Semantic Web 12 (2021) 711–740
DOI 10.3233/SW-200401
IOS Press

Recursion in SPARQL

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

[a] *Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile and IMFD Chile, Chile*
E-mail: jreutter@ing.puc.cl
[b] *Faculty of Engineering and Sciences Universidad Adolfo Ibáñez, Data Observatory Foundation and IMFD Chile, Chile*
E-mail: adrian.soto@uai.cl
[c] *Instituto de Ingeniería Matemática y Computacional, Pontificia Universidad Católica de Chile and IMFD Chile, Chile*
E-mail: dvrgoc@ing.puc.cl

Editor: Oscar Corcho, Universidad Politécnica de Madrid, Spain
Solicited review: Three anonymous reviewers

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, and the literature has already proposed several extensions to allow searching for more complex patterns.

We propose a rather different, but simpler approach: add a general purpose recursion operator directly to SPARQL. In this paper we provide a formal syntax and semantics for this proposal, study its theoretical properties, and develop algorithms for evaluating it in practical scenarios. We also show how to implement this extension as a plug-in on top of existing systems, and test its performance on several synthetic and real world datasets, ranging from small graphs, up to the entire Wikidata database.

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 [28]. 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 [26], 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. [10,11,17,20–22,40]), 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. [3,16] for a good overview of languages and extensions). However, none of these extensions have yet made it to the language, nor are they supported on any widespread SPARQL processor.

Looking for a recursive extension of SPARQL that can be easily implemented and adopted in practice, we

[*]Corresponding author. E-mail: adrian.soto@uai.cl.

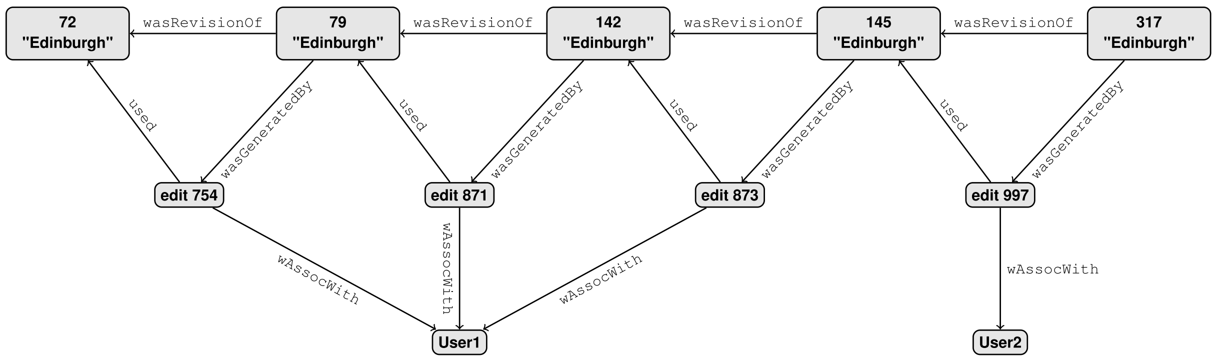


Fig. 1. RDF database of Wikipedia traces. The abbreviation `wAssocWith` is used instead of `wasAssociatedWith` and the `prov:`prefix is omitted from all the properties in this graph.

turn to the option of adding a general recursion operator to SPARQL in a similar way to what has been done in SQL. To illustrate the need for such an operator we consider the case of tracking provenance of Wikipedia articles presented by Missier and Chen in [34]. They use the PROV standard [48] 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 [49]. 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 $(A_1, \texttt{prov:wasRevisionOf}, A_2)$ 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 $(E, \texttt{prov:wasAssociatedWith}, U)$ if the edit E was made by the user U. A snapshot of the data, showing provenance of articles about Edinburgh, is depicted in Fig. 1.

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 Fig. 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 [11,32], 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 [31] 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 [31] 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. The main question we are trying to answer here is whether general purpose recursion is possible in SPARQL . We begin by showing what the general form of recursion looks like, what type of recursions we can support, the expressive power of the language, 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 [1,24]. We then argue 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 [46] 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. Finally, we experimentally evaluate the performance of our implementation. For this, we begin by evaluating recursive queries in the context of smaller datasets such as YAGO and LMDB. We then compare our implementation to Jena and Virtuoso when it comes to property paths, using the GMark graph database benchmark [9], allowing us to gauge the effect of dataset size and query selectivity on execution times. In order to see how our solution scales, we also use the wikidata database and test the performance of recursive queries in this setting.[1] We note that our main objective is not to find an optimal algorithms for evaluating recursion in SPARQL, but rather to show that recursion can be added in a non-intrusive way to the language, while still being capable of processing realistic workloads.

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 [25,52], or with some limitations that ensure they can be efficiently evaluated, most notable amongst them being Virtuoso [38]. The systems 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. From our analysis of expressive power we note that recursive queries we introduce are capable of expressing the transitive closure of any binary operator [31] and can thus be used to express property paths and any of their extensions [5,21,30,40,43]. 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 [51], or to allow recursion as a programming language construct [8,35], however none of them view recursion as a part of the language, but as an outside add-on. On the other hand, there is a wide body of work on implementing more powerful recursion in terms of datalog or other variants of logic programming (see e.g. [6,12,36]), but in this paper we are more interested in functionalities that can be added to SPARQL with little cost to systems in terms of extra software development.

Remark A preliminary version of this article was presented at the International Semantic Web Conference in 2015 [44]. The main contributions added to this manuscript not present in the conference version can be summarized as follows:

- *Proofs of all the results.* While the conference version of the paper provides only brief sketches of how the main results are proved, here we give complete proofs of all the stated theorems.
- *Convergence conditions for recursion.* We refine the analysis of when the recursion converges over RDF datasets, and fix a gap present in the conference version of the paper by introducing the notion of domain preserving queries.
- *Analysis of expressive power.* We analyse the expressive power of our language by comparing it to previous proposals, including Datalog [1], regular queries [43], and TriAL [32].
- *Support for negation.* We discuss possible extensions to the semantics in order to support negation inside recursive queries, or the `BIND` operator, which allows creating new values.
- *Extensive experimental evaluation.* The conference version of the paper only showcased the performance of our implementation over smaller datasets. Here we do a more robust analysis using the GMark graph database benchmark and the Wikidata dataset, in order to compare our approach to existing systems.

2. Preliminaries

Before defining our recursive operator we present the fragment of RDF and SPARQL we support. We first define what an RDF graph is, what operators we support and then we define their syntax and semantics.

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* and $\mathbf{L}$ an infinite set of *Literals*.[2] An *RDF triple* is a tuple

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

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

(s, p, o) from $(\mathbf{I} \cup \mathbf{L}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{L})$, where s is called the *subject*, p the *predicate*, and o the *object*. We recall the definition of IRI given in [7]. An IRI is an identifier of resources that extends the syntax of URIs to a much wider repertoire of characters for internationalization purposes. In the context of Semantic Web, IRIs are used for denoting resources.

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. Finally, we say that a dataset D is *contained* in a dataset D', and write $D \subseteq D'$ if (1) the default graph G in D is contained in the default graph G' in D', and (2) for every graph $\langle u, G\rangle$ in D, there is a graph $\langle u, G'\rangle$ in D' such that $G \subseteq G'$. Now we define the syntax and semantics of the fragment of SPARQL we will be working on through this paper, which is based on the one presented in [27].

SPARQL syntax. We assume a countable infinite set $\mathbf{V}$, called the set of variables, $\varnothing$ the unbound value and a set $\mathbf{F}$, called the set of functions, that consists in functions of the form $f : (\mathbf{I} \cup \mathbf{L} \cup \{\varnothing\})^n \rightarrow \mathbf{I} \cup \mathbf{L} \cup \{\varnothing\}$. The prefix "?" is used to denote variables (e.g., ?x).

The set of SPARQL queries is defined recursively as follows:

- An element of $(\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ is a query. Queries of this form are called *triple patterns*.
- If Q_1, Q_2 are queries, then:
  - ∗ $(Q_1$ UNION $Q_2)$ is a query called a UNION query.
  - ∗ $(Q_1$ AND $Q_2)$ is a query called an AND query.
  - ∗ $(Q_1$ OPTIONAL $Q_2)$ is a query called an OPTIONAL query.
  - ∗ $(Q_1$ MINUS $Q_2)$ is a query called a MINUS query.
- If Q is a query and g is a variable or an IRI, then (GRAPH g Q) is a query called a GRAPH query.
- If Q is a query and $\mathcal{X} \subset \mathbf{V}$ is a finite set of variables, then (SELECT $\mathcal{X}$ WHERE Q) is a query called a SELECT query.
- If Q is a query and φ is a SPARQL buit-in condition (see below), then (Q FILTER φ) is a query called a FILTER query.
- If Q is a query, $f : (\mathbf{I} \cup \mathbf{L} \cup \{\varnothing\})^n \rightarrow (\mathbf{I} \cup \mathbf{L} \cup \{\varnothing\})$ is a function in $\mathbf{F}$, and ?y, ?x$_1, \ldots,$?x$_n$ are variables such that ?y does not occur in Q nor in $\{$?x$_1, \ldots,$?x$_n\}$, then (Q BIND $f($?x$_1, \ldots,$?x$_n)$ AS ?y) is a BIND query.
- A SPARQL built-in condition (or simply a filter-condition) is defined recursively as follows:
  - ∗ An equality $t_1 = t_2$, where t_1, t_2 are elements of $\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}$, is a filter-condition.
  - ∗ If ?x is a variable then bound(?x) is a filter-condition.
  - ∗ A Boolean combination of filter-conditions (with operators $\wedge$, $\vee$, and $\neg$) is a filter-condition.

