u ?y } }
}
SELECT ?x ?y
FROM <http://db.ing.puc.cl/temp>
WHERE {?x ?u ?y}
```

Fig. 4. Example of a linear recursion.

formed. We propose the following syntax for to restrict the depth of recursion to a user specified number k:

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

Here all the keywords are the same as when defining linear recursion, and $k \geqslant 1$ is a natural number. The semantics of such queries is defined using Algorithm 2, where the loop between steps 4 and 12 is executed precisely $k-1$ times.

As we said before, this extension is also useful for handling queries that include negation, or that create values by means of blanks or a BIND clause. If we fix the number of iterations of a recursive query, we can ensure that those kind of queries would end regardless of the existence of a fixed point.

5. Experimental Evaluation

In this section we will discuss how our implementation performs in practice and how it compares to alternative approaches that are supported by existing RDF Systems. Though our implementation has more expressive power, we will see that the response time of our approach is similar to the response time of existing approaches, and also our implementation outperforms the existing solutions in several use cases.

Technical details. Our implementation of linear recursive queries was carried out using the Apache Jena framework (version 3.7.0)[21] as an add-on to the ARQ SPARQL query engine. It allows the user to run queries either in main memory, or using disk storage when needed. The disk storage was managed by Jena TDB (version 1). 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[8].

Datasets. We test our implementation using three different datasets. The first one is Linked Movie Database (LMDB) [39], an RDF dataset containing information about movies and actors. The second dataset we use is a part of the YAGO ontology [40] and consists of all the facts that hold between instances. For the experiments the version from May 2018 was used. The last dataset was generated using the GMark benchmark [41], which is a benchmark designed to test the evaluation of property paths in RDF engines. All the experiments were run on a MacBook Pro with an Intel Core i5 2.6 GHz processor and 8GB of main memory. All the datasets can be found at https://alanezz.github.io/RecSPARQL/.

5.1. Evaluating real use cases

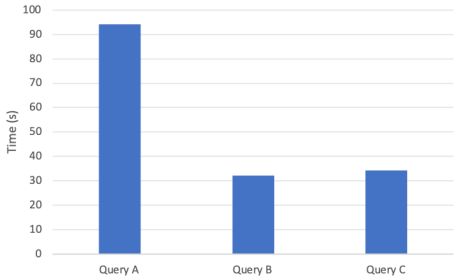
The first thing we do is to test our implementation against realistic use cases. As we have mentioned, we do not aim to obtain the fastest possible algorithms for these particular use cases (this is out of the scope of this paper), but rather aim for an implementation whose execution times are reasonable. For this, we took the LMDB and the YAGO datasets, and built a series of queries asking for relationships between entities. Since YAGO also contains information about movies, we have the advantage of being able to test the same queries over different datasets (their ontology differs). The specifications for each database can be found in the Figure 5. Note that the size is the one used by Jena TDB to store the datasets.

Graph	Number of triples	Size
LMDB	6147996	1.09 Gb
Yago	6215350	1.54 Gb

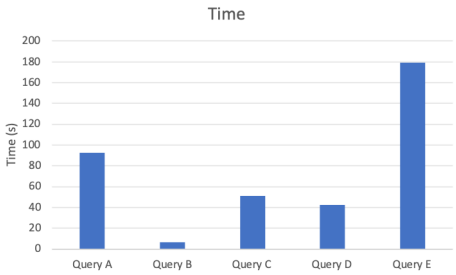
Fig. 5. Specifications for the LMDB and Yago datasets.

To the best of our knowledge, it is not possible to compare the full scope of our approach against other implementations. While it is true that our formalism is similar to the recursive part of 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 the recursion with common table expressions functionality, that is part of the SQL standard. OpenLink Virtuoso does have a *transitive closure* operator that can be used with its SQL engine, but this operator is quite limited in the sense that it can only compute transitivity when starting in a given IRI. Our queries were more general than this, and thus we could not compare them directly. For this reason, in this set of experiments we will only discuss about the practical applicability of the results.

Our round of experiments consists of three movie-related queries, which will be executed both on LMDB



(a) Query times on LMDB dataset



(b) Query times on YAGO dataset

QA	QB	QC
37349	1172	14568

(c) The number of output tuples for LMDB queries

QA	QB	QC	QD	QE
29930	85	3617	7	44

(d) The number of output tuples for Yago queries

Fig. 6. Running times and the number of output tuples for the three datasets.

[8]The implementation we use is available at https://alanezz.github.io/RecSPARQL/.

and YAGO, and two additional queries that are only run in YAGO, because LMDB does not contain this information. All of these queries are similar to that of Example 3.1 (precise queries are given in the appendix). The queries executed in both datasets are the following:

- QA: the first query 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 that of an Erdős number.
- QB: the second query returns all actors with a finite Bacon number such that all the collaborations were done in movies with the same director.
- QC: the third query 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 queries executed only in the YAGO dataset where the following:

- QD: the fourth query answers with the places where the city Berlin is located in from a transitive point of view, starting from Germany, then Europe and so forth.
