Navigational and rule-based languages for graph databases[⋆]

Juan Reutter and Domagoj Vrgoč

Pontificia Universidad Católica de Chile and Center for Semantic Web research, CL

Abstract. One of the key differences between graph and relational databases is that on graphs we are much more interested in navigational queries. As a consequence, graph database systems are specifically engineered to answer these queries efficiently, and there is a wide body of work on query languages that can express complex navigational patterns. The most commonly used way to add navigation into graph queries is to start with a basic pattern matching language and augment it with navigational primitives based on regular expressions. For example, the friend-of-a-friend relationship in a social network is expressed via the primitive (friend)+, which looks for paths of nodes connected via the friend relation. This expression can be then added to graph patterns, allowing us to retrieve, for example, all nodes A,B and C that have a common friend-of-a-friend.
But, in order to alleviate some of the drawbacks of isolating navigation in a set of primitives, we have recently witnessed an effort to study languages which integrate navigation and pattern matching in an intrinsic way. A natural candidate to use is Datalog, a well known declarative query language that extends first order logic with recursion, and where pattern matching and recursion can be arbitrarily nested to provide much more expressive navigational queries.
In this paper we review the most common navigational primitives for graphs, and explain how these primitives can be embedded into Datalog. We then show current efforts to restrict Datalog in order to obtain a query language that is both expressive enough to express all these primitives, but at the same time feasible to use in practice. We illustrate how this works both over the base graph model and over the more general RDF format underlying the semantic web.

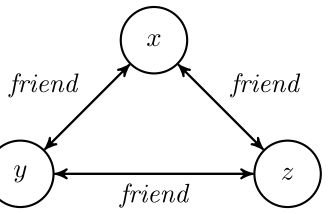
1 Introduction

Graph structured data is quickly becoming one of the more popular data paradigms for storing data in computer applications. Social networks, bioinformatics, astronomic databases, digital libraries, Semantic Web, and linked government data, are only a few examples of applications in which structuring data as graphs is essential. There is a growing body of literature on the subject and there are now several vendors of graph database systems [54, 47, 3]. See also [4,

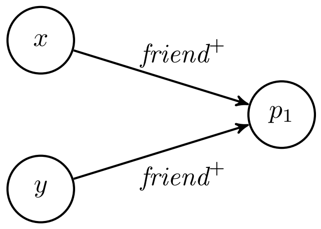
⋆ This work was supported by Iniciativa Científica Milenio Grant 120004

12] for surveys of the area. The simplest model of a graph database is that of edge-labelled directed graphs, where the nodes of the graph represent objects and the labelled edges represent relationships between these objects.

A standard way of querying graph data is to use a pattern matching language to look for small subgraphs of interest. For example, in a social network one can match the following pattern to look for a clique of three individual that are all friends with each other:



However, in several graph database applications one also needs to look for more complex conditions between nodes that are not necessarily adjacent to each other, which are known as navigational queries. The most commonly used way to add navigation into graph queries is to start with a basic pattern matching language and augment it with navigational primitives based on regular expressions. For example, the friend-of-a-friend relationship in a social network is expressed via the primitive friend^+, which looks for paths of nodes connected via an indirect chain of friends. We can then add this primitive into the following pattern, to look for pairs of persons with an indirect friend p_1 in common:



Navigational queries showcase one of the key differences between graph and relational databases, because these queries use a form of recursion that traditional relational languages and engines are not designed to deal with. As a consequence, graph database systems are specifically engineered to answer navigational queries efficiently, and there is a wide body of work on query languages that can express complex navigational patterns [12].

The first and most common navigational primitive that was proposed is that of *Regular Path Queries*, or RPQs [27], which corresponds precisely to regular expressions. But nowadays there exists a wide range of extensions of RPQs that add numerous other features such as the ability to traverse edges backwards, existential test operators such as the one used in XPath [32], negation over paths [29], etc. Graph systems also implement these types of navigational primitives; for example the graph system Neo4j [54] implements a subset of RPQs, and

SPARQL [35], the standard query language for RDF graphs [39], features *Property Paths*: another extension of RPQs. The combination of graph patterns and RPQs is usually modelled as a language where RPQs can be added to standard *conjunctive queries*, resulting in *Conjunctive Regular Path Queries*, or CRPQs [19]. This basic navigational pattern matching language are nowadays well understood, and there are also several extensions that have been proposed or are now implemented (see [12]).

But isolating navigation in a set of primitives has drawbacks for both systems and users. First, the algorithmic challenges needed to support efficient navigation are different from those needed to support efficient pattern matching. Thus, one usually ends up implementing two separate engines to compute the answers of a navigational query: one for pattern matching and one for dealing with the navigational primitives. This makes other database problems such as query planning and query optimization substantially more difficult, since now one has to implement techniques that work across both engines. This issue is also closely related to the fact that the navigational queries are generally not an *algebraically closed* language, in the sense that one cannot re-apply the same operators used for navigation to a graph pattern (for example, there is no notion of applying the transitive Kleene star of regular expressions to the navigational pattern of indirect friends shown above). Thus, posing queries in these language can also be uncomfortable for users familiar with algebraically closed languages such as the relational algebra, or SQL in general. But, additionally, by focusing on primitives designed to deal with paths we leave out the possibility of expressing other complex navigational relationships that cannot be reduced to a set of path operations.

In order to alleviate this situation, we have recently witnessed an effort to study languages which integrate navigation and pattern matching in an intrinsic way. A natural candidate to use is Datalog, a well known declarative query language that extends first order logic with recursion, and where pattern matching and recursion can be arbitrarily nested to provide much more expressive navigational queries. Datalog is one of the most popular languages in databases and has been used in numerous applications (one example is information extraction on the Web [31]).

The first attempt to define a Datalog-like language for graph databases is GraphLog [24], a language specifically designed to be closed, in the sense that GraphLog queries not only mix navigation and pattern matching directly, but also one can design queries where the Kleene star is directly applied to patterns in order to form a different graph database. However, dealing with Datalog has its own challenges, since it makes standard problems such as query evaluation and query containment substantially more difficult than they are for navigational queries based on pattern matching and primitives based on regular expressions. Thus, in the last years there have been numerous proposals to restrict GraphLog and similar languages so that they enjoy these other good algorithmic properties [55, 17, 16, 18, 52, 15]. A good example of such a restriction are Regular Queries [52], a subset of Datalog that has almost the same algorithmic properties as

CRPQs but whose expressive power is a good approximation of what GraphLog can do.

In this paper we review the most common navigational primitives for graphs, and explain how these primitives can be embedded into Datalog. We then show current efforts to restrict Datalog in order to obtain a query language that is both expressive enough to subsume all these primitives, but at the same time feasible to use in practice. To show how these concepts can be used in a specific graph application, we then move to RDF databases, the graph format underlying the Semantic Web. We review Property Paths and Nested Regular Expressions, the choices of navigational primitives for RDF, and then show the specific problems we encounter when trying to design an algebraically closed language for RDF.

About the languages included in this survey We would like to stress that this is not intended to be a complete survey of graph querying features and languages. The objective of this article is to provide a good overview of query features that separate graph databases from the traditional relational format, and to do this we will focus on navigational aspects of graph querying. In particular, we will place a strong emphasis on languages based on regular expressions, and also show how more declarative formalisms (such as e.g. Datalog) can be used to capture interesting properties over graphs. Note that we do not consider several other important features of graph query languages such as e.g. attribute values and how these mix with navigational queries [45].

Organization We introduce the formal model of graph databases and review relational queries in Section 2. Basic graph query languages based on regular expression are described in Section 3. In Section 4 we talk about how Datalog can be used to capture navigation over graphs, and in Section 5 we illustrate what problems we face when applying graph query languages over the RDF format underlying the Semantic Web. We conclude in Section 6.

2 Notation

Graph databases. Let Σ be a finite alphabet. A *graph database* G over Σ is a pair (V, E), where V is a finite set of nodes and $E \subseteq V \times \Sigma \times V$ is a set of edges. That is, we view each edge as a triple $(v, a, v') \in V \times \Sigma \times V$, whose interpretation is an a-labelled edge from v to v' in G. When Σ is clear from the context, we shall simply speak of a graph database. Figure 1 shows an example of a graph database that stores information about a social network: here the nodes represent individuals that can be connected by relation *knows*, indicating that a person knows another person, or by *helps*, indicating that a person has helped another person in the past. Unless we specify otherwise, the size $|G|$ of G is simply the number of nodes in V plus the number of tuples in E.

We define the finite alphabet $\Sigma^{\pm} = \Sigma \cup \{a^- \mid a \in \Sigma\}$, that is, $\Sigma^{\pm}$ is the extension of Σ with the *inverse* of each symbol. The *completion* $G^{\pm}$ of a graph database G over Σ, is a graph database over $\Sigma^{\pm}$ that is obtained from G by adding the edge (v', a^-, v) for each edge (v, a, v') in G.

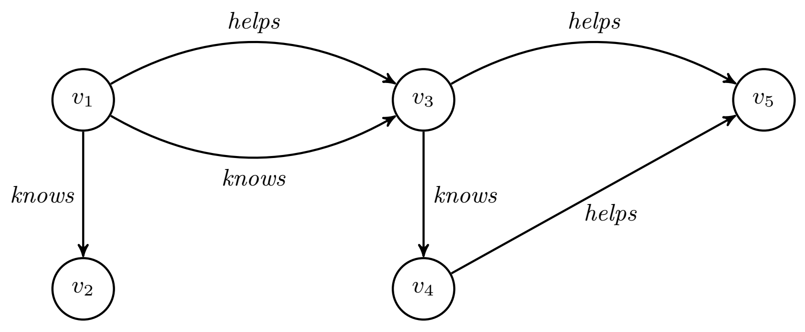


Fig. 1. A graph database over alphabet $\{knows, helps\}$, in which nodes are v_i, $1 \leq i \leq 5$.

A *path* ρ from v_0 to v_m in a graph $G = (V, E)$ is a sequence (v_0, a_0, v_1), $(v_1, a_1, v_2), \cdots, (v_{m-1}, a_{m-1}, v_m)$, for some $m \geq 0$, where each (v_i, a_i, v_{i+1}), for $i < m$, is an edge in E. In particular, all the v_i's are nodes in V and all the a_j's are letters in Σ. The *label* of ρ, denoted by $\lambda(\rho)$, is the word $a_0 \cdots a_{m-1} \in \Sigma^*$. We also define the empty path as (v, ε, v) for each $v \in N$; the label of such path is the empty word ε.

Relational queries: CQs and UCQs. A relational schema is a set $\sigma = \{R_1, \ldots, R_n\}$ of relation symbols, with each R_i having a fixed arity. Let $\mathbf{D}$ be a countably infinite domain. An instance I of σ assigns to each relation R in σ of arity n a finite relation $R^I \subseteq \mathbf{D}^n$. We denote by $\mathrm{dom}(I)$ the set of all elements from $\mathbf{D}$ that appear in any of the relations in I.