Notably, we have chosen to rule out the EXISTS keyword. This is because of a disagreement in the semantics of this operator (see the discussion in [27]). However, once this agreement is settled, extending all of our results for queries with EXISTS will be a straightforward task.

Mappings and mappings sets. Since we have all the syntax of the language, we have to discuss the semantics, but before, we need to introduce the notion of *mappings* and some operators over *sets of mappings*. A *SPARQL mapping* is a partial function $\mu : \mathbf{V} \rightarrow \mathbf{I} \cup \mathbf{L}$. Abusing notation, for a triple pattern t we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. Additionally, when $\mathcal{X}$ is a set of variables, and μ a mapping, we denote by $\mu_{\mathcal{X}}$ the mapping with the domain $\mathrm{dom}(\mu) \cap \mathcal{X}$, and such that $\mu_{\mathcal{X}}(?x) = \mu(?x)$, for every $?x$ in its domain.

Two mappings μ_1 and μ_2 are said to be compatible, denoted $\mu_1 \sim \mu_2$ if and only if for every common variable X holds $\mu_1(X) = \mu_2(X)$. Given two compatible mappings μ_1 and μ_2, the join of μ_1 and μ_2, denoted $\mu_1 \smile \mu_2$, is the mapping with domain $\mathrm{dom}(\mu_1) \cup \mathrm{dom}(\mu_2)$ that is compatible with μ_1 and μ_2.

Let Ω_1 and Ω_2 be two sets of mappings. Then, the operators $\bowtie$, $\cup$, $-$, and $⟕$ are defined over sets of mappings as follows:

$$\begin{aligned}\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \smile \mu_2 \mid \\ &\qquad \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\},\\ \Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\ \Omega_1 - \Omega_2 &= \{\mu_1 \mid \mu_1 \in \Omega_1, \text{ and there does}\\ &\qquad \text{not exist } \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\},\\ \Omega_1 ⟕ \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2).\end{aligned}$$

Given a query Q we write $\text{var}(Q)$ to denote the set of variables occurring in Q. We use this notation also for filter-conditions, i.e., $\text{var}(\varphi)$ are the variables occurring in the filter-condition φ.

SPARQL semantics. Let $D = \{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ be a dataset, G a graph among $G_0, G_1, \ldots, G_n$ and Q a SPARQL query. Then, the evaluation of Q over D with respect to G, denoted $[\![Q]\!]_G^D$, is the set of mappings recursively defined as follows:

- If t is a triple pattern, then $[\![t]\!]_G^D$ is the set of all mappings μ such that $\text{dom}(\mu) = \text{dom}(t)$ and $\mu(t) \in G$.
- If Q is (Q_1 AND Q_2) then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D \bowtie [\![Q_2]\!]_G^D$.
- If Q is (Q_1 UNION Q_2) then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D \cup [\![Q_2]\!]_G^D$.
- If Q is (Q_1 MINUS Q_2) then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D - [\![Q_2]\!]_G^D$.
- If Q is (Q_1 OPTIONAL Q_2) then $[\![Q]\!]_G^D = [\![Q_1]\!]_G^D ⟕ [\![Q_2]\!]_G^D$.
- If Q is (GRAPH g Q'), then $[\![Q]\!]_G^D = [\![Q']\!]_{\text{gr}_D(g)}^D$, if $g \in \mathbf{I}$, or $[\![Q]\!]_G^D = \bigcup_{u \in \mathbf{I}}([\![Q']\!]_{\text{gr}_D(u)}^D \bowtie \{g \mapsto u\})$ in case of $g \in \mathbf{V}$.
- If Q is (SELECT $\mathcal{X}$ WHERE Q_1) then $[\![Q]\!]_G^D = \{\mu_{\mathcal{X}} \mid \mu \in [\![Q_1]\!]_G^D\}$.
- If Q is (Q_1 FILTER φ) then $[\![Q]\!]_G^D$ is the set of mappings $\mu \in [\![Q_1]\!]_G$ such that $\mu(\varphi)$ is true.
- If Q is (Q_1 BIND $f(?\text{x}_1, \ldots, ?\text{x}_n)$ AS ?y) then let μ be a mapping and y_μ be the value $f(\mu(?\text{x}_1), \ldots, \mu(?\text{x}_n))$. Then, $[\![Q]\!]_G^D$ is the set of SPARQL mappings

$$\begin{aligned}&\{\mu \in [\![Q_1]\!]_G^D \mid y_\mu = \varnothing\}\\ &\quad \cup \big\{\mu \smile \{?\text{y} \mapsto y_\mu\} \mid\\ &\quad \mu \in [\![Q_1]\!]_G^D \text{ and } y_\mu \neq \varnothing\big\}.\end{aligned}$$

And finally, we define the semantics of built-in conditions. Let μ be a finite mapping from variables to elements in $\mathbf{I} \cup \mathbf{L}$, φ be a SPARQL filter-condition, and t_1, t_2 be elements in $\mathbf{V} \cup \mathbf{I} \cup \mathbf{L}$. Let $\nu : \mathbf{V} \cup \mathbf{I} \cup \mathbf{L} \to \mathbf{V} \cup \mathbf{I} \cup \mathbf{L}$ be the function defined as

$$\nu(t) = \begin{cases}\mu(t) & \text{if } t \in \text{dom}(\mu),\\ \varnothing & \text{if } t \in \mathbf{V} \setminus \text{dom}(\mu),\\ t & \text{if } t \notin \mathbf{V}.\end{cases}$$

The truth value of $\mu(\varphi)$ is defined recursively as follows:

- If φ is an equality $t_1 = t_2$ then:
  1. $\mu(\varphi)$ is error if $\nu(t_1) = \varnothing$ or $\nu(t_1) = \varnothing$.
  2. $\mu(\varphi)$ is true if $\nu(t_1) \neq \varnothing$, $\nu(t_1) \neq \varnothing$, and $\nu(t_1) = \nu(t_2)$.
  3. $\mu(\varphi)$ is false if $\nu(t_1) \neq \varnothing$, $\nu(t_1) \neq \varnothing$, and $\nu(t_1) \neq \nu(t_2)$.
- If φ is an expression of the form bound(?x) then $\mu(\varphi)$ is true if $?\text{x} \in \text{dom}(\mu)$. Otherwise, $\mu(\varphi)$ is false.
- If φ is a Boolean combination of conditions using operators $\wedge$, $\vee$ and $\neg$, then the truth value of $\mu(\varphi)$ is the usual for 3-valued logic (where error is interpreted as unknown).

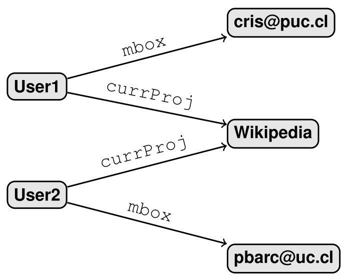
The CONSTRUCT operator The fragment of SPARQL graph patterns, as well as its generalisation to SELECT queries, has drawn most of the attention in the Semantic Web community. However, as the results of such queries are set of mappings instead of graphs, we shall use the CONSTRUCT operator in order to obtain a base for recursion. A SPARQL CONSTRUCT query, or *c-query* for short, is an expression

CONSTRUCT H WHERE Q,

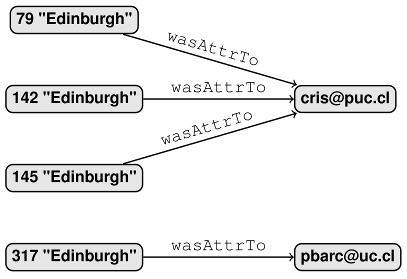
where H is a set of triples from $(\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V})$, called a *template*,[3] and Q is a SPARQL query.

The idea behind the CONSTRUCT operator is that the mappings resulting of evaluating Q over the dataset are used to construct an RDF graph according to the template H: for each such mapping μ, one replaces each variable $?x$ in H for $\mu(?x)$, and add the resulting triples to the answer. Since all the patterns in the template are triples we will end up with an RDF graph as

[3]For brevity we leave out FROM named constructs, and we leave the study of blanks in templates as future work.



(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.

desired. We illustrate how they work by means of an example.

Example 2.1. Let G be the graph in Fig. 1, and G_1 the graph in Fig. 2(a). Suppose that 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 that we have a dataset with the 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
}
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 $\boldsymbol{q}$ over D. In this case, it is depicted in Fig. 2. Formally, the *answer* $\mathsf{ans}(q, D)$ to a c-query of the form q = CONSTRUCT H WHERE Q over a dataset D with default graph G_0 is the RDF graph consisting of all triples in $\mu(H)$, for each mapping μ in $[\![Q]\!]^D_{G_0}$.