- QE: the fifth query returns all the people who are transitively related to someone, through the `isMarriedTo` relation, living in the United States or some place located within the United States.

Note that QA, QD and QE are also expressible as property paths. To fully test recursive capabilities of our implementation we use another two queries, QB and QC, that apply various tests along the paths computing the Bacon number. Recall that the structure of queries QB and QC is similar to the query from Example 3.1 and cannot be expressed in SPARQL either.

The results of the evaluation can be found in Figures 6(a) and 6(b). As we can see the running times, although high, are reasonable considering the size of the datasets and the number of output tuples (Figures 6(c) and 6(d)). The query QE is the only query with a small size in its output and a high time of execution. This fact can be explained because the query is a combination of 2 property paths that required to instantiate 2 recursive graphs before computing the answer.

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

5.2. Comparison with Property Paths using the GMark benchmark

As mentioned previously, 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 show the results of comparing the execution of property paths queries in our implementation using our recursive language against the implementation of property paths in popular systems.

We used the GMark benchmark [41] to measure the running time of property paths queries using Recursive SPARQL, and to compare such times with respect to Apache Jena and Openlink Virtuoso.

Using the GMark benchmark, one can generate queries and datasets to test property paths, and one of its advantages is that the size of the datasets and the patterns described by the queries are parameterized by the user. The benchmark allowed us to generate 3 different graphs. The specifications for each graph can be found in Figure 7. We also generated 10 SPARQL queries that could have one or more property paths of different complexities. The queries can be found in the appendix. The results of evaluating such queries over the graphs are presented in the Figure 8.

Graph	Number of triples	Size
Graph 1	220564	271 mb
Graph 2	447851	535 mb
Graph 3	671712	605 mb

Fig. 7. Specifications for the graphs generated by GMark.

Note first that every property path query is easily expressible using linear recursion. With this observation in mind we must also remark that comparing the performance of the recursive implementation of property paths to the one in current systems is not fair as they are specialized in executing one particular 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

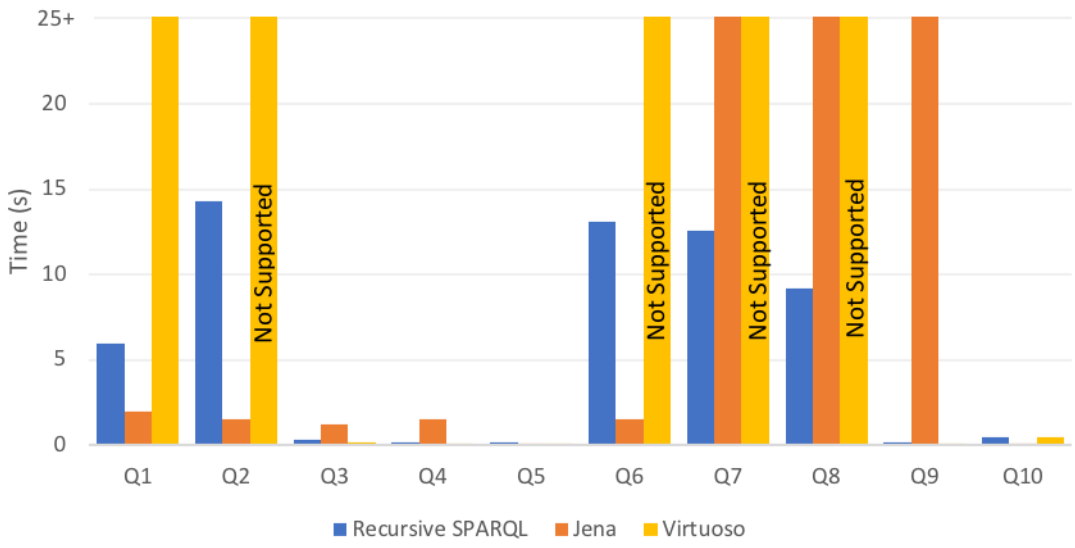


Fig. 8. Times for $G1$.

reason highly efficient systems like Virtuoso should run property paths queries much faster.

Our implementation can answer all the queries. Despite of the higher expressive power of our language, we are able to finish all the queries in a reasonable time, keeping the same order of magnitude than Apache Jena and Virtuoso and even having better running times in some of the queries.

Comparison with Virtuoso. Virtuoso cannot run queries 2, 6, 7 and 8, because the SPARQL engine requires an starting point for property paths queries, which was not possible to give for such queries. We can see that Virtuoso outperforms Jena and the Recursive implementation in almost all the queries that they can run, except for Query 1, where the running time goes beyond 25 seconds. As we will discuss later, this can be explained because of the semantic they use to evaluate property paths, which makes Virtuoso to have many duplicated answers. For the remaining queries, we can see that the execution time is almost equals.

Comparison with Jena. Apache Jena can also answer all queries. However, our recursive implementation is only clearly outperformed in Query 2 and Query 6. This is mainly because those queries have predicates with the form:

```
?x <:p1|:p2>* ?z
```

and our system is not optimized for working with unions of predicates. Remarkably, and even though all of the generated queries are relatively simple, our implementation reports a faster running time in half of the queries we test. Note that Q7, Q8 and Q9 have an answer time considerably worse in Jena than in our recursive implementation, where the time goes beyond the 25 seconds. We can only speculate that this is because the property paths has many paths of short length and because Apache Jena cannot manage properly the queries with two or more star triple patterns.