A *conjunctive query* (CQ) over a relational schema σ is a formula $Q(\bar{x}) = \exists \bar{y} \varphi(\bar{x}, \bar{y})$, where $\bar{x}$ and $\bar{y}$ are tuples of variables and $\varphi(\bar{x}, \bar{y})$ is a conjunction of relational atoms from σ that use variables from $\bar{x}$ and $\bar{y}$. We say that $\bar{x}$ are the free variables of the query Q. Conjunctive queries with no free variables are called boolean CQs; if Q is a boolean CQ, we identify the answer **false** with the empty relation, and **true** with the relation containing the 0-ary tuple.

We want to use CQs for querying graph databases over a finite alphabet Σ. In order to do this, given an alphabet Σ, we define the schema $\sigma(\Sigma)$ that consists of one binary predicate symbol E_a, for each symbol $a \in \Sigma$. For readability purposes, we identify E_a with a, for each symbol $a \in \Sigma$. Each graph database $G = (V, E)$ over Σ can be represented as a relational instance $\mathbf{D}(G)$ over the schema $\sigma(\Sigma)$: The database $\mathbf{D}(G)$ consists of all facts of the form $E_a(v, v')$ such that (v, a, v') is an edge in G (for this we assume that $\mathbf{D}$ includes all the nodes in V).

A conjunctive query over Σ is simply a conjunctive query over $\sigma(\Sigma^{\pm})$. The answer $Q(G)$ of a CQ Q over G is $Q(\mathbf{D}(G^{\pm}))$. A union of CQs (UCQ) Q over Σ is a disjunction $\theta_1(\bar{x}) \vee \cdots \vee \theta_k(\bar{x})$ of CQs over Σ with the same free variables. The answer $Q(G)$ is $\bigcup_{1 \leq j \leq k} \theta_j(G)$, for each graph database G.

Example 1. Consider a social network over alphabet $\{knows, helps\}$ such as the one from Figure 1. The CQ

$$Q(x) \; = \exists y \exists z \big(knows(x,y) \wedge helps(y,z) \big)$$

retrieves all people that know a helper (a person that helps someone else). In the graph G from Figure 1, we have that $v_1 \in Q(G)$, since v_1 knows v_3, and v_3 helps v_5. Similarly, $v_3 \in Q(G)$.

On the other hand, if we want to retrieves all people that either help someone, or that know someone who is a helper, we can use the following UCQ

$$Q'(x) \; = \exists p \; helps(x,p) \vee \exists y \exists z \; (knows(x,y) \wedge helps(y,z)).$$

3 Navigational languages for graph databases

In this section we review the most widely used graph navigational primitives, and introduce query languages that are obtained when we combine these primitives in order to build more complex graph patterns. For each of these languages we study their expressive power, as well as the complexity of some computational tasks associated with them.

3.1 Path queries

The most simple navigational querying mechanism for graph databases is provided by means of regular expressions, which are commonly known as *regular path queries*, or *RPQs* [1, 27, 20]. Formally, an RPQ Q over Σ is a regular language $L \subseteq \Sigma^*$, and it is specified using a regular expression R. The idea behind regular path queries is to select all pairs of nodes whose labels belong to the language of the defining regular expression. That is, given a graph database $G = (V, E)$ and an RPQ R, both over Σ, the evaluation $[\![R]\!]_G$ of R over G is the set of all pairs $(v, v') \in V$ such that there is path ρ between v and v', and the label $\lambda(\rho)$ of this path is a word in the language of R.

Example 2. Consider again the social network with relations *knows* and *helps* from Figure 1. We can use RPQs to extract basic navigational information about this graph. For example, the query $knows^+$ retrieves all pairs of persons that are connected by a path $v_1, \dots, v_n$ of individuals, where each v_i knows v_{i+1}. Furthermore, $knows \cdot helps$ can be used to retrieve all individuals that know a helper, and $knows + helps$ retrieves all nodes connected by a paths of individuals linked by either a *knows* edge or a *helps* edge.

The idea of using regular expressions for querying graph databases has been well established in the literature [19, 48], and several extensions have been proposed for RPQs. The most popular is 2RPQs [20], which adds to RPQs the possibility of traversing the edges in a backwards direction. Furthermore, the language of 2RPQs has been subsequently extended to *Nested regular path queries*

$$\begin{aligned}
[\![\varepsilon]\!]_G &= \{(u,u) \mid u \text{ is a node id in } G\} \\
[\![a]\!]_G &= \{(u,v) \mid (u,a,v) \in G\} \\
[\![a^-]\!]_G &= \{(u,v) \mid (v,a,u) \in G\} \\
[\![R_1 \cdot R_2]\!]_G &= [\![R_1]\!]_G \circ [\![R_2]\!]_G \\
[\![R_1 + R_2]\!]_G &= [\![R_1]\!]_G \cup [\![R_2]\!]_G \\
[\![R^*]\!]_G &= [\![\varepsilon]\!]_G \cup [\![R]\!]_G \cup [\![R \cdot R]\!]_G \cup [\![R \cdot R \cdot R]\!]_G \cup \cdots \\
[\![\,[R]\,]\!]_G &= \{(u,u) \mid \text{there exists } v \text{ s.t. } (u,v) \in [\![R]\!]_G\}.
\end{aligned}$$

Table 1. Semantics of NRPQs. Here a is a symbol in Σ, and R, R_1 and R_2 are arbitrary NRPQs. The symbol $\circ$ denotes the usual composition of binary relations.

(NRPQs), with the inclusion of an *existential test* operator, similar to the one in XPath [32]. NRPQs were proposed in [50] for querying Semantic Web data, and as a querying formalism they offer a substantial increase in expressive power in comparison with 2RPQs, while maintaining the same query evaluation properties [50]. For this reason the language of nested regular path queries has received a fair deal of attention in the last few years [10, 14, 17].

Just as RPQs and 2RPQs, NRPQs specify pairs of node ids in a graph database, subject to the existence of a path satisfying a certain regular condition among them. The syntax of NRPQs over an alphabet Σ is given by the following grammar:

$$R \ := \ \varepsilon \ \mid \ a \ (a \in \Sigma) \ \mid \ a^- \ (a \in \Sigma) \ \mid \ R \cdot R \ \mid \ R^* \ \mid \ R + R \ \mid \ [R]$$

As usual we use R^+ as shorthand for $R \cdot R^*$. Moreover, when it is clear from the context, we omit the concatenation operator. Thus, if r_1 and r_2 are NRPQs, we sometimes just write $r_1 r_2$ instead of $r_1 \cdot r_2$. The size $|R|$ of an NRPQ is the number of characters used to represent R, and the nesting depth of R is the maximum number of nested [] operators in R (that is, the nesting depth of an expression that does not use any [] operator is 0, the nesting depth of $[R]$ is 1 plus the nesting depth of R, and all other operations preserve the maximum nesting depth of its subexpressions).

Although the semantics of an NRPQ R can be defined in terms of paths, it is best to define the binary relation $[\![R]\!]_G$, corresponding to the evaluation of R over a graph database G, in an inductive fashion. We present the definition in Table 1.

Note that NRPQs subsume RPQs and 2RPQs. In fact, 2RPQs are just NRPQs that do not use the test operator $[R]$, and RPQs are NRPQs that use neither $[R]$ nor the inverse operator $^-$.

Example 3. Let G be the graph database in Figure 1 that used labels *knows* and *helps*. Recall that in Example 2 we used the RPQ (which is also an NRPQ) $R_1 = \mathit{knows}^+$ to retrieve all pairs of nodes connected by a path in which all the edges are labelled *knows*. In particular, the pair (v_1, v_2) belongs to $[\![R_1]\!]_G$, and so do (v_1, v_3) and (v_1, v_4). If we now consider instead the NRPQ

$$R_2 \ = \ (\mathit{knows} + \mathit{knows}^-)^+,$$

we are now searching for a path of *knows*-labelled edges that may be traversed in either direction. Thus, the pair (v_2, v_4) now belongs to $[\![R_2]\!]_G$, as we can travel from v_2 to v_1 by using $knows^-$, and then to v_4 via two *knows*-labelled edges. This query is not an RPQ, but it is a 2RPQ. The NRPQ

$$R_3 \ = \ (knows[helps])^+$$

asks for all nodes x and y that are connected by a path of *knows*'s, but such that from each node in this path, except from x, there is also an outgoing edge labelled *helps*. The pair (v_1, v_4) belongs to $[\![R_3]\!]_G$, but $(v_1, v_2) \notin [\![R_3]\!]_G$, as v_2 has no outgoing *helps*-labelled edge. This query is neither an RPQ nor a 2RPQ.

Query evaluation. As usual, the query evaluation problem asks, given a query R, a graph database G and a pair (u, v) of nodes from G, whether (u, v) belongs to the evaluation $[\![R]\!]_G$. The problem of evaluating RPQs was first considered in [27], where the resemblance between graph databases and automata was exploited to produce a simple algorithm that is linear in both the size of the graph and the size of the query. The idea is the following. Given a graph database $G = (V, E)$ over Σ, an RPQ R and a pair (u, v) of nodes from V, in order to decide wether (u, v) belongs to $[\![R]\!]_G$ one constructs from G the automaton $A_G(u, v) = (V, \Sigma, u, \{v\}, E)$ and the automaton A_R that accepts the language given by R. Note that $A_G(u, v)$ is obtained by viewing G as an NFA in which the initial state is u, the only final state is v, and the transition function is given by the edge relation E. Then it is not difficult to show that (u, v) belong to $[\![R]\!]_G$ if and only if the language of the product automaton $A_G(u, v) \times A_R$ is nonempty. Since the size of the automata is linear in the size of R, the size of the resulting product automata is $O(|G| \cdot |R|)$, and the reachability test is linear, giving us the desired time bounds. But we can also obtain an NLogspace upper bound by performing the usual on-the-fly reachability test on the said product automaton. Hardness follows by reduction from the connectivity problem.

This result was lifted to 2RPQs in [19], the evaluation algorithm is based on the idea that evaluating a 2RPQ R over Σ on a graph G is the same as evaluating R over the completion $G^{\pm}$ of G, but now treating R as an RPQ over the extended alphabet $\Sigma^{\pm}$. Thus, all one needs to do is to obtain $G^{\pm}$ and then compute $[\![R]\!]_{G^{\pm}}$ just as explained above.

Proposition 1 ([19]). *The query evaluation problem for a 2RPQ R and a graph G is* NLogspace*-complete, and can be solved in $O(|G| \cdot |R|)$.*

It turns out that one can also obtain the same linear bounds even for NRPQs. The idea is to start with the innermost sub-expressions of the form $[R']$ in R, where the nesting depth of R' is 0. We evaluate R' using the algorithm for 2RPQs, and then augment graph G with a self-loop labeled $[R']$ in node u' whenever there is a node v' such that $(u', v') \in [\![R']\!]_G$. We can now repeat the process, treating R as an as an NRPQ with 1 less level of nesting over the extended alphabet that considers $[R]$ as an additional label. However, this time we need to assume that the nesting depth of the expression is fixed in order to obtain an NLogspace upper bound.

Proposition 2 ([50]). *The query evaluation problem for an NRPQ R and a graph G can be solved in $O(|G| \cdot |R|)$. The problem is* NLogspace*-complete if the nesting depth of R is assumed to be fixed.*

Query answering has also been studied in the context of description logics, and interestingly, the complexity of the evaluation problem for nested regular path queries is usually higher than that for 2RPQs, even when considering knowledge bases given by simple DL-lite ontologies [14].