For readability, we will use Q to refer to SPARQL queries using the SELECT form, and $\boldsymbol{q}$ for c-queries whenever it is convenient to use this distinction. However, we often deal with queries that can be of either form. In this case, we use the notation $\mathsf{ans}(q, D)$ to speak of the answer of $\boldsymbol{q}$ over dataset D. it is defined as above if $\boldsymbol{q}$ is a c-query, and we define $\mathsf{ans}(q, D) = [\![\boldsymbol{q}]\!]^D_{G_0}$, for G_0 the default graph of D, when $\boldsymbol{q}$ is a query of the SELECT form.

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 [26].

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. [5,17,18,40]) and as a result several extensions have been brought forward to combat this problem [2,13,21,30,32,40]. 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 RDF documents.

3.1. A fixed point based recursive operator

We have decided to implement a different approach and define a more expressive 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 [31], 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 Fig. 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 Fig. 3.

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

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

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

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

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

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

When D_{LFP} exists and is a finite set, we say that the recursive query $\boldsymbol{q}$ *converges* over D.

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

Input: Query $\boldsymbol{q}$ of the form (1), dataset D
Output: Evaluation $\mathsf{ans}(\boldsymbol{q}, D)$ of $\boldsymbol{q}$ 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 $\mathsf{ans}(\boldsymbol{q}_2, D \cup \{\langle t, G_{\text{Temp}}\rangle\})$
```

In this definition, 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.[4]

To clarify Definition 3.2, we show how the temporary graph `<http://db.ing.puc.cl/temp>` evolves during the execution of the query from Fig. 3, when evaluated the graph in Fig. 1. The different values of temporary graph are show in Fig. 4. Here we call G^i_{Temp} the instance of G_{Temp} at the ith iteration of the loop presented in the Algorithm 1. We have two things to note: (1) G^0_{Temp} is an empty graph, and (2) since we are working with graphs, there are no duplicated triples. Finally, we have $G^3_{\text{Temp}} = G^4_{\text{Temp}}$, and thus we stop the loop at the fourth iteration.

Obviously, the semantics of recursive queries only makes sense as long as the required 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, we need to restrict the

[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.

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp AS {
    CONSTRUCT {?x ?u ?y}
    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
WHERE {
  GRAPH <http://db.ing.puc.cl/temp> {
    ?x ?u ?y
  }
}
```

Fig. 3. Example of a recursive query.

G^1_{Temp}

s	p	o
:72	:user1	:79
:79	:user1	:142
:142	:user1	:145
:145	:user2	:317

G^2_{Temp}

s	p	o
:72	:user1	:79
:79	:user1	:142
:142	:user1	:145
:145	:user2	:317
:72	:user2	:142
:79	:user2	:145

G^3_{Temp}

s	p	o
:72	:user1	:79
:79	:user1	:142
:142	:user1	:145
:145	:user2	:317
:72	:user2	:142
:79	:user2	:145
:72	:user2	:145

Fig. 4. The step-by-step evaluation of the recursive graph <http://db.ing.puc.cl/temp>.

language that can be applied to such inner queries.[5] We also discuss other possibilities to allow us using any operator we want.

3.2. Ensuring fixed point of queries

If we want to guarantee the termination of Algorithm 1, we need to impose two conditions. The first, and most widely studied, is that query q_1 must be *monotone*: a c-query q is monotone if for all pair of datasets D_1, D_2 where $D_1 \subseteq D_2$ it holds that $\mathsf{ans}(q, D_1) \subseteq \mathsf{ans}(q, D_2)$. However, we also need to impose that the recursive c-query q_1 preserves the domain: we say that a c-query q preserves the domain if there is a finite set S of IRIs such that, for every dataset D, the IRIs in $q(D)$ either come from S or are already present in D. Let us provide some insight about the need for these conditions.

[5]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.

Monotonicity The most typical example of a problematic non-monotonic behaviour is when we use negation 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 MINUS clause.

```
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT {?x ?y "a"}
  WHERE {
    { ?x ?y ?z } MINUS {
      GRAPH <http://db.puc.cl/temp> {
        { ?x ?y "a" }
      }
    }
  }
}
SELECT * WHERE {
  GRAPH <http://db.puc.cl/temp> {
    ?x ?y ?z
  }
}
```

Also consider a instance for the default graph with only one triple:

```
:s :p "b"
```

In the first iteration, the graph <temp> would have the triple:

```
:s :p "a"
```

but in the next iteration the graph <temp> will be empty because of the MINUS clause. Then, the <temp> graph will be alternating between an empty graph and a graph with the triple :s :p "a". 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 [4,29].

Preserving the domain The BIND clause allows us to generate new values that were not in the domain of the database before executing a recursive query. Since completely new values may be generated for the temporary graph at each iteration, this may also imply that a (finite) fixed point may not exists, even if the query is monotone.