When we increase the size of the graph, the results have the same behaviour. It is also more evident which queries are easier and harder to evaluate for the existing systems. The result for the increased size of the graph can be found in the Figures 9 and 10.

Number of outputs. As we said before, one interesting thing that we note from the previous experiments is the time that Virtuoso took to answer the query Q1 in the three dataset. We suspected that this could be generated because Virtuoso generates many duplicate results, thus the output should be higher with respect to rec-SPARQL and Jena. We count the number of outputs for the queries ran over the first graph. The results can be seen in Figure 11.

The first thing to note is that we avoid duplicate answers in our language, mainly because of the `UNION` operator used in lineal recursion, which deletes the duplicates answers. Also we do not consider paths of

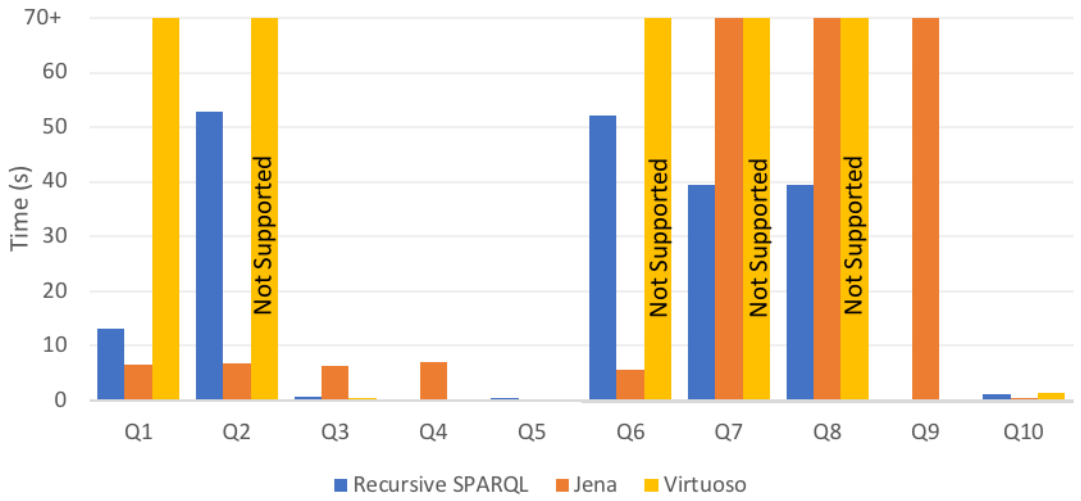


Fig. 9. Times for $G2$.

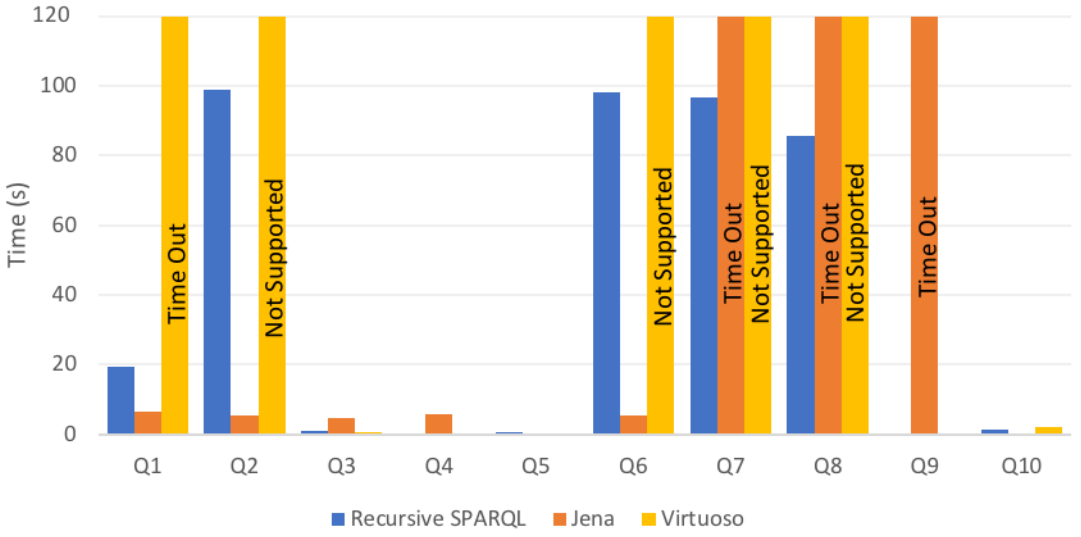


Fig. 10. Times for $G3$.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
RecSPARQL	24723	3964	1455	9	169	3964	2604	126	2	906
Jena	103814	90398	89128	198	802	90398	10838	94638	89523	5373
Virtuoso	1849915	-	2267	198	804	-	-	-	454	6345

Fig. 11. Number of outputs for the GMark queries over the first graph.