Query containment. Another important problem in the study of query languages is that of containment. Formally, the containment problem asks, given queries R_1 and R_2 over Σ, whether $[\![R_1]\!]_G \subseteq [\![R_2]\!]_G$ on all possible graph databases over Σ. Checking query containment is a fundamental problem in database theory, and is relevant to several database tasks such as data integration [43], query optimisation [2], view definition and maintenance [34], and query answering using views [21].

As an example, query $(aa)^+$ is contained in (a^+), because all nodes connected by a path of even number of as are also connected by a path of a's. The problem becomes more interesting when considering 2RPQs, for example, the query aa is contained in $a(a^- a)^+ a$, since in particular every path of two as will be part of the evaluation of the second query.

The containment problem has also received substantial attention in the graph database community. Calvanese et al. showed that the problem can be solved in Pspace for RPQs and 2RPQs [19]. The proof uses the fact that an RPQ R_1 is contained in an RPQ R_2 over all graph databases if and only if they are contained only over *paths*, which allows us to work instead over strings: one can show that R_1 is contained in R_2 over graphs if an only if the language given by the regular expression R_1 is contained in the language of R_2. This gives us an immediate Pspace tight bound for the complexity of containment since testing containment of two regular expressions is Pspace-complete [49]. Likewise, for 2RPQs we can limit the search space for a counterexample to *semipaths*, or paths in which edges may be reversed, enabling us to reason about containment of 2RPQs by a clever rewriting of queries into 2-way automata [19], showing that the containment problem for 2RPQs is still in Pspace.

For NRPQs the picture is a bit more complicated, since we cannot concentrate anymore on path-like structures. For example, consider the NRPQ $R_1 = a[b]a[b] + c^*$. Since the left disjunct of R_1 is not satisfiable over paths, we have that R_1 is contained in $R_2 = c^*$ over paths. However, R_1 is not contained in R_2 if we consider all possible graphs over $\{a, b, c\}$. Nevertheless, one can still solve the containment of NRPQs in Pspace. The idea of the algorithm is to transform NRPQs into alternating two-way automata over a special types of trees, which can be subsequently encoded into strings.

Proposition 3 ([19, 51]). *The containment problem for two NRPQs is* Pspace*-complete. It is* Pspace*-hard even when the input are RPQs.*

3.2 Adding conjunction, union and projection

It has been argued (see, e.g., [27, 24, 1, 19]) that analogs of conjunctive queries whose atoms are navigational primitives such as RPQs, 2RPQs or NRPQs are much more useful in practice than the simple binary primitives. This motivated in [30] the study of *conjunctive regular path queries*, or *CRPQs*, and the further definition of *conjunctive two-way regular path queries* (C2RPQs, [19]) and *conjunctive nested regular path queries* (CNRPQs, [11, 14]).

In such queries, multiple path queries can be combined, and some variables can be existentially quantified. Formally, a CNRPQ Q over a finite alphabet Σ is an expression of the form:

$$Q(\bar{z}) \;=\; \bigwedge_{1\leq i\leq m} (x_i, L_i, y_i), \tag{1}$$

such that $m > 0$, each L_i is an NRPQ, and $\bar{z}$ is a tuple of variables among $\bar{x}$ and $\bar{y}$. The atom $Q(\bar{z})$ is the *head* of the query, the expression on the right of the equality is its *body*. A query with the head $Q()$ (i.e., no variables in the output) is called a *Boolean* query. CRPQs and C2RPQs are defined analogously, requiring instead the L_is to be RPQs or 2RPQs, respectively.

Intuitively, a query of the form (1) selects tuples $\bar{z}$ for which there exist values of the remaining node variables from $\bar{x}$ and $\bar{y}$ such that each RPQ (respectively, 2RPQ or NRPQ) in the body is satisfied. Formally, given Q of the form (1) and a graph $G = (V, E)$, a valuation is a map $\tau : \bigcup_{1\leq i\leq m}\{x_i, y_i\} \to V$. We write $(G, \tau) \models Q$ if $(\tau(x_i), \tau(y_i))$ is in $[\![L_i]\!]_G$. Then the evaluation $Q(G)$ of Q over G is the set of all tuples $\tau(\bar{z})$ such that $(G, \tau) \models Q$. If Q is Boolean, we let $Q(G)$ be `true` if $(G, \tau) \models Q$ for some τ (that is, as usual, the singleton set with the empty tuple models `true`, and the empty set models `false`).

Example 4. Recall the social network from Figure 1 that connects people via the *helps* and *knows* relationships. The following query looks for two individuals u and v that are connected both by a path of *helps* relations and by a path of *friends* relations:

$$Q(x, y) \;=\; (x, \mathit{helps}^+, y) \wedge (x, \mathit{knows}^+, y).$$

Note that the query in the example above has the same structure as the pattern $(x, \mathit{helps}, y) \wedge (x, \mathit{knows}, y)$ we used to compute people connected by both *knows* and *helps* in Example 1. And indeed, there is a tight connection between relational conjunctive queries and the notion of CRPQs, C2RPQs and CNRPQs. Namely, CQs can be seen as queries over a relational representation of a graph, where we can use the edge labels as basic navigational primitives. In CRPQs this is generalised, and we can now use any regular language in place of simple edge labels. Likewise, in C2RPQs we can use regular expressions over $\Sigma^{\pm}$, and in CNRPQs we can use any NRPQ (see [9] for a more detailed study of this type of graph patterns).

Query evaluation. All three classes of graph queries we consider here contain the class of CQs, so they inherit the NP-hardness bound for query evaluation from CQs [22]. And using the fact that the evaluation of each primitive is in polynomial time, it is not difficult to show that this bound is tight: to check wether a tuple $\bar{a}$ belongs to the answer of a CNRPQ $Q(\bar{z})$ of the form (1) over a graph G, one just need to guess a valuation τ that maps $\bar{z}$ to $\bar{a}$, and then verify, for each conjunct (x_i, L_i, y_i) of Q, that $(\tau(x_i), \tau(y_i))$ is in $[\![L_i]\!]_G$ (in polynomial time since L_i is an NRPQ). In databases it is also customary to study the evaluation problem when the query is considered to be fixed, which is known as the *data complexity* of the evaluation problem. For CQs the data complexity is in AC^0, which is contained in NLogspace, and we can plug-in the NLogspace algorithm to compute the answers of each NRPQ to obtain an NLogspace upper bound for the evaluation of any fixed CNRPQ. Hardness follows directly from Proposition 1, since RPQs are a special case of CNRPQs.

Observation 1 *The query evaluation problem for CNRPQs is* NP*-complete. It is* NLogspace*-complete in data complexity (when the query is fixed).*

Query containment. The containment problem for CRPQs was first studied by Calvanese et al. [19], and from there onwards we have seen a great deal of work devoted to the containment problem for various restrictions and extensions of these languages. The first observation is that, for containment, we only need to focus on boolean queries.

Observation 2 *There is a polynomial time reduction from the containment problem for nested C2RPQs to the containment problem for boolean nested C2RPQs.*

The idea of the reduction is to replace each query $Q(\bar{z})$ of the form (1) over Σ with free variables $\bar{z} = z_1, \ldots, z_n$ with the following boolean query Q^b over an extended alphabet $\Sigma \cup \{\$_1, \ldots, \$_n\}$

$$Q^b = \bigwedge_{1 \leq i \leq m} (x_i, L_i, y_i) \wedge \bigwedge_{1 \leq j \leq n} (z_j, \$_j, z_j) \qquad (2)$$

It is straightforward to show that a query Q_1 is contained in a query Q_2 iff Q_1^b is contained in Q_2^b.

Let us now explain how to decide the containment problem for boolean CRPQs. Let Q_1 and Q_2 be two boolean CRPQs over Σ. The basic idea in [19] is the following. Given two CRPQs, Q_1 and Q_2, we first construct an NFA $\mathcal{A}_1$, of exponential size, that accepts precisely the "codifications" of the graph databases that satisfy Q_1, and then construct an NFA $\mathcal{A}_2$, of double-exponential size, that accepts precisely the "codifications" of the graph databases that do not satisfy Q_2. Then it is possible to prove that $Q_1 \not\subseteq Q_2$ if and only the language accepted by $\mathcal{A}_1 \cap \mathcal{A}_2$ is nonempty. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are of exponential size, the latter can be done in Expspace by using a standard "on-the-fly" verification algorithm [57]. The same work also shows that the containment is also hard for Expspace, so this algorithm is essentially the best one can do. Moreover, the same technique is shown to work when both Q_1 and Q_2 are C2RPQs. The containment problem for CNRPQs was studied indirectly in [11] in the context of graph data exchange, and an EXPSPACE upper bound also follows from [16, 18].

Proposition 4. *The query containment problem for CNRPQs is* EXPSPACE-*complete. It remains* EXPSPACE-*hard even for CRPQs.*

Adding Unions. Further extensions make a case for considering unions of these queries, obtaining the classes of UCRPQs, UC2RPQs and UCNRPQs (where the capital U stands for union). It is not difficult to show that these queries have actually more expressive power than their union-free counterparts, and one can also show that the same bounds hold for both containment and query evaluation problems.

4 Datalog for querying graphs

It is evident that the base navigational languages we introduced in the previous section lack the expressive power to be used as a standalone query language for graph databases. But unfortunately none of extensions we have seen so far (from CRPQs to UCNRPQs) is *algebraically closed*, which is a key disadvantage from both the user and the system point of view. Indeed, algebraic closure has proved to be a prevalent property in several other widely used query languages. To name a few examples, note first that relational algebra is defined as the closure of a set of relational operators [2]. Also, the class of CQs is closed under projection and join, while UCQs are also closed under union [2]. Similarly, the class of 2RPQs is closed under concatenation, union, and transitive closure. In contrast, UC2RPQs and UCNRPQs are not closed under transitive closure, because even the transitive closure of a binary UC2RPQ query is not a UC2RPQ query.

In this section we show how to obtain query languages that are *algebraically closed*. All of these languages are based on Datalog, so we must start by introducing Datalog programs, and showing how they are used to query graphs. The problem, however, is that when using full Datalog as our query language we lose the nice evaluation and containment properties that UCNRPQs enjoy. Is there a navigational graph language that is algebraically closed, but that at the same time enjoys the good properties of UCNRPQs? We answer this positively, with the introduction of Regular Queries.

4.1 Datalog as a graph query language

A Datalog *program* Π consists of a finite set of rules of the form $S(\bar{x}) \leftarrow R_1(\bar{y}_1), \ldots, R_m(\bar{y}_m)$, where $S, R_1, \ldots, R_m$ are predicate symbols and $\bar{x}, \bar{y}_1, \ldots, \bar{y}_m$ are tuples of variables. A predicate that occurs in the head of a rule is called *intensional* predicate. The rest of the predicates are called *extensional* predicates. We assume that each program has a distinguished intensional predicate called *Ans*. Let P be an intensional predicate of a Datalog program Π and I an instance over the schema given by the extensional predicates of Π.