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

```
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT {?x :number ?b}
  WHERE {
    { ?x :type :person .
      ?x :age ?a . BIND (?a AS ?b) }
    UNION {
      GRAPH <http://db.puc.cl/temp> {
        ?x :number ?aux .
        BIND(?aux + 1 AS ?b)
      }
    }
  }
}
SELECT * WHERE {
  GRAPH <http://db.puc.cl/temp>{
    ?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, and will thus never terminate adding new triples into the dataset.

On the other hand, it is easy to verify that the query from Example 3.3 is monotone. By the Knaster–Tarski theorem [45], a monotone query always has a fixed point, however, such a fixed point need not be a finite dataset. Indeed, the fixed point for the query in Example 3.3 would have to contain triples linking each person in the original dataset with all the numbers larger than her or his initial age. This would make the fixed point infinite and thus not be a valid RDF graph.

One way to ensure that a fixed point of a monotone query is necessarily finite, is to make the underlying domain over which it operates finite. For instance, in Example 3.3, we are assuming that the domain over which queries operate is the set of all possible triple over $\mathbf{I} \cup \mathbf{L}$, and not just the ones using IRIs and literals from the queried dataset. On the other hand, in the case of domain preserving queries, when considering the sequence $(D_i)_i$ from Definition 3.2, our monotone queries can only construct triples over a finite set of IRIs and literals (the initial dataset, plus another finite set), thus making the fixed point necessarily finite.

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.

Existence of a fixed point If we are working with queries that are both monotone and domain preserving, we can guarantee that the sequence $(D_i)_i$ from Definition 3.2 always converges to a well defined RDF dataset. More precisely, as an immediate consequence of the Knaster-Tarski theorem [45], we can obtain the following:

Proposition 3.1. *Let D be a dataset and $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ two monotone queries that are domain preserving, and let $\boldsymbol{q}$ = WITH RECURSIVE t AS $\{\boldsymbol{q}_1\}$ $\boldsymbol{q}_2$ be a recursive query. Then $\boldsymbol{q}$ converges over D, and we can use Algorithm 1 to evaluate $\boldsymbol{q}$.*

It is important to note that the query $\boldsymbol{q}_1$ need not be monotone over the domain of all possible RDF datasets in order for $\boldsymbol{q}$ to converge over D. Indeed, in order to apply the Knaster–Tarski theorem, it suffices that $\boldsymbol{q}_1$ is monotone over the datasets that appear in the sequence $(D_i)_i$ from Definition 3.2.

Next, we study which SPARQL queries are both domain preserving and monotone, in order to restrict the recursion to such queries.

3.3. Fragments where the recursion converges

We know that monotonicity and domain preservation allows us to define a class of recursive queries which will always have a least fixed point. The question then is: how to define a fragment of SPARQL that is both monotonic and domain preserving?

First, how can we guarantee that queries are monotonic? An easy option here is to simply disallow all the operators which can simulate negation such as OPTIONAL, MINUS, or negative FILTER conditions. Second, when it comes to guaranteeing that queries are domain preserving, we can simply prohibit them to use operators that can create new values such as BIND, or to use blanks in construct templates, queries or subqueries. This leads us to a first subclass of SPARQL queries for which can be used inside recursive queries.

Definition 3.3 (positive SPARQL and rec-SPARQL)**.** A SPARQL query is *positive* if it does not use any of the following operators:

1. It does not use operators OPTIONAL, MINUS, BIND;
2. Every construct (Q FILTER φ) is such that φ uses only equalities and positive boolean combinations using $\wedge$ and $\vee$; and
3. In every subquery (SELECT $\mathcal{X}$ WHERE Q) we have that Q is also positive.

Likewise, a positive c-query is a c-query using positive SPARQL in its definition. The language of positive rec-SPARQL comprises every positive SPARQL query, and also queries of the form

$$\text{WITH RECURSIVE } t \text{ AS } \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2, \tag{2}$$

where t is an IRI from $\mathbf{I}$, $\boldsymbol{q}_1$ is a positive c-query, and $\boldsymbol{q}_2$ is a positive rec-SPARQL query.

Given that positive SPARQL queries are both monotone and domain preserving, as an easy corollary of Proposition 3.1, we obtain the following:

Proposition 3.2. *If we have a recursive query $\boldsymbol{q}$ = WITH RECURSIVE t AS $\{\boldsymbol{q}_1\}$ $\boldsymbol{q}_2$, where $\boldsymbol{q}_1$ a positive query, and $\boldsymbol{q}_2$ is a positive rec-SPARQL query, then $\boldsymbol{q}$ converges over D.*

While positive queries do the trick, one might argue that they are quite restrictive. We can therefore wonder, whether it is possible to allow some form of negation, or the use of OPTIONAL?

In fact, literature has pointed out some more milder restrictions, that still do the trick. For instance, even though we disallow OPTIONAL, we note that our fragment of SPARQL is expressive enough to express rec-SPARQL queries in which $\boldsymbol{q}_1$ is given by positive SPARQL with well-designed optionals [40]. This is because for CONSTRUCT queries, the fragment we consider has been shown to contain queries defined by union of well designed graph patterns [31]).

The second, more expressive fragment to consider, loosens the restrictions we put on the use of negation. The idea is that bad behaviour of negation, such as the one shown in Example 3.2, only occurs when negation involves the same graph that is being constructed in the fixed point. Therefore, we will consider a fragment of rec-SPARQL queries which allow negation whenever it does not involve the graph being constructed in the recursive portion of the query.

Definition 3.4 (Stratified positive rec-SPARQL)**.** Stratified positive rec-SPARQL extends the language of positive rec-SPARQL by allowing queries of the form (2) above in which $\boldsymbol{q}_1$ can use constructs of the form (Q_1 MINUS Q_2), as long as every expression (GRAPH g Q) in Q_2 is such that g is an IRI different from t.

Essentially, with stratified positive rec-SPARQL we allow some degree of negation, but we need to make sure this negation does not involve the temporal graph under construction.

While stratified positive rec-SPARQL need not be monotone with respect to the domain of all possible RDF datasets, they are monotone in the context of the sequence $(D_i)_i$ from Definition 3.2. More precisely, any stratified positive rec-SPARQL of the form (2) is such that $\boldsymbol{q}_1$ is monotone with respect to the named graph t, meaning that, for datasets D and D' that only differ in the named graph t, if $\langle t, G\rangle$ is the graph named t in D, $\langle t, G'\rangle$ is the graph t in D', and $G \subseteq G'$, then $\mathsf{ans}(\boldsymbol{q}_1, D) \subseteq \mathsf{ans}(\boldsymbol{q}_1, D')$. We can then confirm that our semantics is well-defined for positive or stratified positive rec-SPARQL as an easy corollary of Proposition 3.1.

Proposition 3.3. *Let D be a dataset and $\boldsymbol{q}$ a stratified positive (or just positive) recursive c-query of the form* WITH RECURSIVE t AS $\{\boldsymbol{q}_1\}$ $\boldsymbol{q}_2$*. Then $\boldsymbol{q}$ converges over D.*

3.4. Expressive power

As a way to gauge the power of our language, we turn to the language *datalog*. Datalog (and ASP) has been used to pinpoint the expressive power of SPARQL (see e.g. [41]). For our case, it suffices to define positive Datalog, and Datalog with negation under stratified semantics.

A positive Datalog *program* Π consists of a finite set of rules of the form $S(\bar{x}) \leftarrow R_1(\bar{y}_1), \dots, R_m(\bar{y}_m)$, where $S, R_1, \dots, R_m$ are predicate symbols, $\bar{y}_1, \dots, \bar{y}_m$ are tuples in $\mathbf{V} \cup \mathbf{I}$ and $\bar{x}$ is a tuple of variables already appearing in $\bar{y}_1, \dots, \bar{y}_m$. In a rule of this form, S is said to appear in the head of the rule, and $R_1, \dots, R_m$ in the body of the rule. A predicate that occurs in the head of a rule is called *intensional* predicate. The rest of the predicates are called *extensional* predicates. Further, we assume that each program has a distinguished intensional predicate called the answer of Π, and denoted by Ans.

Let P be an intensional predicate of a positive Datalog program Π and I a set of predicates. For $i \geqslant 0$, $P^i_\Pi(I)$ denote the collection of facts about the intensional predicate P that can be deduced from I by at most i applications of the rules in Π. Let $P^\infty_\Pi(I)$ be $\bigcup_{i \geqslant 0} P^i_\Pi(I)$. Then, the *answer* $\Pi(I)$ of Π over I is $\text{Ans}^\infty_\Pi(I)$.

Datalog programs with negation extend positive datalog with rules of the form

$$S(\bar{x}) \leftarrow R_1(\bar{y}_1), \dots, R_m(\bar{y}_m),$$
$$\neg P_1(\bar{z}_1), \dots, \neg P_n(\bar{z}_n),$$

in which we require that every variable in $\bar{z}_1, \dots, \bar{z}_n$ is also in $\bar{y}_1, \dots, \bar{y}_m$. We are interested in datalog programs with *stratified negation*. The dependency graph of a program Π is a directed graph whose nodes are the predicates of Π and whose edges can be labelled with $+$ and $-$. There is a $+$ edge from predicate Q to predicate P, if Q occurs positively in the body of a rule ρ of Π and P is the predicate in the head of ρ. Likewise, there is an edge labelled with $-$ if Q occurs negated in the body of a rule the body of a rule ρ of Π and P is the predicate in the head of ρ. A program Π has *stratified negation* if one can partition the set of predicates into sets $C_1, \dots, C_\ell$, such that (1) for each edge from Q to P labelled with $+$ in the dependency graph of Π, if Q belongs to set C_i, then P belongs to C_j with $j \geqslant i$, and (2) for each edge from Q to P labelled with $-$ in the dependency graph of Π, if Q belongs to set C_i, then P belongs to C_j with $j > i$.

For positive datalog programs Π with stratified negation, we can compute the answer $\Pi(I)$ of Π over a set I of predicates as follows. Let $C_1, \dots, C_\ell$ be a partition satisfying the conditions for stratified negation. We compute sets $I_1, \dots, I_\ell$ of predicates, as follows. Let $I = I_0$. For each $1 \leqslant k \leqslant \ell$, let Π_k be the set of rules mentioning a predicate in C_k on their head. Notice that by definition of stratification, all predicates mentioned in the head or body of Π_k must belong to some $C_{k'}$ with $k' \leqslant k$. Then $I_k = I_{k-1} \cup \bigcup_{P \in C_k} P^\infty_{\Pi_k}(I_{k-1})$. Finally, the answer $\Pi(I)$ is $\text{Ans}^\infty_{\Pi_\ell}(I)$.