length 0, because those results do not give us any relevant information about the answer. Then we can see that Apache Jena has always more results than us, this is mainly because they consider paths of length 0. We did not rewrite the queries because we wanted keep them as close as possible to the benchmark. In Virtuoso one needs an starting point for property path queries, then this system does not consider paths of

length 0 and for that reason in some queries they have less outputs than Apache Jena. However, in most of the queries they give more results than Apache Jena and RecSPARQL, because they give many duplicate answers. This happen mainly in the first query, which is the simpler one. They give several duplicate answers and it causes the answer time to be considerably worse. The same effect happen for the 2 bigger graphs. This results can be found in Figures 12 and 13.

5.3. Limiting the number of iterations

In section 4.2 we presented a way of limiting the depth of the recursion. We argue that this functionality should find good practical uses, because 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 wast majority of use cases we will be interested in using property paths only up to depth four or five.
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 concatenation 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 (as a reference we include the rewritings of the query QA, QB and QC from Subsection 5.1 using only SPARQL operators in the online appendix). The second reason we would like to argue for is that, when implemented using Algorithm 2, 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.

We substantiate this claim by running two rounds of experiments on LMDB and YAGO datasets, using queries QA, QB and QC from Subsection 5.1 and running them for an increasing number of steps. We evaluate each of the queries using Algorithm 2 and run it for a fixed number of steps until the algorithm saturates. Then we use a SPARQL rewriting of a recursive query where the depth of recursion is fixed and evaluate it in Jena and Virtuoso.

Figure 14 shows the results over LMDB and Figure 15 shows the results over YAGO. 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. Note that we did not run the second query over the Yago dataset, because it ends in two iterations, and it would not show any trend. We also did not run queries QD and QE. Query QD was timing out also after two iterations, and query QE is composed by two property paths, then there is not a straightforward way to transform it in a query with unions.

6. Conclusions and looking ahead

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 the majority of interesting use cases and show how to implement them as an extension to Jena framework. We then test on real world datasets to show that, although very expressive, these queries run in reasonable time even on a machine with limited computational resources. Additionally, we also include the variant of the recursion operator that runs the recursive query for a limited number of steps and show that the proposed implementation outperforms equivalent queries specified using only SPARQL 1.1 operators.

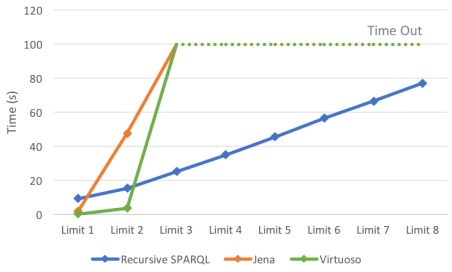
Given that recursion can express many requirements outside of the scope of SPARQL 1.1, coupled with the fact that implementing the recursive operator on top of existing SPARQL engines does not require to change their core functionalities, allows us to make a strong case for including recursion in the future iterations of the SPARQL standard. Of course, such an expressive recursive operator is not expected to beat specific algorithms for smaller fragments such as property paths. But nothing prevents the language to have both a syntax for property paths and also for recursive queries, with different algorithms for each operator. But, on the other hand, systems looking for more lightweight al-

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
RecSPARQL	50190	8153	3188	7	345	8153	6116	308	4	2134
Jena	208437	181043	178465	409	1547	181043	16730	189628	179168	11022
Virtuoso	3624482	-	5256	409	1561	-	-	-	968	13002

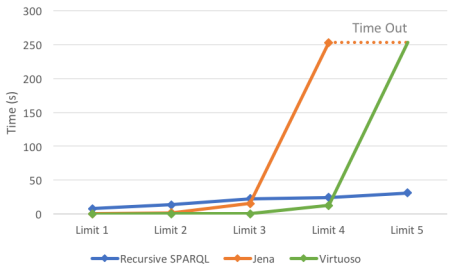
Fig. 12. Number of outputs for the GMark queries over the second graph.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
RecSPARQL	74967	12015	4719	17	533	12015	16040	487	4	2942
Jena	311559	270506	266777	766	2674	270506	-	-	-	15951
Virtuoso	-	-	7743	766	2716	-	-	-	1384	18726

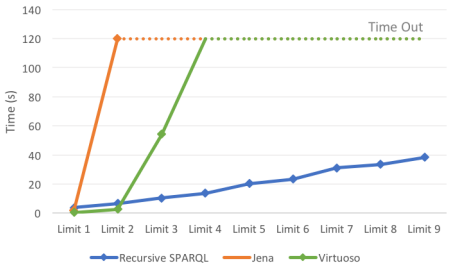
Fig. 13. Number of outputs for the GMark queries over the third graph.