For $i \geq 0$, $P^i_\Pi(I)$ 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 $\bigcup_{i\geq 0} P^i_\Pi(I)$. Then, the *answer* $\Pi(I)$ of Π over I is $Ans^\infty_\Pi(I)$.

A predicate P *depends* on a predicate Q in a Datalog program Π, if Q occurs in the body of a rule ρ of Π and P is the predicate in the head of ρ. The *dependence graph* of Π is a directed graph whose nodes are the predicates of Π and whose edges capture the dependence relation: there is an edge from Q to P if P depends on Q. A program Π is *nonrecursive* if its dependence graph is acyclic, that is, no predicate depends recursively on itself.

We can view Datalog queries as a graph language. In order to do this we proceed just as for CQs: given an alphabet Σ, we use the schema $\sigma(\Sigma)$ that consists of one binary predicate for each symbol $a \in \Sigma$. We can then represent each graph G over Σ as its straightforward relational representation $\mathbf{D}(G)$ over $\sigma(\Sigma)$. A (nonrecursive) Datalog program over a finite alphabet Σ is a (nonrecursive) Datalog program Π whose extensional predicates belong to $\sigma(\Sigma^\pm)$. The *answer* $\Pi(G)$ of a (nonrecursive) Datalog program Π over a graph database G over Σ is $\Pi(\mathbf{D}(G^\pm))$.

The idea of using Datalog as a graph query language comes from Consens and Mendelzon [24], where it was introduced under the name GraphLog, as an alternative to UCRPQs that could express other types of graph properties, in particular those which are not monotone. To keep the complexity low, and to ensure that the intentional predicates in Datalog programs continue to resemble graphs, the arity of all predicates in the programs where restricted to be *binary*. However, GraphLog includes features that we shall not review in this section, such as including negated predicates. Nevertheless, Datalog programs as we have defined here are still enough to express all of the primitives we reviewed in the previous section, as well as conjunctions, unions, and of course more expressive forms of recursion.

Example 5. Consider again the CRPQ in Example 4. It can be expressed via the following Datalog program:

$$\begin{aligned}
Hpath(x,y) &\leftarrow helps(x,y).\\
Hpath(x,z) &\leftarrow Hpath(x,y), helps(y,z).\\
Kpath(x,y) &\leftarrow knows(x,y).\\
Kpath(x,z) &\leftarrow Kpath(x,y), knows(y,z).\\
Ans(x,y) &\leftarrow Hpath(x,y), Kpath(x,y).
\end{aligned}$$

The program uses three intentional predicates: *Hpath*, whose intention is to store all pairs of nodes that belong to the evaluation of the RPQ $helps^+$ (that is, the transitive closure of *helps*), *Kpath*, intended to store the result of $knows^+$, and *Ans*, which selects those pairs that appear both in *Hpath* and *Kpath*.

Datalog can also express NRPQs. For example, consider the NRPQ

$$(knows[helps])^+,$$

which intuitively computes those pairs of nodes connected by a path of people that know one another, but requiring as well that each node in the path is a helper. This query is computed by the program

$$\begin{aligned} N(x,y) &\leftarrow \mathit{knows}(x,y), \mathit{helps}(y,z). \\ \mathit{Ans}(x,y) &\leftarrow N(x,y). \\ \mathit{Ans}(x,z) &\leftarrow \mathit{Ans}(x,y), N(y,z). \end{aligned}$$

4.2 Binary Linear Datalog

The examples in the previous section suggest that Datalog programs subsume all of our navigational primitives, and even all CNRPQs. But we can actually show more: each CNRPQ (and in fact, each UCNRPQ) can be expressed by a fragment of Datalog that is particularly well behaved for query evaluation. We say that a Datalog program Π is *linear* if we can partition its rules into sets $\Pi_1, \dots, \Pi_n$ such that (1) the predicates in the head of the rules in Π_i do not ocurr in the body of any rule in any set Π_j, with $j < i$; and (2) the body of each rule in Π_i has at most one occurrence of a predicate that occurs in the head of a rule in Π_i[1]. A binary linear Datalog program is just a linear program where all intensional predicates have arity 2, except possibly for Ans.

As usual, we say that a language $\mathcal{L}_1$ can be expressed using a language $\mathcal{L}_2$ if for every query in $\mathcal{L}_1$ there is an equivalent query in $\mathcal{L}_2$. If in addition $\mathcal{L}_2$ has a query not expressible in $\mathcal{L}_1$, then $\mathcal{L}_2$ is strictly more expressive than $\mathcal{L}_1$. The languages are equivalent if each can be expressed using the other. They are incomparable if none can be expressed using the other.

Observation 3 *Binary linear Datalog programs are strictly more expressive than UCNRPQs.*

To show that every UCNRPQ can be expressed as a Datalog program we proceed just as in the example above. Unions, conjunctions and concatenations, and the empty string are all straightforwardly expressed in Datalog, and if one has programs that compute expressions R_1 and R_2 into predicates P_{R_1} and P_{R_2}, then the query $R_1[R_2]$ can be computed using the rule $\mathit{And}(x,y) \leftarrow P_{R_1}(x,y), P_{R_2}(y,z)$. Finally, if one has a program to compute R into predicate P_R, then R^+ is given by the predicate P_{R^+}, computed as follows:

$$\begin{aligned} P_{R^+}(x,y) &\leftarrow P_R(x,y). \\ P_{R^+}(x,y) &\leftarrow P_{R^+}(x,z), P_R(z,y). \end{aligned}$$

Moreover, as the following examples show, one can use binary linear Datalog to express a large number of interesting queries that cannot be expressed as UCNRPQs.

[1] These programs are sometimes referred to as *stratified linear* programs, or *piecewise linear Programs* [56].

Example 6. Let us come back to our graph of relationships with labels *knows* and *helps* from Figure 1. We say that a person p is a *friend* of a person p' if p knows and helps p' at the same time. The following program returns all the *indirect* friends, that is, all pairs of people connected by a chain of friends.

$$\begin{aligned}
F(x,y) &\leftarrow \mathit{knows}(x,y), \mathit{helps}(x,y).\\
\mathit{Fchain}(x,y) &\leftarrow F(x,y).\\
\mathit{Fchain}(x,y) &\leftarrow \mathit{Fchain}(x,y), F(y,z).\\
\mathit{Ans}(x,y) &\leftarrow \mathit{Fchain}(x,y).
\end{aligned}$$

Suppose now that a person p' is an *acquaintance* of p if p knows p' and they have an indirect friend in common. The pairs of people connected by a chain of acquaintances can be expressed by the following RQ.

$$\begin{aligned}
F(x,y) &\leftarrow \mathit{knows}(x,y), \mathit{helps}(x,y).\\
\mathit{Fchain}(x,y) &\leftarrow F(x,y).\\
\mathit{Fchain}(x,y) &\leftarrow \mathit{Fchain}(x,y), F(y,z).\\
A(x,y) &\leftarrow \mathit{knows}(x,y), \mathit{Fchain}(x,z), \mathit{Fchain}(y,z).\\
\mathit{Achain}(x,y) &\leftarrow A(x,y).\\
\mathit{Achain}(x,y) &\leftarrow \mathit{Achain}(x,y), A(y,z).\\
\mathit{Ans}(x,y) &\leftarrow \mathit{Achain}(x,y).
\end{aligned}$$

With a little bit of work one can use the techniques from [17] to show that the two queries above cannot be expressed with UCNRPQs. $\square$

Thus, it appears that binary linear Datalog programs are a good candidate for querying graph databases: the language is algebraically closed, and it can express all UCNRPQs. But what are the algorithmic properties of this language? The good news is query evaluation, as we can show that binary linear Datalog programs enjoy the same complexity as the conjunctive languages we have reviewed in the previous section.

Proposition 5 ([23, 24]). *The query evaluation problem for binary linear Datalog programs is* NP*-complete in combined complexity and* NLOGSPACE*-complete in data complexity.*

But, as it usually happens when working with Datalog programs, the containment problem becomes substantially more difficult when we move from UCNRPQs to binary linear Datalog. The following upper bound follows from [25, 26] (there is also a refinement in [18]), while the lower bound follows from lower bounds of slightly less expressive languages in [17].

Proposition 6. *The query containment problem for binary linear Datalog queries is non-elementary.*

As we see, the problem with this language is that we are allowing too much freedom in choosing the way these programs are navigated. Thus, we need to further restrict the language in order to obtain something manageable from the point of view of containment.

4.3 Regular Queries

An *extended Datalog rule* is a rule of the form $S(\bar{x}) \leftarrow R_1(\bar{y}_1), \ldots, R_m(\bar{y}_m)$, where S is a predicate and, for each $1 \leq i \leq m$, R_i is either a predicate or an expression P^+ for a binary predicate P. An *extended* Datalog program is a finite set of extended Datalog rules. For an extended Datalog program, we define its extensional/intensional predicates and its dependence graph in the obvious way. Again we assume that there is a distinguished intensional predicate Ans. As expected, a nonrecursive extended Datalog program over an alphabet Σ is an extended Datalog program whose extensional predicates are in $\sigma(\Sigma^{\pm})$ and whose dependence graph is acyclic.

Intuitively, extended Datalog rules offer some degree of recursion, but the recursion is limited so that it mimics the transitive closure operator of regular expressions. One can further define the language that results of combining multiple of these rules, which is known as Regular Queries [52]. Formally, A *regular query* (RQ) Ω over a finite alphabet Σ is a nonrecursive extended Datalog program over Σ, where all intensional predicates, except possibly for Ans, have arity 2.

The semantics of an extended Datalog rule is defined as in the case of a standard Datalog rule considering the semantics of an atom $P^+(y, y')$ as the pairs (v, v') that are in the transitive closure of the relation P. The semantics of a RQ is then inherited from the semantics of Datalog in the natural way. We denote by $\Omega(G)$ the answer of a RQ Ω over a graph database G.

Example 7. Recall the first query in Example 6, that computed chains of friends over a graph of relationships with labels *knows* and *helps*, and where a person p is a *friend* of a person p' if p knows and helps p' at the same time. The following RQ returns the desired information:

$$\begin{aligned} F(x,y) &\leftarrow knows(x,y), helps(x,y). \\ Ans(x,y) &\leftarrow F^+(x,y). \end{aligned}$$

The second query in Example 6 computed all chains of acquaintances, where a person p' is an *acquaintance* of p if p knows p' and they have an indirect friend in common. This query can be expressed with the following RQ:

$$\begin{aligned} F(x,y) &\leftarrow knows(x,y), helps(x,y). \\ A(x,y) &\leftarrow knows(x,y), F^+(x,z), F^+(y,z). \\ Ans(x,y) &\leftarrow A^+(x,y). \end{aligned}$$

□

Expressive Power. Note that RQs are also a closed language, since in particular the transitive closure of a binary RQ is always a RQ. This makes RQs a natural graph query language, but what about its expressive power?

The first observation is that every NRPQ can be expressed as a regular query, and in fact RQs subsume UCNPQs. In order to provide a translation from UCNRPQs to RQs one can do a construction such as the one in Example 5, except that now the + operator in NRPQs is translated directly as an an expression P^+. Furthermore, we have just shown, in Example 7, that both the *chain-of-friends* and the *chain-of-acquaintances* queries in Example 6 can be expressed as RQs, and since these queries are not expressible as UCNRPQs, it follows that regular queries are strictly more expressive than UCNRPQs.