Since we are using Datalog to query RDF databases, we only focus on programs in which all extensional predicates are ternary predicates of the form T_g, with g an IRI. We can then represent each dataset D as a set I of predicates containing one ternary relation T_g per each graph in g. We can then treat datalog programs as c-queries: on input D, the constructed graph of a program Π contains all predicates in $\Pi(I)$, where I is the representation of the dataset D we have just defined. Abusing the notation, we will write $\Pi(D)$ to speak of this RDF graph.

In order to study the expressive power of rec-SPARQL, we first compare positive c-queries and stratified positive c-queries with datalog. It follows from Polleres and Wallner [41] (see Table 1) that our fragment of positive c-queries can be translated into positive datalog, in the following sense: For each c-query $\boldsymbol{q}$ one can construct a positive datalog program Π such that $\mathsf{ans}(\boldsymbol{q}, D) = \Pi(D)$ for every dataset D. Moreover, by inspecting the construction by Polleres and Wallner, one also gets that stratified positive *c*-queries can be translated into datalog with stratified negation, in the same terms. From these results, we further obtain that any positive or stratified positive rec-SPARQL query can be translated as well into positive datalog or datalog with stratified negation, respectively.

Proposition 3.4. *Let $\boldsymbol{q}$ be a positive (resp. stratified positive) rec-SPARQL query. Then one can construct a positive (resp. stratified positive) datalog program Π such that $\mathsf{ans}(\boldsymbol{q}, D) = \Pi(D)$ for every dataset D.*

Proof. The idea is to proceed inductively. For a query $\boldsymbol{q}$ of the form $\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2$, let Π_1 be the translation of $\boldsymbol{q}_1$, and let Ans_1 be the answer predicate of $\boldsymbol{q}_1$. Likewise, let Π_2 the translation of $\boldsymbol{q}_2$. Our program Π for $\boldsymbol{q}$ contains all rules in Π_1 and Π_2, plus the rule

$$T_t(x, y, z) \leftarrow \text{Ans}_1(x, y, z) \tag{3}$$

Where T_t is the predicate representing all triples in graph named t. This rule is positive, and maintains stratification. The answer predicate of Π is Ans_2. $\square$

We note that the other direction is currently open: we do not know if every stratified positive datalog program can be translated into stratified positive rec-

SPARQL. On the surface, and given the result that SPARQL can express all stratified positive datalog queries (see e.g. [31], it would appear that we can. However, rules in datalog programs may incur in all sort of simultaneous fixed points, as the dependency graph in programs need not be a tree. In standard datalog one can always flatten these simultaneous rules so that the resulting dependency graph is tree-shaped, but the only constructions we are aware of must incur in predicates with more than three positions, that we don't currently know how to store in the graphs constructed by rec-SPARQL.

On the other hand, when datalog programs disallow all sort of simultaneous fixed points, a translation is surely possible.

Proposition 3.5. *Let Π be a positive (resp. stratified positive) datalog program, and assume that the only cicles on the dependency graph of Π are self loops. Then one can build a positive (resp. stratified positive) rec-SPARQL query $\boldsymbol{q}$ such that $\Pi(D) = \mathsf{ans}(\boldsymbol{q}, D)$ for every dataset D.*

Proof. That non-recursive datalog can be expressed as a c-query already follows from previous work [31]. By examining that proof, we get the following result: for each predicate S in a non-recursive program Π, if we assume that for each rule in Π of the form

$$S(\bar{x}) \leftarrow R_1(\bar{y}_1), \dots, R_m(\bar{y}_m),$$
$$\neg P_1(\bar{z}_1), \dots, \neg P_n(\bar{z}_n),$$

mentioning S, we have that all collections of triples $R_1{}^{\infty}_{\Pi}(D), \dots, R_m{}^{\infty}_{\Pi}(D), P_1{}^{\infty}_{\Pi}(D), \dots, P_n{}^{\infty}_{\Pi}(D)$ are stored in named graphs $t_{R_1}, \dots, t_{R_m}, t_{P_1}, \dots, t_{P_n}$ (and assuming again for simplicity that these graphs are empty in D), then one can construct a c-query $\boldsymbol{q}_S$ such $\mathsf{ans}(\boldsymbol{q}_S, D)$ contains precisely $S^{\infty}_{\Pi}(D)$.

Thus, all that remains to do is to lift this result for datalog programs when the only recursion is given by rules of the form

$$R_i(\bar{x}) \leftarrow R_1(\bar{y}_1), \dots, R_m(\bar{y}_m),$$
$$\neg P_1(\bar{z}_1), \dots, \neg P_n(\bar{z}_n).$$

In this case, we let $\boldsymbol{q}_{R_i}$ be the query constructed as explained before, but we use the recursive c-query WITH RECURSIVE t_{R_i} AS $\{\boldsymbol{q}_{R_i}\}$ $\boldsymbol{q}'$, with $\boldsymbol{q}'$ the query `SELECT * WHERE GRAPH t {x y z}` that simply selects every triple from the constructed graph. □

While this result may sound narrow, it already gives us a tool to compare again some other recursive languages proposed for graphs and RDF. For example, the language of regular queries [43] is a datalog program whose only cicles in its dependency graph are self loops, so we immedately have that rec-SPARQL can express any Regular Query. Another language with a similar recursion structure is TriAL [32], and we can also use that rec-SPARQL can express any TriAl query. On the other hand, it does not give us much in terms of more expressive languages such as [6], for which the comparison would require more work.

3.5. *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 $\boldsymbol{q}$ using the SELECT result form, a tuple $\bar{a}$ of IRIs from $\mathbf{I}$ and a dataset D with default graph G_0, and asks whether $\bar{a}$ is in $\mathsf{ans}(\boldsymbol{q}, D)$. For CONSTRUCT queries, the problem CONSTRUCTQUERYANSWERING receives a recursive query $\boldsymbol{q}$ using 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 $\mathsf{ans}(\boldsymbol{q}, D)$.

Proposition 3.6. *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 [31] that the problem CONSTRUCTQUERYANSWERING is NP-complete for non recursive c-queries, and Pérez et al. show in [39] 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 [39] or construct queries [31]. 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($\boldsymbol{q}$) and CONSTRUCTQUERYANSWERING($\boldsymbol{q}$), 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 3.7. SELECTQUERYANSWERING($\boldsymbol{q}$) *and* CONSTRUCTQUERYANSWERING($\boldsymbol{q}$) *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 3.6, 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 [39] and [31]). 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. [47]). The problem is as follows. Consider 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 $(u(v)$, `:p`, `"C"`) to a named graph `gr:C` of the database instance.
- For each element $v \in T$, we add the triple $(u(v)$, `:p`, `"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 gr: <http://example.org/graph>
WITH RECURSIVE http://db.puc.cl/temp
AS {
  CONSTRUCT { ?z :p "C" }
  WHERE {
    { GRAPH gr:C { ?z :p "C" } }
    UNION {
      { ?x ?y ?z } .
      GRAPH <http://db.puc.cl/temp> {
        ?x :p "C" } .
      GRAPH <http://db.puc.cl/temp> {
        ?y :p "C" }
      }
    }
  }
}
ASK WHERE {
  GRAPH gr:T {
    ?x :p "T"
  } .
  GRAPH <http://db.puc.cl/temp> {
    ?x :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 none of the nodes in T are reachable. Note that this reduction can be immediately adapted to reflect hardness for queries using CONSTRUCT or SELECT. $\square$

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* [1,24]. This restriction is common when implementing recursive operators in other database languages, most notably in SQL [42], but also in graph databases [18], 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

Having defined our recursive language, the next step is to outline a strategy for implementing it inside of a SPARQL system. In this section we show how this can be done by focusing on *linear* queries. While the use of linear queries is well established in the SQL context, here we show how this approach can be lifted to SPARQL. In doing so, we will argue that not only do linear queries allow for much faster evaluation algorithms than generic recursive queries, but they also contain many queries of practical interest. Additionally, we outline some alternatives of recursion which can support the use of negation or BIND operators.

4.1. Linear recursive queries

The concept of *linear recursion* is widely used 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.

Commonly defined for logic programs, the idea of linear queries is that each recursive construct can refer to the recursive graph or predicate being constructed only once. To achieve this, 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

$$\begin{aligned}&\text{WITH RECURSIVE } t \text{ AS } \{\\ &\qquad \text{CONSTRUCT } H \qquad\qquad (4)\\ &\qquad \text{WHERE } \boldsymbol{p}_{\text{base}} \text{ UNION } \boldsymbol{p}_{\text{rec}} \;\}\; \boldsymbol{q}_{\text{out}}\end{aligned}$$

with H is a construct template as usual, $\boldsymbol{q}_{\text{out}}$ a linear recursive query, $\boldsymbol{p}_{\text{base}}$ and $\boldsymbol{p}_{\text{rec}}$ positive SPARQL queries, 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. Consequently, we define linear positive rec-SPARQL and linear stratified positive rec-SPARQL by restricting the operators in $\boldsymbol{p}_{\text{base}}$ and $\boldsymbol{p}_{\text{rec}}$. The semantics of linear positive and stratified positive recursive queries is inherited from Definition 3.2.

Notice that we enforce a syntactic separation between base and recursive query. This is done so that we can keep track of changes made in the temporary graph without the need of computing the difference of two graphs, as discussed in Section 4.2. This simple yet powerful syntax resembles the design choices taken in most SQL commercial systems supporting recursion,[6] and is also present in graph databases [18].

To give an example of a linear query, we return to the query from Fig. 3. First, we notice that this query is not linear. Nevertheless, it can be restated as the query from Fig. 5 that uses one level of nesting (meaning that the query $\boldsymbol{q}_{\text{out}}$ is again a linear recursive query). 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 original dataset into a new graph named `http://db.ing.puc.cl/temp1`, and then use this graph as a basis for computing the required reacha-

[6]In SQL one cannot execute a recursive query which is not divided by UNION into a base query (inner query) and the recursive step (outer query) [42].