(a) Query Q_A

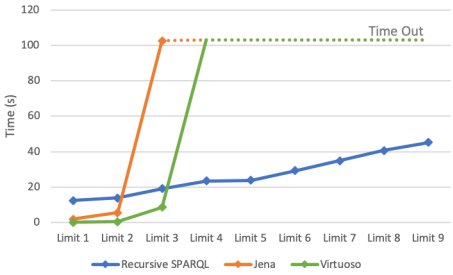


(b) Query Q_B

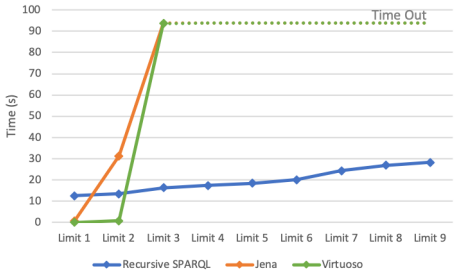


(c) Query Q_C

Fig. 14. Limiting the number of iterations for the evaluation of Q_A, Q_B and Q_C over LMDB.



(a) Query Q_A



(b) Query Q_C

Fig. 15. Limiting the number of iterations for the evaluation of Q_A and Q_C over Yago.

ternatives may prefer to eschew algorithms for property paths and just compile everything into our recursive operator.

There are several other areas where a recursive operator should bring immediate impact. To begin with, it has been shown that a wide fragment of recursive SHACL constraints can be compiled into recursive SPARQL queries [42], and a similar result should hold for ShEx constraints [43]. Another interesting direction is managing ontological knowledge. Indeed, it was shown that even a mild form of recursion is sufficient to capture RDFS entailment regimes [3] or OWL2 QL

entailment [44], and it stands open to which extent can rec-SPARQL help us capture more complex ontologies, and evaluate them efficiently. Furthermore, rec-SPARQL may also be used for other applications such as Graph Analytics or Business Intelligence.

Looking ahead, there are several directions we plan to explore. We believe that the connection between recursive SPARQL and RDF shape schemas should be pursued further, and so is the connection with more powerful languages for ontologies. There is also the subject of finding the best semantics for recursive SPARQL queries involving non-monotonic definitions. Stable model semantics may or may not be the best option, and even if it is, it would be interesting to see if one can obtain a good implementation by leveraging techniques developed for logic programming, or provide tools to compile recursive SPARQL queries into a logic program. Regarding blanks and numbers, perhaps one can also find a reasonable fragment, or a reasonable extension to the semantics of recursive queries, that can deal with numbers and blanks, but that can still be evaluated under the good properties we have showcase for linear recursion.

References

[1] P. Hitzler, M. Krotzsch and S. Rudolph, *Foundations of semantic web technologies*, CRC Press, 2011.

[2] S. Harris and A. Seaborne, SPARQL 1.1 query language, *W3C Recommendation* **21** (2013).

[3] J. Pérez, M. Arenas and C. Gutierrez, nSPARQL: A navigational language for RDF, *J. Web Sem.* **8**(4) (2010), 255–270.

[4] P. Barceló, J. Pérez and J.L. Reutter, Relative Expressiveness of Nested Regular Expressions, in: *AMW*, 2012, pp. 180–195.

[5] P. Barceló, G. Fontaine and A.W. Lin, Expressive Path Queries on Graphs with Data, in: *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer, 2013, pp. 71–85.

[6] P. Bourhis, M. Krötzsch and S. Rudolph, How to best nest regular path queries, in: *Informal Proceedings of the 27th International Workshop on Description Logics*, 2014.

[7] V. Fionda, G. Pirrò and M.P. Consens, Extended property paths: Writing more SPARQL queries in a succinct way, in: *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[8] V. Fionda, G. Pirrò and C. Gutierrez, N auti lod: A formal language for the web of data graph, *ACM Transactions on the Web (TWEB)* **9**(1) (2015), 5.

[9] V. Fionda and G. Pirrò, Explaining graph navigational queries, in: *European Semantic Web Conference*, Springer, 2017, pp. 19–34.

[10] K. Anyanwu and A.P. Sheth, ρ-Queries: enabling querying for semantic associations on the semantic web, in: *12th International World Wide Web Conference (WWW)*, 2003.

[11] M. Bienvenu, D. Calvanese, M. Ortiz and M. Simkus, Nested Regular Path Queries in Description Logics, in: *KR 2014, Vienna, Austria, July 20-24, 2014*, 2014.

[12] L. Libkin, J. Reutter and D. Vrgoč, Trial for RDF: adapting graph query languages for RDF data, in: *PODS*, ACM, 2013, pp. 201–212.

[13] L. Libkin, J.L. Reutter, A. Soto and D. Vrgoč, TriAL: A Navigational Algebra for RDF Triplestores, *ACM Trans. Database Syst.* **43**(1) (2018), 5–1546.

[14] K.J. Kochut and M. Janik, SPARQLeR: Extended SPARQL for semantic association discovery, in: *The Semantic Web: Research and Applications*, Springer, 2007, pp. 145–159.

[15] P. Missier and Z. Chen, Extracting PROV provenance traces from Wikipedia history pages, in: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013, pp. 327–330.

[16] W3C, PROV Model Primer, 2013.

[17] W3C, PROV-O: The PROV Ontology, 2013.

[18] E.V. Kostylev, J.L. Reutter and M. Ugarte, CONSTRUCT Queries in SPARQL, in: *ICDT*, 2015, pp. 212–229.