The next observation is RQs are actually contained in binary linear Datalog. To see this, note first that each expression P^+ in an extended Datalog program can be computed by the following linear Datalog rules (assuming now P^+ is a new predicate):

$$P^+(x,y) \leftarrow P(x,y).$$
$$P^+(x,y) \leftarrow P^+(x,z), P(z,y).$$

Thus, every RQ Ω can be translated in polynomial time into a binary linear Datalog program Π_Ω: one just transforms Ω into a regular Datalog program by treating each of the expressions P^+ as a new predicate, and then adds the rules shown above for each such predicate P^+. It is not difficult to see that the resulting program is indeed linear: since Regular Queries are nonrecursive we can use the same ordering on the rules of Ω to derive a partition for the rules in Π_Ω.

Being a subset of binary linear Datalog, the query evaluation problem for regular queries remains NP-complete in combined complexity and NLogspace-complete in data complexity. Moreover, as promised, the complexity of query containment is elementary.

Proposition 7 (from [52]). *The query containment problem for regular queries is* 2Expspace-*complete.*

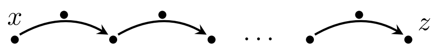
Other Fragments There are several other languages that are either more expressive or incomparable to regular queries. Amongst the list of the most modern ones we have *extended* CRPQs [8], which extends CRPQs with *path variables*, XPath for graph databases [44, 41], and algebraic languages such as [29, 46]. Although all these languages have interesting evaluation properties, the containment problem for all of them is undecidable. Another body of research has focused on fragments of Datalog with decidable containment problems. In fact, regular queries are also investigated in [16, 18] (under the name of *nested positive* 2RPQs). But there are other restrictions that also lead to the decidability of the containment problem. Some of these include Monadic Datalog programs [55, 17], programs whose rules are either guarded or frontier-guarded [17, 18, 15], and first order logic with transitive closure [16]. However, most of these fragments have non-elementary tight bounds for the containment problem, and elementary upper bounds are only obtained when the depth of the programs is fixed. Thus, regular queries seems to be the most expressive fragment of first-order logic with transitive closure that is known to have an elementary containment problem.

5 Moving to RDF

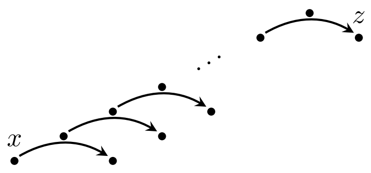
The Semantic Web and its underlying data model, RDF, are usually cited as one of the key applications of graph databases, but there is some mismatch between them. The basic concept of RDF is a *triple* (s, p, o), that consists of the subject s, the predicate p, and the object o, drawn from a domain of internationalised resource identifiers (IRI's). Thus, the middle element need not come from a finite alphabet, and may in addition play the role of a subject or an object in another triple. For instance, $\{(s, p, o), (p, s, o')\}$ is a valid set of RDF triples, but in graph databases, it is impossible to have two such edges.

We take the notion of reachability for granted in graph databases, but what is the corresponding notion for triples, where the middle element can serve as the source and the target of an edge? Then there are multiple possibilities, two of which are illustrated below.

Query $\mathsf{Reach}_{\rightarrow}$ looks for pairs (x, z) connected by paths of the following shape:



and $\mathsf{Reach}_{\nearrow}$ looks for the following connection pattern:



SPARQL, the standard query language for RDF graphs, defines property paths, a navigational query language that resembles 2RPQs, but is more tailored at querying graphs that can draw labels from infinite alphabets (for example, they include an explicit operator $!a$ to specify that two nodes should be connected by an edge labelled with something different from a, which makes little sense when dealing with a finite alphabet). However, property paths do not allow navigation through the middle element in triples, so queries such as $\mathsf{Reach}_{\nearrow}$ cannot be expressed with property paths. To alleviate the situation, [50] propose to treat navigation primitives over a different graph encoding of RDF files that uses a finite alphabet of just three labels, and study the addition of NRPQs over this representation to the language of SPARQL, a language they denote as nSPARQL. We describe both property paths and nSPARQL in Section 5.2

However, as shown by [46], there are natural reachability patterns for triples, similar to those shown above, that *cannot* be defined in graph encodings of RDF

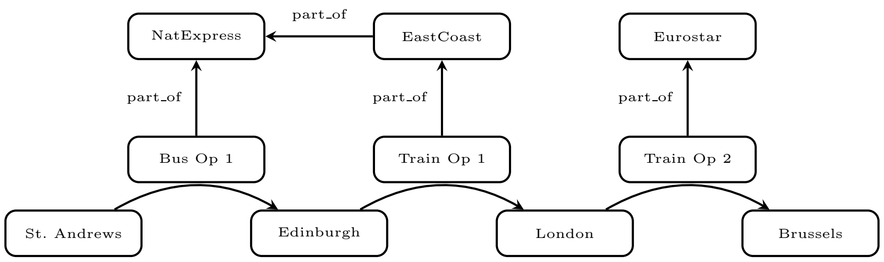


Fig. 2. RDF graph storing information about cities and transport services between them

[6] using nested regular path queries, nor in nSPARQL itself. In fact, queries $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$ demonstrate that there is no such thing as *the reachability* for triples. Moreover, we have again the problem of *closure*, as using graph languages for RDF leads us again to non-composable languages that need two different engines. The alternative is to completely redefine the concept of reachability for RDF graphs. One possibility is to take all possible analogs of compositions of tertiary relation, and devise a navigational language built from these operators. We review this approach in section 5.2.

5.1 Preliminaries

RDF graphs. RDF graphs consist of triples in which, unlike in graph databases, the middle component need not come from a fixed set of labels. Let $\mathbf{I}$ be a countably infinite domain of Internationalized resource identifiers (IRI's). An RDF triple is $(s, p, o) \in \mathbf{I} \times \mathbf{I} \times \mathbf{I}$, where s is referred to as the subject, p as the predicate, and o as the object. An RDF graph is just a collection of RDF triples.[2] We formalise this notion as follows:

Definition 1. *An RDF graph (or a* triplestore*) is a pair $T = (O, E)$, where*

- *$O \subseteq \mathbf{I}$ is a finite set of IRIs;*
- *$E \subseteq O \times O \times O$ is a set of triples; and*
- *for each $o \in O$ there is a triple $t \in E$ such that o appears in T.*

Note that the final condition is used in order to simulate how RDF data is structured in practice, namely that it is presented in terms of sets of triples, so all the objects we are interested in actually appear in the triple relation.

Example 8. The RDF graph T in Figure 2 contains information about cities, modes of transportation between them, and operators of those services. Each triple is represented by an arrow from the subject to the object, with the arrow itself labeled with the predicate. Examples of triples in T are (`Edinburgh`,

[2] To simplify the presentation in this paper we only consider *ground* RDF graphs [50], i.e. RDF graphs which do not contain any blank nodes nor literals.

`Train Op 1`, `London`) and (`Train Op 1`, `part_of`, `EastCoast`). For simplicity, we assume from now on that we can determine implicitly whether an object is a city or an operator. This can of course be modeled by adding an additional outgoing edge labeled `city` from each city and `operator` from each service operator.

5.2 Property Paths

Navigational properties (e.g. reachability) are among the most important functionalities of RDF query languages. In this section we introduce property paths, the W3C standard for querying navigational patterns in RDF, show how they work and how difficult it is to evaluate them, and also discuss some of their shortcomings and some proposals to fix those.

Property paths are a feature of SPARQL, the standard query language for RDF [35], that allow asking navigational queries over RDF graphs. Intuitively, a property path views an RDF document as a labelled graph where the predicate IRI in each triple acts as an edge label. It then extracts each pair of nodes connected by a path such that the word formed by the edge labels along this path belongs to the language of the expression specifying the property path. This idea is of course based on the family of regular path queries for graphs, but as we will see, there are several important differences.

Let us first review the definition of property paths, following the SPARQL 1.1 specification [35]. For consistency we stick to the graph database notation, but note that the standard sometimes uses different symbols for operators; for example, inverse paths e^- and alternative paths $e_1 + e_2$ from our definition are denoted there by $\hat{}e$ and $e_1 \mid e_2$, respectively.

Definition 2. Property paths *are defined by the grammar*

$$e \;:=\; a \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^+ \mid e^* \mid e? \mid !\{a_1, \dots, a_k\} \mid !\{a_1^-, \dots, a_k^-\},$$

where $a, a_1, \dots, a_k$ are IRIs in $\mathbf{I}$. Expressions of the last two forms (i.e., starting with !) are called negated property sets*.*

The definition is based on 2RPQs, with the only difference being negated property sets. When dealing with singleton negated property sets brackets may be omitted: for example, $!a$ is a shortcut for $!\{a\}$. Besides the forms in Definition 2 the SPARQL 1.1 specification includes a third version of the negated property sets $!\{a_1, \dots, a_k, b_1^-, \dots, b_\ell^-\}$, which allows for negating both normal and inverted IRIs at the same time. We however do not include this extra form in our formalisation, since it is equivalent to the expression $!\{a_1, \dots, a_k\} + !\{b_1^-, \dots, b_\ell^-\}$.

The normative semantics for property paths is given in the following definition.

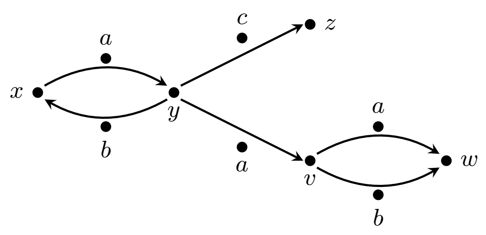


Fig. 3. Illustrating how negated property sets work. Triples in this RDF graph are $(x, a, y), (y, b, x), (y, c, z), (y, a, v), (v, a, w)$ and (v, b, w).

Definition 3. *The* evaluation $[\![e]\!]_T$ *of a property path* e *over an RDF graph* $T = (O, E)$ *is a set of pairs of IRIs from* O *defined as follows:*

$$\begin{aligned}
[\![a]\!]_T &= \{(s, o) \mid (s, a, o) \in E\},\\
[\![e^-]\!]_T &= \{(s, o) \mid (o, s) \in [\![e]\!]_T\},\\
[\![e_1 \cdot e_2]\!]_T &= [\![e_1]\!]_T \circ [\![e_2]\!]_T,\\
[\![e_1 + e_2]\!]_T &= [\![e_1]\!]_T \cup [\![e_2]\!]_T,\\
[\![e^+]\!]_T &= \textstyle\bigcup_{i\geq 1} [\![e^i]\!]_T,\\
[\![e^*]\!]_T &= [\![e^+]\!]_T \cup \{(a, a) \mid a \in O\},\\
[\![e?]\!]_T &= [\![e]\!]_T \cup \{(a, a) \mid a \in O\},\\
[\![!\{a_1, \dots, a_k\}]\!]_T &= \{(s, o) \mid \exists a \textit{ with } (s, a, o) \in E \textit{ and } a \notin \{a_1, \dots, a_k\}\},\\
[\![!\{a_1^-, \dots, a_k^-\}]\!]_T &= \{(s, o) \mid (o, s) \in [\![!\{a_1, \dots, a_k\}]\!]_T\},
\end{aligned}$$

where $\circ$ *is the usual composition of binary relations, and* e^i *is the concatenation* $e \cdot \ldots \cdot e$ *of* i *copies of* e.