```
PREFIX prov: <http://www.w3.org/ns/prov#>
WITH RECURSIVE http://db.ing.puc.cl/temp1 AS {
    CONSTRUCT { ?x ?u ?y }
    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 }
    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 WHERE {
  GRAPH <http://db.ing.puc.cl/temp> {
    ?x ?u ?y
  }
}
```

Fig. 5. Example of a linear recursion.

bility condition, that will be dumped into a second temporary graph http://db.ing.puc.cl/temp2.[7]

4.2. Algorithm for linear recursive queries

The main reason why linear queries are widely used in practice is the fact that they can be computed piece by piece, without ever invoking the complete database being constructed. More precisely, if a query $Q = \mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\boldsymbol{q}_1\}\ \boldsymbol{q}_2$ is linear, then for every dataset D, the answer $\mathsf{ans}(Q, D)$ of the query can be computed as the least fixed point of the sequence given by

$$D_0 = D, \qquad D_{-1} = \emptyset,$$

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

In other words, in order to compute the $i + 1$-th iteration of the recursion, 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.

Unfortunately, it is undecidable to check whether a given recursive query satisfies the property outlined above (under usual complexity-theoretic assumptions, see [23]), so this is why we must guarantee it with syntactic restrictions. We also note that most of the recursive extensions proposed for SPARLQ have the aforementioned property: from property paths [26] to nSPARQL [40], SPARQLeR [30], regular queries [43] or Trial [32], as well as our example.

As for the algorithm, 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 [24] for a good survey). In order to describe our algorithm for evaluating a query of the shape (4), we abuse the notation and speak of $\boldsymbol{q}_{\text{base}}$ to denote the query $\mathsf{CONSTRUCT}\ H\ \ \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}$ and $\boldsymbol{q}_{\text{rec}}$ to denote the query $\mathsf{CONSTRUCT}\ H\ \ \mathsf{WHERE}\ \boldsymbol{p}_{\text{rec}}$. Our algorithm for query evaluation is presented in Algorithm 2.

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 algo-

[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 Appendix.

Algorithm 2 Computing the answer for linear recursive c-queries of the form (4)

Input: Query Q of the form (4), dataset D
Output: Evaluation $\mathsf{ans}(Q, D)$ of Q over D