[19] S. Abiteboul, R. Hull and V. Vianu, *Foundations of Databases*, Addison-Wesley, 1995.

[20] T.J. Green, S.S. Huang, B.T. Loo and W. Zhou, Datalog and Recursive Query Processing., *Foundations and Trends in Databases* **5**(2) (2013), 105–195.

[21] The Apache Jena Manual, 2015.

[22] A. Gubichev, S.J. Bedathur and S. Seufert, Sparqling kleene: fast property paths in RDF-3X, in: *GRADES*, 2013.

[23] N. Yakovets, P. Godfrey and J. Gryz, WAVEGUIDE: Evaluating SPARQL Property Path Queries, in: *EDBT 2015*, 2015, pp. 525–528.

[24] Open Link Virtuoso, 2015.

[25] N. Yakovets, P. Godfrey and J. Gryz, Evaluation of SPARQL Property Paths via Recursive SQL, in: *AMW*, 2013.

[26] M. Atzori, Computing Recursive SPARQL Queries, in: *ICSC*, 2014, pp. 258–259.

[27] 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.

[28] J.L. Reutter, A. Soto and D. Vrgoč, Recursion in SPARQL, in: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, 2015, pp. 19–35.

[29] M. Consens and A.O. Mendelzon, GraphLog: A visual formalism for real life recursion, in: *9th ACM Symposium on Principles of Database Systems (PODS)*, 1990, pp. 404–416.

[30] F. Alkhateeb, J.-F. Baget and J. Euzenat, Extending SPARQL with regular expression patterns (for querying RDF), *J. Web Sem.* **7**(2) (2009), 57–73.

[31] L. Libkin, *Elements of Finite Model Theory*, Springer, 2004.

[32] R. Angles and C. Gutierrez, The multiset semantics of SPARQL patterns, in: *International semantic web conference*, Springer, 2016, pp. 20–36.

[33] M. Kaminski and E.V. Kostylev, Beyond well-designed SPARQL, in: *19th International Conference on Database Theory (ICDT 2016)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[34] I. Niemelä, Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm, *Ann. Math. Artif. Intell.* **25**(3–4) (1999), 241–273. doi:10.1023/A:1018930122475. https://doi.org/10.1023/A:1018930122475.

[35] J. Pérez, M. Arenas and C. Gutierrez, Semantics and complexity of SPARQL, *ACM Transactions on Database Systems* **34**(3) (2009).
[36] M.Y. Vardi, On the complexity of bounded-variable queries, in: *PODS*, Vol. 95, 1995, pp. 266–276.
[37] PostgreSQL documentation.
[38] H. Gaifman, H. Mairson, Y. Sagiv and M.Y. Vardi, Undecidable optimization problems for database logic programs, *Journal of the ACM (JACM)* **40**(3) (1993), 683–713.
[39] Linked movie database.
[40] YAGO: A High-Quality Knowledge Base.
[41] G. Bagan, A. Bonifati, R. Ciucanu, G.H.L. Fletcher, A. Lemay and N. Advokaat, gMark: Schema-Driven Generation of Graphs and Queries, *IEEE Transactions on Knowledge and Data Engineering* **29**(4) (2017), 856–869.
[42] J. Corman, F. Florenzano, J.L. Reutter and O. Savković, Validating SHACL constraints over a SPARQL endpoint, in: *International Semantic Web Conference (to appear)*, Springer, 2019.
[43] I. Boneva, J.E.L. Gayo and E.G. Prud'hommeaux, Semantics and validation of shapes schemas for RDF, in: *International Semantic Web Conference*, Springer, 2017, pp. 104–120.
[44] S. Bischof, M. Krötzsch, A. Polleres and S. Rudolph, Schema-agnostic query rewriting in SPARQL 1.1, in: *International Semantic Web Conference*, Springer, 2014, pp. 584–600.

Appendix. Appendix

Queries in Section 4.1

Query from Subsection 4.1 stated without nesting:

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp AS {
    CONSTRUCT {?x ?u ?y}
    FROM NAMED <http://db.ing.puc.cl/temp>
    WHERE{{
        ?x prov:wasRevisionOf ?z .
        ?x prov:wasGeneratedBy ?w .
        ?w prov:used ?z .
        ?w prov:wasAssociatedWith ?u}
    UNION{
        ?x prov:wasRevisionOf ?z .
        ?x prov:wasGeneratedBy ?w .
        ?w prov:used ?z .
        ?w prov:wasAssociatedWith ?u .
        GRAPH <http://db.ing.puc.cl/temp> {?z ?u ?y}}}
}
SELECT ?x ?y
FROM <http://db.ing.puc.cl/temp>
WHERE ?x ?u ?y
```

Queries from Subsection 5.1

The query Q_A is represented by the following recursive query:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS{
    CONSTRUCT {<http://data.linkedmdb.org/resource/actor/29539>
    <http://relationship.com/collab> ?act}
    FROM NAMED <http://db.ing.puc.cl/temp>
FROM <Quad.defaultGraphIRI>
WHERE {
{?mov <http://data.linkedmdb.org/resource/movie/actor>
<http://data.linkedmdb.org/resource/actor/29539> .