As we can see, for the most part, the semantics is the same as when dealing with 2RPQs. The only real difference comes from the interpretation of negated property sets. Intuitively, two IRIs are connected by a negated property set if they are subject and object of a triple in the graph whose predicate is not mentioned in the set under negation. Note that, according to Definition 3, the expression $!\{a_1^-, \dots, a_k^-\}$ retrieves the inverse of $!\{a_1, \dots, a_k\}$, and thus it respects the direction: a negated inverted IRI returns all pairs of nodes connected by some other inverted IRI. To exemplify, consider the RDF graph T from Figure 3. We have that $[\![!a]\!]_T = \{(y, x), (y, z), (v, w)\}$ as we can find a forward looking predicate different from a for any of these pairs. Note that there is an a-labelled edge between v and w, but since there is also a b-labelled one, the pair (v, w) is in the answer. On the other hand, $[\![!a^-]\!]_T = \{(x, y), (z, y), (w, v)\}$, because we can traverse a backward looking predicate (either b^- or c^-) between these pairs.

Note that $!\{a_1, \dots, a_k\}$ is not equivalent to $!a_1 + \ldots + !a_k$. To see this consider again the graph T from Figure 3. We have $[\![!a]\!]_T = \{(y, x), (y, z), (v, w)\}$ and $[\![!b]\!]_T = \{(x, y), (y, z), (y, v), (v, w)\}$, while $[\![!\{a, b\}]\!]_T = \{(y, z)\}$.

Query evaluation and query containment. Syntactically, property paths without negated property sets are nothing more than 2RPQs, with the only minor exception that the empty 2RPQ ε is not expressible as a property path expression. However, negated property sets are a unique feature that we have not reviewed yet. Note that if we were working with graph databases, where predicates come from a finite alphabet Σ, then one could easily replace $!a$ with a disjunction of all other symbols in Σ. But since we are dealing with RDF graphs, which have predicates from the infinite set of IRIs $\mathbf{I}$, we cannot treat this feature in such a naive way. However, one can still show that the query evaluation problem remains in low polynomial time.

Proposition 8. *(from [40]) For every property path e and RDF graph T the problem of deciding whether a pair (a, b) of IRIs belongs to $[\![e]\!]_T$ can be solved in time $O(|T| \cdot |e|)$.*

The idea of the algorithm is to do the usual product of the graph and the automata, now taking into account the negated property sets. We proceed in two steps.

– First we transform $T = (O, E)$ into a graph database $G = (V, E')$ over Σ as follows. Let $Pred(T)$ be the set of all predicates appearing in the triples in T, that is, $Pred(T) = \{p \mid \exists s, o \text{ such that } (s, p, o) \in E\}$. The set Σ of labels is $Pred(T) \cup \{p^- \mid p \in Pred(T)\}$. The set of nodes V is defined as the set of all the objects and subjects appearing in the triples of T. Finally, the set of edges E' contains an edge labelled with p from a node u to a node v if the triple (u, p, v) is in T, and an edge labelled with p^- from u to v if the triple (v, p, u) is in T.
– To do the cross product construction we treat e as an automata over the extended alphabet that includes all negated property sets as additional labels. We can then do the usual cross product construction, except we force a transition labelled with $\{!p_1, \dots, !p_n\}$ in e to be matched to edges in G that are labelled with anything not in $\{p_1, \dots, p_n\}$.

When it comes to query containment one can show that property paths behave similarly as 2RPQs. However, the techniques required to show this now differ slightly due to the inclusion of negated property sets. Just as in the case of 2RPQs, we can show that given two property paths e_1 and e_2, we can check in PSPACE if it holds that $[\![e_1]\!]_T \subseteq [\![e_2]\!]_T$ for every RDF graph T. Interestingly, when we allow conjunctions, projections and unions, the bound for UC2RPQs is still preserved, but the basic ideas for CRPQ containment [19] can no longer be used to prove this directly. Overall, we get:

Proposition 9. *(from [40]). The query containment problem for property paths is* PSPACE*-complete. If we allow combining property paths using union, conjunction and projection, the problem becomes* EXPSPACE*-complete.*

Nested Regular Path Queries in the RDF context. As we have mentioned, the navigational capabilities currently in use in SPARQL are quite limited, in the sense that one cannot define paths that follow through properties such as the one in the query $\mathsf{Reach}_{\nearrow}$. To address this limitation, Pérez et al. [50] propose to use NRPQs over a codification which transforms RDF graphs into graph databases.

Formally, given an RDF graph T, we define the transformation $\delta_{\text{NNE}}(T) = (V, E)$ as a graph database over alphabet $\Sigma = \{\text{next}, \text{node}, \text{edge}\}$, where V contains all IRIs from T, and for each triple (s, p, o) in T, the edge relation E contains edges (s, edge, p), (p, node, o) and (s, next, o). An example of coding an RDF graph using this technique is shown in Figure 4.

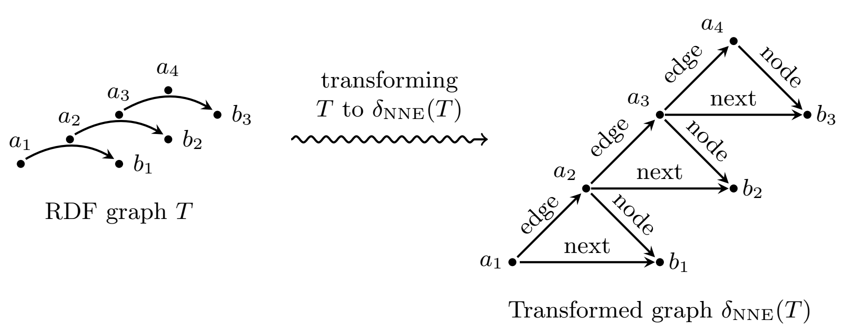

a_4
a_3
a_2
a_1
b_3
b_2
b_1
RDF graph T
transforming
T to $\delta_{\text{NNE}}(T)$
edge
node
next
Transformed graph $\delta_{\text{NNE}}(T)$

Fig. 4. Transforming an RDF graph into a graph database using δ_{NNE}.

Notice that the RDF document from Figure 4 corresponds to a part of the reachability pattern $\mathsf{Reach}_{\nearrow}$ introduced above. As we stated, this type of pattern is not expressible using property paths, but it can be computed as an NPRQ (and in fact, an RPQ) over the translation δ_{NNE} of an RDF graph. To be more precise, one can show that evaluating $\mathsf{Reach}_{\nearrow}$ over an arbitrary RDF graph T yields the same answer as evaluating the RPQ $edge^{+}node$ over the transformation $\delta_{\text{NNE}}(T)$. Besides allowing to express more complicated navigational queries, this transformation scheme was also used in several important practical RDF applications (e.g. to address the problem of interpreting RDFS features within SPARQL [50]). Since the transformation δ_{NNE} can be computed in linear time, using NRPQs over these codification is basically the same as using them over the original RDF graph, from the point of view of computational complexity. In particular, the bounds from Proposition 2 still hold with respect to the size of the RDF graph.

Although more powerful than property paths, NRPQs are still not capable of expressing some queries which one would naturally ask in the RDF context. To see this, consider the transportation network from Figure 2. Suppose one wants to answer the following query:

Q: Find pairs of cities (x, y) such that one can travel from x to y using services operated by the same company.

A query like this is likely to be relevant, for instance, when integrating numerous transport services into a single ticketing interface. For the graph T

in Figure 2, the pair (`Edinburgh`, `London`) belongs to $Q(T)$ (here $Q(T)$ denotes the answer of the query Q over the RDF graph T), and one can also check that (`St. Andrews`, `London`) is in $Q(T)$, since recursively both operators are part of NatExpress (using the transitivity of part_of). However, the pair (`St. Andrews`, `Brussels`) does not belong to $Q(T)$, since we can only travel that route if we change companies, from NatExpress to Eurostar.

If we try to use NRPQs, or even their conjunctive variants, to answer such a query, we immediately run into problem, as we are trying to verify that the company at the end of a chain of `part_of` edges is the same before we proceed to the next city. In essence, to answer the query Q, we would need to find patterns such as the one in Figure 5 in our RDF graph.

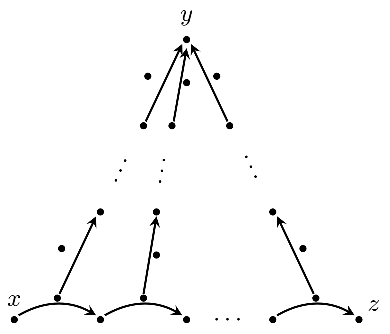


Fig. 5. A pattern required to answer the query Q.

In fact, it was shown in e.g. [46] and [18] that the query Q above can not be expressed using NRPQs over RDF graphs (or their δ_{NNE}-codification). On the other hand, queries such as Q seem quite natural in the context of RDF, so is there an efficient way of specifying and answering them? We give an answer to this question in the following section, where we introduce a (recursive) algebra for RDF graphs which can ask queries such as Q, and many others.

5.3 Native navigation in triples

In previous section we saw that treating RDF graphs as ordinary graph databases might not always allow us to answer the queries we want, and we also showed that the notion of reachability is not as straightforward when dealing with RDF as it is in graph databases. There is also another problem we did not consider so far. Namely, applying graph queries to RDF graphs leaves us with a set of pairs of nodes, while the initial data we started with contained triples. This means that all of these languages violate the *closure* property; that is, they start in one data model (triples), but end up in binary relations. To see why this might be a problem, note that once we obtain an answer to such query over RDF, we can no longer ask another query over this answer. Another way of saying this is that graph queries over RDF data do not *compose*.

So how can we overcome these issues? It is clear that for this we need a language which works directly over triples and which is composable in a sense that it does not leave the initial data model. Natural candidates to start with are of course Datalog, as we have shown in the previous section, but also the relational algebra [2], perhaps the most famous database composable language. We take for now the algebraic approach to language design, and introduce an algebra designed specifically for triples. We start with a plain version and then add recursive primitives that provide the crucial functionality for handling navigational queries. We also show how this algebra can be transformed into a Datalog fragment that resembles the binary linear Datalog programs we defined previously (but of course the binary restriction has to be dropped).

The operations of the usual relational algebra are selection, projection, union, difference, and cartesian product. Our language must remain *closed*, i.e., the result of each operation ought to be a set of triples. This clearly rules out projection. Selection and Boolean operations are fine. Cartesian product, however, would create a relation of arity six, so instead we use *joins* that only keep three positions in the result.