```
1:  Set $G_{\text{temp}} = \mathsf{ans}(\boldsymbol{q}_{\text{base}}, D)$ and $G_{\text{ans}} = G_{\text{temp}}$
2:  Set size $= |G_{\text{ans}}|$
3:  loop
4:      Set $G_{\text{temp}} = \mathsf{ans}(\boldsymbol{q}_{\text{rec}}, D \cup \{(t, G_{\text{temp}})\})$
5:      Set $G_{\text{ans}} = G_{\text{ans}} \cup G_{\text{temp}}$
6:      if size $= |G_{\text{ans}}|$ then
7:          break
8:      else
9:          size $= |G_{\text{ans}}|$
10:     end if
11: end loop
12: return $\mathsf{ans}(\boldsymbol{q}_{\text{out}}, D \cup \{\langle t, G_{\text{ans}}\rangle\})$
```

rithm 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.

Expressive power of linear queries In Section 3.4 we show that all stratified positive recursive SPARQL queries can be defined in datalog, but have no matching result in the other direction. When it comes to linear queries, we can mirror the same result. The definition of linear datalog is quite straightforward: a datalog rule is linear if there is at most one atom in the body of the rule that is recursive with the head. For example, the rule $E(x, y) : -R(x), E(x, y)$ is linear, while $E(x, y) : -R(x), E(x, z), E(z, y)$ is not, since it uses the head predicate recursively twice. A datalog program is linear if all of its rules are linear.

With this in mind, we can again show that every linear positive rec-SPARQL query can be transformed into linear positive datalog, and likewise for linear stratified positive rec-SPARQL and linear stratified datalog (note that the translation outlined in the proof of Proposition 3.4 preserves linearity). Unfortunately, for the other direction we have the same problem, as we are not sure whether our recursive languages can express every linear positive or stratified positive datalog program. Of course, we can also mirror Proposition 3.5 for linear queries. Thus, linear languages such as regular queries [43] or Trial [32] can be also translated into linear queries.

4.3. Supporting arbitrary queries in recursive clauses

Although we show in Section 3 that recursive queries which include some form of negation can be impossible to evaluate, there is no doubt that queries including negation are very useful in practice. In this section we briefly discuss how such queries can be mixed with linear recursion.

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

$$\begin{aligned}&\mathsf{WITH\ RECURSIVE}\ t\ \mathsf{AS}\ \{\\&\quad \mathsf{CONSTRUCT}\ H\\&\quad \mathsf{WHERE}\ \boldsymbol{p}_{\text{base}}\ \mathsf{UNION}\ \boldsymbol{p}_{\text{rec}}\\&\quad \}\ \mathsf{MAXRECURSION}\ k\ \boldsymbol{q}_{\text{out}}\end{aligned} \tag{5}$$

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.

It is easy to see that this extension is useful for handling queries which include negation, or which create values by means of blanks or a BIND clause. Namely, if we fix the number of iterations of a recursive query, we can ensure that these queries terminate their execution, regardless of the existence of a fixed point.

Other ways of supporting negation Limiting the number of iterations can also give us a way of allowing more complex c-queries inside the recursive part of recursive queries. This is not an elegant solution, but can be made to work: since the number of iterations is bounded, we don't longer need queries to ensure that our graph has a fixed point operator.

We also mention that there are other, more elegant solutions, but we do not investigate them further as they drive us out of what can be implemented on top of SPARQL systems, and is out of the scope of this paper. For example, 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. [37]). Moreover, one could redefine rec-SPARQL to consider a *partial fixed point* in Definition 3.2 instead of the least-fixed point. This approach simply assumes a query that does not converge gives an empty result. It is a clean theoretical solution, but it is not a good approach for practice.

Studying these extensions to rec-SPARQL is an important topic for future work, and in particular the stable model semantics approach may require an interesting combination of techniques from both databases and logic programming.

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) [46] 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 four different datasets. The first one is Linked Movie Database (LMDB) [33], an RDF dataset containing information about movies and actors. The second dataset we use is a part of the YAGO ontology [50] and consists of all the facts that hold between instances. For the experiments the version from May 2018 was used. In order to test the performance of our implementation on synthetic data, we turn to the GMark benchmark [9], and generate data with different characteristics using this tool. Finally, we use the Wikidata "truthy" dump from 2018/11/15 containing over 3 billion triples, in order to test whether our implementation scales. All the datasets apart from Wikidata can be found at https://alanezz.github.io/RecSPARQL/.

Experiments The experiments we run are divided into four batches:

- **Common use cases.** In the first round of experiments we turn to YAGO and LMDB datasets, which allow defining recursive queries rather naturally. The main objective of these experiments is to show that our implementation can handle complex recursive patterns in reasonable time over real world datasets.
- **Comparison with SPARQL engines.** In order to compare with the recursive properties supported by SPARQL, we turn to the GMark [9] property path benchmark, and compare our implementation with pache Jena and Openlink Virtuoso, two popular SPARQL systems.
- **Performance over large datasets.** To verify whether our solution scales, we run a sequence of recursive queries over the Wikidata dataset containing over 3 billon triples, and compare our response times with the ones provided by the Wikidata endpoint.
- **Limiting recursion depth.** Finally, we test the solution proposed in Section 4.3, which stops the recursive iteration after a predetermined number of steps. Here we show that this approach is not only useful fro dealing with recursion, but also when evaluating repeated joins.

The experiments involving smaller datasets (LMDB, YAGO, and GMark) were run on a MacBook Pro with an Intel Core i5 2.6 GHz processor and 8 GB of main memory. To handle the size of Wikidata, we used a server Devuan GNU/Linux 3 (beowulf) with an Intel Xeon Silver 4110 CPU @ 2.10 GHz processor and 120 GB of memory.

Next, we elaborate on each batch of experiments, as specified above.

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

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

Table 1
Specifications for the LMDB and Yago datasets

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

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 Table 1. Note that the size is the one used by Jena TDB to store the 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 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.

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

– 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.

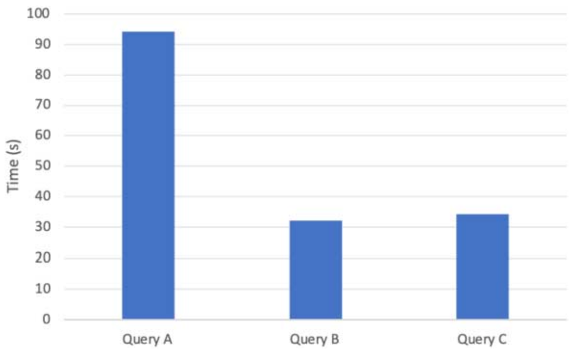
Note that QA, QD and QE are also expressible as property paths. To fully test recursive capabilities of our implementation we use another two queries, QB and QC, that apply various tests along the paths computing the Bacon number. Recall that the structure of queries QB and QC is similar to the query from Example 3.1 and cannot be expressed in SPARQL 1.1 either.

The results of the evaluation can be found in Figs 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 (Figs 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.

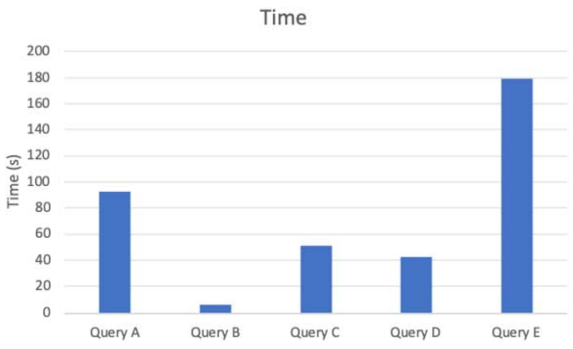
5.2. *Comparison with property paths using the GMark benchmark*

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 [9] 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.



(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.

The GMark benchmark allows generating 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 parametrized by the user. Using the benchmark we generated three different graphs of increasing size, named G_1, G_2 and G_3. The specifications for each graph can be found in Table 2. 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 run times our queries are presented in the Fig. 7 for the graph G_1, in Fig. 8 for G_2, and in Fig. 9 for G_3.

Note first that every property path query is easily expressible using linear recursion. With this observation

Table 2

Specifications for the graphs generated by GMark

Graph	Number of triples	Size
G_1	220564	271 mb
G_2	447851	535 mb
G_3	671712	605 mb

in mind we must also remark that we are comparing the performance of our more general recursive engine with property paths, which are a much less expressive language. For this reason highly efficient systems like Virtuoso should run property paths queries faster: they do not need to worry about being able to compute more recursive queries. Of course, it would also be interesting to compare our engine with specific ad-hoc techniques for computing property paths.

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 patterns of 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 Figs 8 and 9.

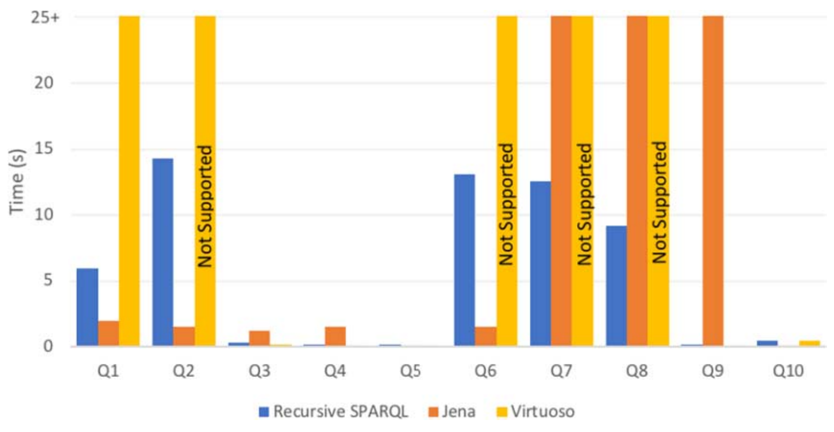


Fig. 7. Times for $G1$.

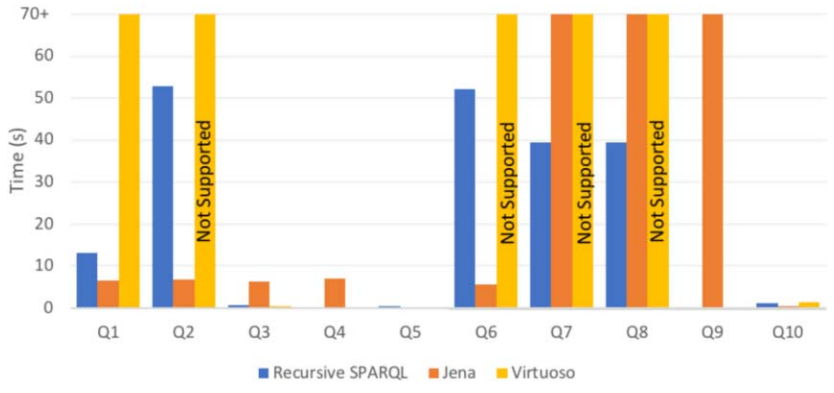


Fig. 8. Times for $G2$.

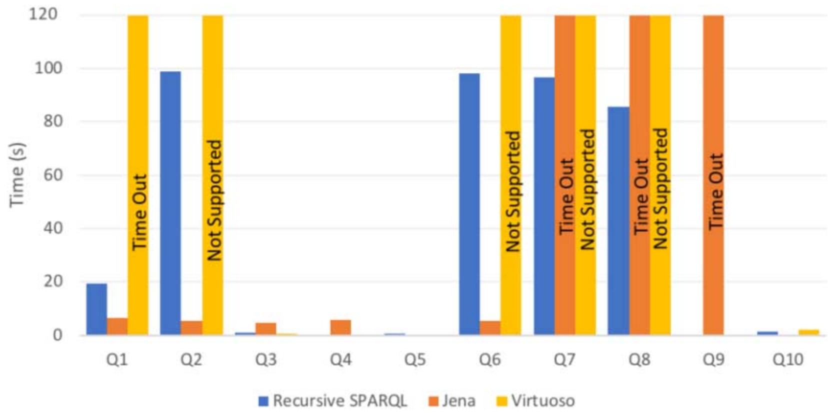


Fig. 9. Times for $G3$.

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 suspect that this could occur because Virtuoso generates many duplicate results, thus the output should be higher with respect to recursive SPARQL and Jena. We count the number of outputs for the queries ran over the first graph. The results can be seen in Table 3.

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 du-

Table 3
Number of outputs for the GMark queries over the graph G_1

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

plicates answers. Also we do not consider paths of 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 a starting point for property path queries, so 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 produce many duplicate answers and thus, the answer time becomes considerably worse. This happen mainly in the first query, which is the simplest one. The same effect happen for the 2 bigger graphs. The number of outputs for the bigger graphs can be found in the Appendix (Tables 5 and 6).

5.3. *Tests over large datasets*

We wanted to know how our recursive operator works when the queries are executed over large datasets. Thus, we decided to try our implementation with queries over the graph of Wikidata, so we load the "truthy" dump from 2018/11/15. This dump contains 3,303,288,386 triples. For this set of experiments we use a server Devuan GNU/Linux 3 (beowulf) with an Intel Xeon Silver 4110 CPU @ 2.10 GHz processor and 120 GB of memory.

We create property paths queries based on (1) the example queries showed at the Wikidata Endpoint and (2) the LMDB queries from Section 5.1. The queries are the following:

– Q1: Sub-properties of property `P276`.
– Q2: All the instances of *horse* or a subclass of *horse*.
– Q3: Parent taxon of the Blue Whale.
– Q4: Metro stations reachable from Palermo Station in *Metro de Buenos Aires*.
– Q5: Actors with finite Bacon number:

Queries Q1, and Q4 are simple star $\star$ queries, while Q3 is a star query where in each iteration only one triple is added to the recursive graph. Q2 is a query of the form `(wdt:p1/wdt:p2*)`, while Q5 combines two properties within a star.

Table 4
Time in seconds taken by the queries over the Wikidata Graph

Query	RecSPARQL	Endpoint
Q1	2.23	0.28
Q2	2.45	1.02
Q3	2.15	0.56
Q4	2.11	0.73
Q5	101.60	Timeout

We rewrote the property paths as `WITH RECURSIVE` queries and we ran them on our server setup. We display the results in Table 4. As a reference, we also put the time that the queries took at the Wikidata endpoint https://query.wikidata.org/. We note that this is not an exact comparison as the endpoint dataset might slightly differ from the one we use, and the server running the endpoint is likely different from ours. The values are expressed in seconds.

As we see, the trend shown with the previous experiments is repeated again with a large dataset: Recursive SPARQL is competitive against existing solutions. Since the dataset of Wikidata is larger than previous datasets and our solution implies to do several joins, we expected easier queries to have better running times in the endpoint than in our implementation. However, the running times our solution displays are still competitive. Finally, we remark the result for Q5, where our implementation could answer the query in a reasonable time, and the endpoint times out.

5.4. *Limiting the number of iterations*

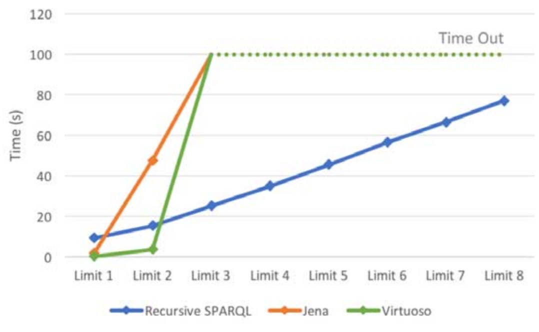
In Section 4.3 we presented a way of limiting the depth of the recursion. We argue that this functionality should find good practical uses, because users are often interested in running recursive queries only for a predefined number of iterations. For instance, very long paths between nodes are seldom 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?

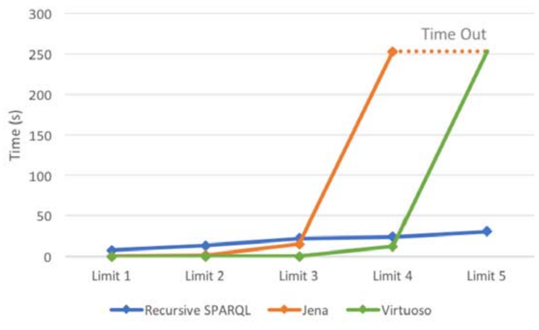
One apparent reason is that it makes queries much easier to write and understand (as a reference we include the rewritings of the query QA, QB and QC from Section 5.1 using only SPARQL operators in the online Appendix). The second reason we would like to argue for is that, when implemented using Algorithm 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 Section 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.

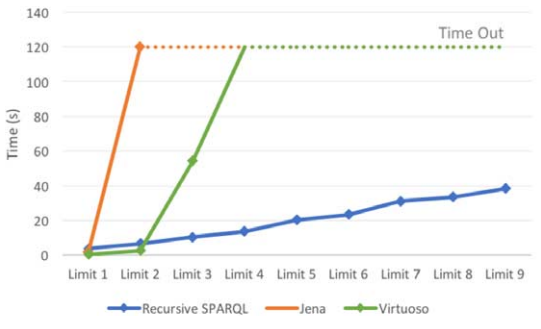
Figure 10 shows the results over LMDB and Fig. 11 shows the results over YAGO. The time out here is again set to two minutes. As we can see, the initial cost is much higher if we are using recursive queries, however as the number of steps increases we can see that they show much better performance and 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 on Jena and Virtuoso, and query QE is composed of two property paths, so there is no straightforward way to transform it in a query with unions.



(a) Query Q_A



(b) Query Q_B

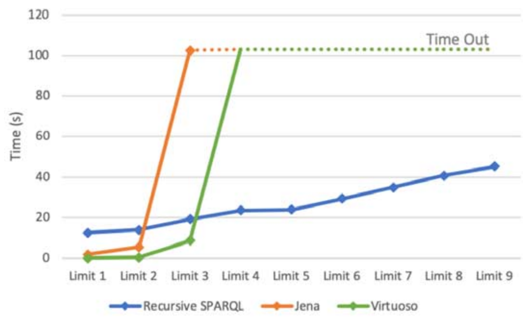


(c) Query Q_C

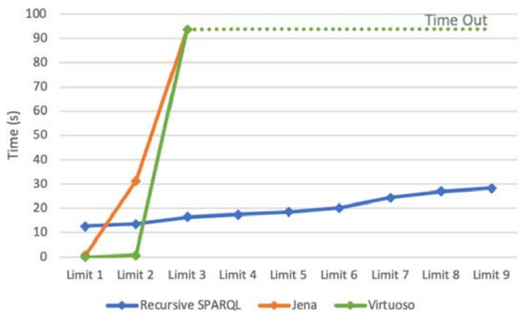
Fig. 10. Limiting the number of iterations for the evaluation of Q_A, Q_B and Q_C over LMDB.

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



(a) Query Q_A



(b) Query Q_C

Fig. 11. Limiting the number of iterations for the evaluation of Q_A and Q_C over Yago.

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.

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 [19], and a similar result should hold for ShEx constraints [15]. 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 [40] or OWL2 QL entailment [14], 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.

Finally, there is also the question of what is the best way of implementing these languages. In this paper we have explored the idea of implementing recursive SPARQL on top of a database system, as is done in SQL. As we discussed, this approach has numerous advantages, and it shows that recursion can be added to SPARQL with little overhead for the companies providing SPARQL processors. However, another option would be to use more powerful engines capable of running full datalog or similarly powerful languages (see e.g. [36] or [12]), which may provide better running times than our on-top-of-system implementation, and should be able to run non-linear queries. This is another source of questions that would require closer work between database and logic programming communities.

Acknowledgements

This work was supported by the Millennium Institute for Foundational Research on Data (IMFD) and by CONICYT-PCHA Doctorado Nacional 2017-21171731.

Appendix

Here we present some of the queries that we ran throughout this work. Note that here we declare the queries using the clauses `FROM` and `FROM NAMED`. This is because some systems need the declaration of the graphs that are used in `GRAPH` clauses.

Queries in Section 4.2

Query from Section 4.2 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 Section 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 Section 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 Section 4.3

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.

Queries over the wikidata endpoint

- **Q1**: Sub-properties of property `P276`:

```
SELECT ?subProperties WHERE {
  ?subProperties wdt:P1647* wd:P276
}
```

- **Q2**: Horse lineages:

```
SELECT ?horse WHERE
{
  ?horse wdt:P31/wdt:P279* wd:Q726
}
```

- **Q3**: Parent taxon of the Blue Whale:

```
SELECT ?taxon WHERE {
  wd:Q42196 wdt:P171* ?taxon
}
```

- **Q4**: Metro stations reachable from Palermo Station in *Metro de Buenos Aires*:

```
SELECT ?metro WHERE
{
  wd:Q3296629 wdt:P197* ?metro
}
```

- **Q5**: Actors with finite Bacon number:

```
SELECT ?actor WHERE
{
  ?actor (^wdt:P161/wdt:P161)*
    wd:Q3454165
}
```

Number of outputs for the bigger graphs in GMark

The number of outputs for the bigger graphs can be found in Tables 5 and 6.

Table 5

Number of outputs for the GMark queries over the graph G_2

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

Table 6

Number of outputs for the GMark queries over the graph G_3

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

References

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

[2] 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. doi:10.1016/j.websem.2009.02.002.

[3] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter and D. Vrgoč, Foundations of modern query languages for graph databases, *ACM Computing Surveys (CSUR)* **50**(5) (2017), 1–40. doi:10.1145/3104031.

[4] R. Angles and C. Gutierrez, The multiset semantics of SPARQL patterns, in: *International Semantic Web Conference*, Springer, 2016, pp. 20–36. doi:10.1007/978-3-319-46523-4_2.

[5] 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.

[6] M. Arenas, G. Gottlob and A. Pieris, Expressive languages for querying the semantic web, in: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2014, pp. 14–26. doi:10.1145/2594538.2594555.

[7] M. Arenas, C. Gutierrez and J. Pérez, On the semantics of SPARQL, in: *Semantic Web Information Management: A Model-Based Perspective*, R. de Virgilio, F. Giunchiglia and L. Tanca, eds, Springer, Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 281–307. ISBN 978-3-642-04329-1. doi:10.1007/978-3-642-04329-1_13.

[8] M. Atzori, Computing recursive SPARQL queries, in: *ICSC*, 2014, pp. 258–259. doi:10.1109/ICSC.2014.54.

[9] 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. doi:10.1109/TKDE.2016.2633993.

[10] 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. doi:10.1007/978-3-642-45221-5_5.

[11] P. Barceló, J. Pérez and J.L. Reutter, Relative expressiveness of nested regular expressions, in: *AMW*, 2012, pp. 180–195.

[12] L. Bellomarini, E. Sallinger and G. Gottlob, The vadalog system: Datalog-based reasoning for knowledge graphs, *Proceedings of the VLDB Endowment* **11**(9) (2018), 975–987. doi:10.14778/3213880.3213888.

[13] 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.

[14] 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. doi:10.1007/978-3-319-11964-9_37.

[15] 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. doi:10.1007/978-3-319-68288-4_7.

[16] A. Bonifati, G. Fletcher, H. Voigt and N. Yakovets, Querying graphs, *Synthesis Lectures on Data Management* **10**(3) (2018), 1–184. doi:10.2200/S00873ED1V01Y201808DTM051.

[17] 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.

[18] 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. doi:10.1145/298514.298591.

[19] J. Corman, F. Florenzano, J.L. Reutter and O. Savkovic, Validating shacl constraints over a sparql endpoint, in: *The Semantic Web – ISWC 2019 – 18th International Semantic Web Conference, Proceedings, Part I*, Auckland, New Zealand, October 26–30, 2019, Lecture Notes in Computer Science, Vol. 11778, Springer, 2019, pp. 145–163. doi:10.1007/978-3-030-30793-6_9.

[20] V. Fionda and G. Pirrò, Explaining graph navigational queries, in: *European Semantic Web Conference*, Springer, 2017, pp. 19–34. doi:10.1007/978-3-319-58068-5_2.

[21] 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.

[22] V. Fionda, G. Pirrò and C. Gutierrez, NautiLod: A formal language for the web of data graph, *ACM Transactions on the Web (TWEB)* **9**(1) (2015), 5. doi:10.1145/2697393.

[23] 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. doi:10.1145/174130.174142.

[24] 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. doi:10.1561/1900000017.

[25] A. Gubichev, S.J. Bedathur and S. Seufert, Sparqling kleene: Fast property paths in RDF-3X, in: *GRADES*, 2013. doi:10.1145/2484425.2484443.

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

[27] D. Hernández, A core SPARQL fragment, 2020, https://users.dcc.uchile.cl/~dhernand/reports/Ei6iutheb1.html.

[28] P. Hitzler, M. Krotzsch and S. Rudolph, *Foundations of Semantic Web Technologies*, CRC Press, 2011.

[29] 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. doi:10.4230/LIPIcs.ICDT.2016.5.

[30] 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. doi:10.1007/978-3-540-72667-8_12.

[31] E.V. Kostylev, J.L. Reutter and M. Ugarte, CONSTRUCT queries in SPARQL, in: *ICDT*, 2015, pp. 212–229. doi:10.4230/LIPIcs.ICDT.2015.212.

[32] 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. doi:10.1145/3154385.

[33] Linked movie database.

[34] 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. doi:10.1145/2457317.2457375.

[35] 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.

[36] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu and J. Banerjee, RDFox: A highly-scalable RDF store, in: *International Semantic Web Conference*, Springer, 2015, pp. 3–20. doi:10.1007/978-3-319-25010-6_1.

[37] 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.

[38] Open Link Virtuoso, 2015.

[39] J. Pérez, M. Arenas and C. Gutierrez, Semantics and complexity of SPARQL, *ACM Transactions on Database Systems* **34**(3) (2009). doi:10.1145/1567274.1567278.

[40] J. Pérez, M. Arenas and C. Gutierrez, nSPARQL: A navigational language for RDF, *J. Web Sem.* **8**(4) (2010), 255–270. doi:10.1016/j.websem.2010.01.002.

[41] A. Polleres and J.P. Wallner, On the relation between SPARQL1.1 and answer set programming, *Journal of Applied Non-Classical Logics* **23**(1–2) (2013), 159–212. doi:10.1080/11663081.2013.798992.

[42] PostgreSQL documentation.

[43] J.L. Reutter, M. Romero and M.Y. Vardi, Regular queries on graph databases, *Theory of Computing Systems* **61**(1) (2017), 31–83. doi:10.1007/s00224-016-9676-2.

[44] J.L. Reutter, A. Soto and D. Vrgoč, Recursion in SPARQL, in: *The Semantic Web – ISWC 2015 – 14th International Semantic Web Conference, Proceedings, Part I*, Bethlehem, PA, USA, October 11–15, 2015, 2015, pp. 19–35. doi:10.1007/978-3-319-25007-6_2.

[45] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, 1955. doi:10.2307/2963937.

[46] The Apache Jena Manual, 2015.

[47] M.Y. Vardi, On the complexity of bounded-variable queries, in: *PODS*, Vol. 95, 1995, pp. 266–276. doi:10.1145/212433.212474.

[48] W3C, PROV Model Primer, 2013.

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

[50] YAGO: A High-Quality Knowledge Base.

[51] N. Yakovets, P. Godfrey and J. Gryz, Evaluation of SPARQL property paths via recursive SQL, in: *AMW*, 2013.

[52] N. Yakovets, P. Godfrey and J. Gryz, WAVEGUIDE: evaluating SPARQL property path queries, in: *EDBT 2015*, 2015, pp. 525–528. doi:10.5441/002/edbt.2015.49.