?mov <http://data.linkedmdb.org/resource/movie/actor> ?act}
UNION {
{?mov <http://data.linkedmdb.org/resource/movie/actor> ?act1} .
{?mov <http://data.linkedmdb.org/resource/movie/actor> ?act} .
GRAPH <http://db.ing.puc.cl/temp>
{<http://data.linkedmdb.org/resource/actor/29539>
<http://relationship.com/collab> ?act1}}
}
}
SELECT ?z FROM NAMED <http://db.ing.puc.cl/temp>
WHERE {GRAPH <http://db.ing.puc.cl/temp>
{<http://data.linkedmdb.org/resource/actor/29539>
<http://relationship.com/collab> ?z}}
```

The following is the formulation of the query Q_B:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
{
CONSTRUCT {<http://data.linkedmdb.org/resource/actor/29539> ?dir ?act}
FROM NAMED <http://db.ing.puc.cl/temp>
FROM <Quad.defaultGraphIRI>
WHERE
{
{?mov <http://data.linkedmdb.org/resource/movie/actor>
<http://data.linkedmdb.org/resource/actor/29539> .
?mov <http://data.linkedmdb.org/resource/movie/actor> ?act .
?mov <http://data.linkedmdb.org/resource/movie/director> ?dir}
UNION
{{?mov <http://data.linkedmdb.org/resource/movie/director> ?dir} .
{?mov <http://data.linkedmdb.org/resource/movie/actor> ?act1} .
{?mov <http://data.linkedmdb.org/resource/movie/actor> ?act} .
GRAPH <http://db.ing.puc.cl/temp>
{<http://data.linkedmdb.org/resource/actor/29539> ?dir ?act1}}
}
}
SELECT ?y ?z FROM NAMED <http://db.ing.puc.cl/temp>
WHERE {GRAPH <http://db.ing.puc.cl/temp>
{<http://data.linkedmdb.org/resource/actor/29539> ?y ?z}}
```

The following is the formulation of the query Q_C:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
{
CONSTRUCT {<http://data.linkedmdb.org/resource/actor/29539>
<http://relationship.com/collab> ?act}
FROM NAMED <http://db.ing.puc.cl/temp>
FROM <Quad.defaultGraphIRI>
WHERE
{
{?mov <http://data.linkedmdb.org/resource/movie/actor>
<http://data.linkedmdb.org/resource/actor/29539> .
?mov <http://data.linkedmdb.org/resource/movie/actor> ?act .
?mov <http://data.linkedmdb.org/resource/movie/director> ?dir .
?dir <http://data.linkedmdb.org/resource/movie/director_name> ?x .
?y <http://data.linkedmdb.org/resource/movie/actor_name> ?x}
UNION
{{?mov <http://data.linkedmdb.org/resource/movie/director> ?dir} .
{?dir <http://data.linkedmdb.org/resource/movie/director_name> ?x} .
{?y <http://data.linkedmdb.org/resource/movie/actor_name> ?x} .
{?mov <http://data.linkedmdb.org/resource/movie/actor> ?act1} .
{?mov <http://data.linkedmdb.org/resource/movie/actor> ?act} .
GRAPH <http://db.ing.puc.cl/temp>
{<http://data.linkedmdb.org/resource/actor/29539>
<http://relationship.com/collab> ?act1}}
}
}
```

```
SELECT ?z FROM NAMED <http://db.ing.puc.cl/temp>
WHERE {GRAPH <http://db.ing.puc.cl/temp>
{<http://data.linkedmdb.org/resource/actor/29539>
<http://relationship.com/collab> ?z}}
```

The following is the formulation of the query Q_D:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
{
CONSTRUCT {
  <http://yago-knowledge.org/resource/Berlin>
  <http://yago-knowledge.org/resource/isLocatedIn> ?x1
  }
  FROM NAMED <http://db.ing.puc.cl/temp>
  FROM <urn:x-arq:DefaultGraph>
  WHERE {
    {
      <http://yago-knowledge.org/resource/Berlin>
      <http://yago-knowledge.org/resource/isLocatedIn> ?x1
    }
    UNION
    {
      ?y <http://yago-knowledge.org/resource/isLocatedIn> ?x1 .
      GRAPH <http://db.ing.puc.cl/temp> {
        <http://yago-knowledge.org/resource/Berlin>
        <http://yago-knowledge.org/resource/isLocatedIn> ?y
      }
    }
  }
}
SELECT * FROM NAMED <http://db.ing.puc.cl/temp>
FROM <urn:x-arq:DefaultGraph>
WHERE {
  ?z <http://yago-knowledge.org/resource/dealsWith> ?v .
  GRAPH <http://db.ing.puc.cl/temp> {
  ?x ?y ?z }
}
```

The following is the formulation of the query Q_E:

```
WITH RECURSIVE http://db.ing.puc.cl/temp AS
{
CONSTRUCT {?x0 <http://yago-knowledge.org/resource/isMarriedTo> ?x1}
FROM NAMED <http://db.ing.puc.cl/temp>
FROM <urn:x-arq:DefaultGraph>
WHERE {
{ ?x0 <http://yago-knowledge.org/resource/isMarriedTo> ?x1 .