Triple joins. To better understand what kind of joins we need, let us first look at the *composition* of two relations. For binary relations S and S', their composition $S \circ S'$ has all pairs (x, y) so that $(x, z) \in S$ and $(z, y) \in S'$ for some z. We can now define reachability with respect to relation S by recursively applying composition: $S \cup S \circ S \cup S \circ S \circ S \cup \ldots$. Note that this is how RPQs or property paths define reachability. So we need an analog of composition for triples. To understand how it may look, we can view $S \circ S'$ as the *join* of S and S' on the condition that the 2nd component of S equals the first of S', and the output consist of the remaining components. We can write it as

$$S \underset{2=1'}{\overset{1,2'}{\bowtie}} S'$$

Here we refer to the positions in S as 1 and 2, and to the positions in S' as $1'$ and $2'$, so the join condition is $2 = 1'$ (written below the join symbol), and the output has positions 1 and $2'$. This suggests that our join operations on triples should be of the form $R \bowtie^{i,j,k}_{\text{cond}} R'$, where R and R' are ternary relations, $i, j, k \in \{1, 2, 3, 1', 2', 3'\}$, and cond is a condition (to be defined precisely later).

But what is the most natural analog of relational composition? Note that to keep three indexes among $\{1, 2, 3, 1', 2', 3'\}$, we ought to project away three, meaning that two of them will come from one argument, and one from the other. Any such join operation on triples is bound to be *asymmetric*, and thus cannot be viewed as a full analog of relational composition.

So what do we do? Our solution is to add *all* such join operations. Formally, given two ternary relations R and R', *join* operations are of the form

$$R \underset{\theta}{\overset{i,j,k}{\bowtie}} R',$$

where

- $i, j, k \in \{1, 1', 2, 2', 3, 3'\}$,
- θ is a set of (in)equalities between elements in $\{1, 1', 2, 2', 3, 3'\} \cup \mathbf{I}$.

As before, we use the indices $1, 2, 3$ to denote positions in the relation to the left of the join symbol, and $1', 2', 3'$ for the ones to the right. In θ we allow comparing if the elements in some position are equal or different, and we also allow comparing them to some IRI as e.g. property paths do.

The semantics is defined as follows: (o_i, o_j, o_k) is in the result of the join iff there are triples $(o_1, o_2, o_3) \in R$ and $(o_{1'}, o_{2'}, o_{3'}) \in R'$ such that

- each condition from θ holds; that is, if $l = m$ is in θ, then $o_l = o_m$, and if $l = o$, where o is an IRI, is in θ, then $o_l = o$, and likewise for inequalities.

Using triple joins we can now define the language with the desired properties.

Triple Algebra. We now define the expressions of the *Triple Algebra*, or TriAL for short. It is a restriction of relational algebra that guarantees closure over triples, i.e., the result of each expression is an RDF graph.

- The set E of all the triples in an RDF graph is a TriAL expression.
- If e is a TriAL expression and θ a set of equalities and inequalities over $\{1, 2, 3\} \cup \mathbf{I}$, then $\sigma_\theta(e)$ is a TriAL expression, called a *selection expression*.
- If e_1, e_2 are TriAL expressions, then the following are TriAL expressions:
  - $e_1 \cup e_2$;
  - $e_1 - e_2$;
  - $e_1 \bowtie_\theta^{i,j,k} e_2$, with i, j, k, θ as in the definition of the join above.

The semantics of the join operation has already been defined. The semantics of the Boolean operations is the usual one. The semantics of the selection is defined in the same way as the semantics of the join (in fact, the operator itself can be defined in terms of joins): one just chooses triples (o_1, o_2, o_3) satisfying θ.

Given an RDF graph T, we write $e(T)$ for the result of evaluating the expression e over T. Note that $e(T)$ is again an RDF graph, and thus TriAL defines closed operations on triplestores.

Example 9. To get some intuition about the Triple Algebra consider the following TriAL expression:

$$R = E \underset{2=1'}{\overset{1,3',3}{\bowtie}} E$$

Indexes $(1, 2, 3)$ refer to positions of the first triple, and indexes $(1', 2', 3')$ to positions of the second triple in the join. Thus, for two triples (x_1, x_2, x_3) and $(x_{1'}, x_{2'}, x_{3'})$, such that $x_2 = x_{1'}$, expression R outputs the triple $(x_1, x_{3'}, x_3)$. E.g., in the triplestore of Figure 2, (`London`, `Train Op 2`, `Brussels`) is joined with (`Train Op 2`, `part_of`, `Eurostar`), producing (`London`, `Eurostar`, `Brussels`); the full result is the following set of triples

`St. Andrews`	`NatExpress`	`Edinburgh`
`Edinburgh`	`EastCoast`	`London`
`London`	`Eurostar`	`Brussels`

When interpreted over the RDF document from Figure 2, R gives us pairs of European cities together with companies one can use to travel from the first city to the second one. Note that this expression fails to take into account that `EastCoast` is a part of `NatExpress`. To add such information to query results (and produce triples such as (`Edinburgh`, `NatExpress`, `London`)), we use $R' = R \cup (R \bowtie^{1,3',3}_{2=1'} E)$.

Adding Recursion. One problem with Example 9 above is that it does not include triples (`city`$_1$,`service`,`city`$_2$) so that relation R contains a triple (`city`$_1$,`service`$_0$,`city`$_2$), and there is a chain, of some length, indicating that `service`$_0$ is a part of `service`. The second expression in Example 9 only accounted for such paths of length 1. To deal with paths of arbitrary length, we need reachability, which relational algebra is well known to be incapable of expressing. Thus, we need to add recursion to our language.

To do so, we expand TriAL with *right* and *left Kleene closure* of any triple join $\bowtie^{i,j,k}_{\theta}$ over an expression R, denoted as $(R \bowtie^{i,j,k}_{\theta})^*$ for right, and $(\bowtie^{i,j,k}_{\theta} R)^*$ for left. These are defined as

$$\begin{aligned}(R \bowtie)^* &= \emptyset \ \cup \ R \ \cup \ \ R \bowtie R \ \ \cup \ \ (R \bowtie R) \bowtie R \ \ \cup \ \ldots,\\ (\bowtie R)^* &= \emptyset \ \cup \ R \ \cup \ \ R \bowtie R \ \ \cup \ \ R \bowtie (R \bowtie R) \ \cup \ \ldots\end{aligned}$$

We refer to the resulting algebra as *Triple Algebra with Recursion* and denote it by TriAL*.

When dealing with binary relations we do not have to distinguish between left and right Kleene closures, since the composition operation for binary relations is associative. However, as the following example shows, joins over triples are not necessarily associative, which explains the need to make this distinction.

Example 10. Consider an RDF graph $T = (O, E)$, with $E = \{(a, b, c), (c, d, e), (d, e, f)\}$. The expression

$$R_1 = (E \underset{3=1'}{\overset{1,2,2'}{\bowtie}})^*$$

computes $R_1(T) = E \cup \{(a, b, d), (a, b, e)\}$, while

$$R_2 = (\underset{3=1'}{\overset{1,2,2'}{\bowtie}} E)^*$$

computes $R_2(T) = E \cup \{(a, b, d)\}$.

Now we show how to formulate the queries mentioned earlier in this section using Triple Algebra.

Example 11. We started Section 5 by saying how there are different types of reachability over RDF graphs, and presented two queries, $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$, which illustrate this claim. It is easy to see that $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$ can be expressed using the TriAL^* expressions E_1 and E_2 below:

$$E_1 = (E \underset{3=1'}{\overset{1,2,3'}{\bowtie}})^* \quad \text{and} \quad E_2 = (\underset{1=2'}{\overset{1',2',3}{\bowtie}} E)^*.$$

Next consider the query Q from Section 5.2. Recall that in this query we are asking for all pair of cities such that one can travel from the first city to the second one using services provided by the same company in a travel network such as the one presented in Figure 2. Abstracting away from a particular RDF graph, this query asks us to find patterns of the form illustrated in Figure 5. Although not expressible using property paths, or NRPQs (over codifications of RDF graphs), we can express this query using the following TriAL^* expression:

$$((E \underset{2=1'}{\overset{1,3',3}{\bowtie}})^* \underset{3=1',2=2'}{\overset{1,2,3'}{\bowtie}})^*.$$

Note that the interior join $(E \underset{2=1'}{\overset{1,3',3}{\bowtie}})^*$ computes all triples (x, y, z), such that $E(x, w, z)$ holds for some w, and y is reachable from w using some E-path. The outer join now simply computes the transitive closure of this relation, taking into account that the service that witnesses the connection between the cities is the same.

Datalog for RDF. Triple Algebra and its recursive versions are in their essence *procedural* languages. It would therefore be nice to see a more declarative option for specifying TriAL queries. We have already seen a good candidate for capturing algebraic and recursive properties of a language in Section 4, that is: Datalog. So it seems natural to look for Datalog fragments that capture TriAL and its recursive version.

Since Datalog works over relational vocabularies, once again we need to explain how to interpret an RDF graph $T = (O, E)$ as a relational structure. However, this is rather straightforward: our relation schema will consist of a single ternary relation E, and in an instance I_T of this schema the interpretation of the relation E is equal to the relation E which stores all the triples in T. Using this we can now describe a Datalog fragment called $\mathsf{TripleDatalog}$, which captures TriAL.

A $\mathsf{TripleDatalog}$ rule is of the form

$$S(\overline{x}) \;\leftarrow\; S_1(\overline{x_1}), S_2(\overline{x_2}), u_1 = v_1, \ldots, u_m = v_m \tag{3}$$

where

1. S, S_1 and S_2 are (not necessarily distinct) predicate symbols of arity 3;
2. $\overline{x}$, $\overline{x_1}$ and $\overline{x_2}$ are variables;

3. u_is and v_is are either variables or IRIs from $\mathbf{I}$;
4. all variables in $\overline{x}$ and all variables in u_j, v_j are contained in $\overline{x_1} \cup \overline{x_2}$.

A $\mathsf{TripleDatalog}^{\neg}$ rule is like the rule (3) but all equalities and predicates, except the head predicate S, can appear negated. A $\mathsf{TripleDatalog}^{\neg}$ *program* Π is a finite set of $\mathsf{TripleDatalog}^{\neg}$ rules. Such a program Π is *non-recursive* if there is an ordering $r_1, \ldots, r_k$ of the rules of Π so that the relation in the head of r_i does not occur in the body of any of the rules r_j, with $j \leq i$.

As is common with non-recursive programs, the semantics of nonrecursive $\mathsf{TripleDatalog}^{\neg}$ programs is given by evaluating each of the rules of Π, according to the order $r_1, \ldots, r_k$ of its rules, and taking unions whenever two rules have the same relation in their head (see [2] for the precise definition). We are now ready to present the first capturing result.

Proposition 10. TriAL *is equivalent to nonrecursive* $\mathsf{TripleDatalog}^{\neg}$ *programs.*

Of course, here we are more interested in expressing navigational properties, so we now turn to TriAL^*, the recursive variant of Triple Algebra. To capture it, we of course add recursion to Datalog rules, and impose a restriction that was previously used in [24]. A $\mathsf{ReachTripleDatalog}^{\neg}$ *program* is a (potentially recursive) $\mathsf{TripleDatalog}^{\neg}$ program in which each recursive predicate S is the head of exactly two rules of the form:

$$\begin{array}{rcl} S(\overline{x}) & \leftarrow & R(\overline{x}) \\ S(\overline{x}) & \leftarrow & S(\bar{x}_1), R(\bar{x}_2), V(y_1, z_1), \ldots, V(y_k, z_k) \end{array} \tag{4}$$

where each $V(y_i, z_i)$ is one of the following: $y_i = z_i$, or $y_i \neq z_i$, and R is a nonrecursive predicate of arity 3, or a recursive predicate defined by a rule of the form 4 that appears before S. These rules essentially mimic the standard linear reachability rules (for binary relations) in Datalog, and in addition one can impose equality and inequality constraints, along the paths.