  ?x1 <http://yago-knowledge.org/resource/owns> ?y }
UNION
{ ?x0 <http://yago-knowledge.org/resource/isMarriedTo> ?y .
  GRAPH <http://db.ing.puc.cl/temp> {
    ?y <http://yago-knowledge.org/resource/isMarriedTo> ?x1
```

```
  }
}
}
}
WITH RECURSIVE http://db.ing.puc.cl/temp2 AS
{
CONSTRUCT {
  ?x0 <http://yago-knowledge.org/resource/isLocatedIn>
  <http://yago-knowledge.org/resource/United_States>
}
FROM NAMED <http://db.ing.puc.cl/temp2>
FROM <urn:x-arq:DefaultGraph>
WHERE {
{
  ?x0 <http://yago-knowledge.org/resource/isLocatedIn>
  <http://yago-knowledge.org/resource/United_States>
}
UNION
{ ?x0 <http://yago-knowledge.org/resource/isLocatedIn> ?y .
  GRAPH <http://db.ing.puc.cl/temp2> {
    ?y <http://yago-knowledge.org/resource/isLocatedIn>
    <http://yago-knowledge.org/resource/United_States>
  }
}
}
}
SELECT *
FROM NAMED <http://db.ing.puc.cl/temp>
FROM NAMED <http://db.ing.puc.cl/temp2>
FROM <urn:x-arq:DefaultGraph>
WHERE {
  ?z1 <http://yago-knowledge.org/resource/owns> ?x2 .
  GRAPH <http://db.ing.puc.cl/temp> { ?x1 ?y1 ?z1 } .
  GRAPH <http://db.ing.puc.cl/temp2> { ?x2 ?y2 ?z2 }
}
```

Queries from Subsection 5.2

The following are the queries generated by the GMark benchmark:

```
PREFIX : <http://example.org/gmark/>
SELECT * WHERE { ?x0 (:p16/^:p16) ?x1 . ?x1 (:p16/^:p16)* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((:p23/^:p23)|(:p25/^:p23)) ?x1 .
?x1 ((:p23/^:p23)|(:p25/^:p23))* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 (:p25/^:p25) ?x1 . ?x1 (:p25/^:p25)* ?x2 }

PREFIX : <http://example.org/gmark/>
```

```
SELECT *  WHERE { ?x0 (^:p22/:p16)* ?x1 . ?x1 (^:p19/:p20) ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 (:p0/:p22/^:p23) ?x1 . ?x1 (:p24/^:p24)* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((:p23/^:p23)|(:p25/^:p23)) ?x1 .
?x1 ((:p23/^:p23)|(:p25/^:p23))* ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((^:p15/:p18)|(^:p18/:p15))* ?x1 .
?x1 ((^:p15/:p8/^:p13)|(^:p15/:p8/^:p14)) ?x2 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 ((:p21/^:p21)|(:p21/^:p22)) ?x1 .
?x1 ((:p21/^:p21)|(:p21/^:p22))* ?x2 .
?x2 (:p16/^:p21)* ?x3 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 (^:p23/:p24) ?x1 .
?x1 (^:p23/:p24)* ?x4 .
?x0 (^:p17/:p21)* ?x2 .
?x0 (^:p25/:p25)* ?x3 }

PREFIX : <http://example.org/gmark/>
SELECT *  WHERE { ?x0 (:p16/^:p23) ?x1 .
?x1 (:p24/^:p24)* ?x2 .
?x2 (:p23/^:p23)* ?x3 }
```

The same queries written in Recursive SPARQL can be found at https://alanezz.github.io/RecSPARQL.

Queries from Subsection 4.2

The following is SPARQL rewriting of the query Q1 computing Bacon number of length at most 5:

```
SELECT ?act WHERE{{?mov
<http://data.linkedmdb.org/resource/movie/actor>
<http://data.linkedmdb.org/resource/actor/29539> .
?mov <http://data.linkedmdb.org/resource/movie/actor> ?act}

UNION { ?mov <http://data.linkedmdb.org/resource/movie/actor>
<http://data.linkedmdb.org/resource/actor/29539> .
?mov <http://data.linkedmdb.org/resource/movie/actor> ?act2  .
?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act}

UNION {?mov <http://data.linkedmdb.org/resource/movie/actor>
<http://data.linkedmdb.org/resource/actor/29539> .
?mov <http://data.linkedmdb.org/resource/movie/actor> ?act3  .
?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act3 .
?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
```

```
?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act  }

UNION {?mov <http://data.linkedmdb.org/resource/movie/actor>
<http://data.linkedmdb.org/resource/actor/29539> .
?mov <http://data.linkedmdb.org/resource/movie/actor> ?act4  .
?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act4 .
?mov2 <http://data.linkedmdb.org/resource/movie/actor> ?act3 .
?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act3 .
?mov3 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
?mov4 <http://data.linkedmdb.org/resource/movie/actor> ?act2 .
?mov4 <http://data.linkedmdb.org/resource/movie/actor> ?act  }}
```

Other rewritings are similar and can be found at https://alanezz.github.io/RecSPARQL.