Note that the negation in $\mathsf{ReachTripleDatalog}^{\neg}$ programs is *stratified.* The semantics of these programs is the standard least-fixpoint semantics [2]. The language GraphLog corresponds to almost the same syntactic class, except it is defined for graph databases, rather than triplestores. Interestingly, one can show that these classes capture the expressive power of FO with the transitive closure operator [24]. In our case, we have a capturing result for TriAL^*.

Theorem 4. *The expressive power of* TriAL^* *and* $\mathsf{ReachTripleDatalog}^{\neg}$ *programs is the same.*

We now give an example of a simple Datalog program computing the query Q from Section 5.2 and Example 11.

Example 12. The following $\mathsf{ReachTripleDatalog}^{\neg}$ *program is equivalent to query* Q *from Section 5.2. Note that the answer is computed in the predicate* Ans.

$$\begin{aligned} S(x_1, x_2, x_3) &\leftarrow E(x_1, x_2, x_3) \\ S(x_1, x_3', x_3) &\leftarrow S(x_1, x_2, x_3), E(x_2, x_2', x_3') \\ Ans(x_1, x_2, x_3) &\leftarrow S(x_1, x_2, x_3) \\ Ans(x_1, x_2, x_3') &\leftarrow Ans(x_1, x_2, x_3), S(x_3, x_2, x_3') \end{aligned}$$

Recall that this query can be written in TriAL^* *as* $Q = ((E \Join^{1,3',3}_{2=1'})^* \Join^{1,2,3'}_{3=1',2=2'})^*$. *The predicate* S *in the program computes the inner Kleene closure of the query, while the predicate* Ans *computes the outer closure.*

Query evaluation and query containment over RDF graphs. We have seen that TriAL^* is a powerful language capable of expressing a wide range of queries over RDF graphs. The question is then, can we evaluate these queries efficiently? As before, to answer this question, we will look at the query evaluation problem, which asks, given an RDF graph T, TriAL^* expression e, and a triple t, if it is true that $t \in e(T)$.

From previous sections we know that many graph query languages (RPQs, NRPQs) have a PTIME upper bound for the evaluation problem, and the data complexity (i.e. when e is assumed to be fixed) is generally NLOGSPACE (which can not be improved since basic reachability is already NLOGSPACE-hard). It can be shown that similar bounds hold for Triple algebra, and even its recursive variant.

Proposition 11. *(from [46]) The query evaluation problem for* TriAL^* *queries is* PTIME*-complete, and it is* NLOGSPACE*-complete when the algebra expression is fixed.*

Of course, the high expressive power of TriAL^* has to come with price, and this is reflected when we consider the query containment problem. Combining the fact that TriAL^* subsumes a variant of XPath over graphs [46], and the fact that the latter has undecidable query containment [42], we obtain the following:

Proposition 12. *The query containment problem for* TriAL^* *expressions is undecidable.*

5.4 More expressive languages

In previous sections we introduced several popular languages for extracting basic navigational patterns from RDF graphs, and showed that they can be evaluated efficiently. However, there are still many interesting properties of RDF graphs that cannot be expressed using these languages, so several more expressive formalisms for extracting information from RDF data have been proposed in the literature. The first one we would like to mention is TriQ [5], which is a Datalog-based language for expressing RDF queries, and which subsumes property paths, NRPQs and TriAL*. In its essence TriQ can be viewed as an extension of the Datalog characterisation of TriAL which does not place such severe restriction on the shape of the recursion as ReachTripleDatalog$^\neg$ does, so it allows us to express more powerful queries while still keeping good query evaluation properties. On a more practical side there has been quite a bit of work on efficiently implementing property paths and their extensions, starting from engines exclusively dedicated to fine-tune the performance of reachability queries over RDF [33, 58], to systems supporting fully recursive queries with SPARQL as their base [7, 53].

Finally, we would like to mention that all of the previously mentioned languages work under the classic assumption that we have all of the data available locally and can process and rewrite it as needed. However, when we consider the native setting where RDF data is used, namely, the Web, this assumption is no longer valid, since it is no longer feasible to keep all the data locally, and it is often not possible to run usual query evaluation algorithms which compute the entire set of answers, but we need to limit ourselves to a subset, or an approximation of the answers which would be available if we were able to have the data locally. In particular, several basic principles of navigational query languages have to be refined to work in this context, and there has been some recent work [36, 37, 28] proposing how to do this over the data published under the Linked Data initiative [13], whose aim is to encourage the publishing of RDF data in a way that allows connecting datasets residing on different servers into one big dataset by enabling them to reference each other.

6 Conclusion

The problem of defining and evaluating navigational graph queries has been the subject of numerous studies in recent years, and we now have several options of languages available for use, that offer a wide range of querying possibilities under a relatively low computational cost. But there is still much work to be done. For example, there are several other options for obtaining closed languages that we have not reviewed, and that so far have received far less attention than the Datalog variants. One such option is to directly define iterations and transitive closures of any graph pattern [38], and there are many other possibilities to explore, each of which having their own algorithmic challenges.

The second important problem that need to be dealt with is that of queries capable of returning the complete paths (see for example [8]). This feature is widely used and needed in practice, but yet the theoretical studies are just starting to appear, and there is still no consensus on what is a good way of integrating navigational queries with the ability to return paths.

Acknowledgements. We would like to thank Pablo Barceló, Jorge Pérez, Miguel Romero, Moshe Vardi, Egor Kostlyev and Leonid Libkin for helpful discussions on the topics of this paper. The authors are supported by the Millennium Nucleus Center for Semantic Web Research Grant NC120004.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kauffman, 1999.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
3. R. Angles. A comparison of current graph database models. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 171–177. IEEE, 2012.
4. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.
5. 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, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 14–26, 2014.
6. M. Arenas, C. Gutierrez, and J. Pérez. *Foundations of RDF databases*. Springer, 2009.
7. M. Atzori. Computing recursive SPARQL queries. In *2014 IEEE International Conference on Semantic Computing, Newport Beach, CA, USA, June 16-18, 2014*, pages 258–259, 2014.
8. P. Barcelo, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *ACM TODS*, 37(4):31, 2012.
9. P. Barceló, L. Libkin, and J. L. Reutter. Querying regular graph patterns. *Journal of the ACM (JACM)*, 61(1):8, 2014.
10. P. Barceló, J. Pérez, and J. L. Reutter. Relative expressiveness of nested regular expressions. In *AMW*, pages 180–195, 2012.
11. P. Barceló, J. Pérez, and J. L. Reutter. Schema mappings and data exchange for graph databases. In *to appear in ICDT*, page TBD, 2013.
12. P. Barceló Baeza. Querying graph databases. In *PODS*, pages 175–188. ACM, 2013.
13. T. Berners-Lee. Linked data. http://www.w3.org/DesignIssues/LinkedData.html, 2006.
14. M. Bienvenu, D. Calvanese, M. Ortiz, and M. Simkus. Nested regular path queries in description logics. In *KR*, 2014.
15. M. Bienvenu, M. Ortiz, and M. Simkus. Navigational queries based on frontier-guarded datalog: Preliminary results. In *Alberto Mendelzon International Workshop on Foundations of Data Management*, page 162, 2015.
16. P. Bourhis, M. Krötzsch, and S. Rudolph. How to best nest regular path queries. In *Description Logics*, 2014.
17. P. Bourhis, M. Krötzsch, and S. Rudolph. Query containment for highly expressive datalog fragments. *arXiv preprint arXiv:1406.7801*, 2014.
18. P. Bourhis, M. Krötzsch, and S. Rudolph. Reasonable highly expressive query languages. In *IJCAI*, pages 2826–2832, 2015.
19. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Containment of conjunctive regular path queries with inverse. In *7th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 176–185, 2000.
20. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Rewriting of regular expressions and regular path queries. *Journal of Computer and System Sciences*, 64(3):443–465, 2002.

21. D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query answering and query containment over semistructured data. In *DBPL*, pages 40–61, 2001.
22. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.
23. S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 55–66. ACM, 1992.
24. M. Consens and A. Mendelzon. Graphlog: A visual formalism for real life recursion. In *9th ACM Symposium on Principles of Database Systems (PODS)*, pages 404–416, 1990.
25. B. Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.
26. B. Courcelle. Recursive queries and context-free graph grammars. *Theoretical Computer Science*, 78(1):217–244, 1991.
27. I. Cruz, A. Mendelzon, and P. Wood. A graphical query language supporting recursion. In *ACM Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD)*, pages 323–330, 1987.
28. V. Fionda, G. Pirrò, and C. Gutierrez. Nautilod: A formal language for the web of data graph. *TWEB*, 9(1):5:1–5:43, 2015.
29. G. H. Fletcher, M. Gyssens, D. Leinders, D. Surinx, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs. *Information Sciences*, 298:390–406, 2015.
30. D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *17th ACM Symposium on Principles of Database Systems (PODS)*, pages 139–148, 1998.
31. G. Gottlob and C. Koch. Logic-based web information extraction. *SIGMOD Record*, 33(2):87–94, 2004.
32. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.
33. A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling kleene: fast property paths in RDF-3X. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, page 14, 2013.
34. A. Gupta, I. S. Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
35. S. Harris, A. Seaborne, and E. Prudhommeaux. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.
36. O. Hartig and J. Pérez. LDQL: A query language for the web of linked data. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, pages 73–91, 2015.
37. O. Hartig and G. Pirrò. A context-based semantics for SPARQL property paths over the web. In *The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings*, pages 71–87, 2015.
38. H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.
39. G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

40. E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoč. Sparql with property paths. In *The Semantic Web-ISWC 2015*, pages 3–18. Springer, 2015.
41. E. V. Kostylev, J. L. Reutter, and D. Vrgoč. Containment of data graph queries. In *ICDT*, pages 131–142, 2014.
42. E. V. Kostylev, J. L. Reutter, and D. Vrgoč. Static analysis of navigational xpath over graph databases. *Inf. Process. Lett.*, 116(7):467–474, 2016.
43. M. Lenzerini. Data integration: a theoretical perspective. In *PODS*, pages 233–246, 2002.
44. L. Libkin, W. Martens, and D. Vrgoč. Querying graph databases with xpath. In *Proceedings of the 16th International Conference on Database Theory*, pages 129–140. ACM, 2013.
45. L. Libkin, W. Martens, and D. Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14, 2016.
46. L. Libkin, J. Reutter, and D. Vrgoč. Trial for rdf: adapting graph query languages for rdf data. In *PODS*, pages 201–212. ACM, 2013.
47. N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras. Dex: A high-performance graph database management system. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 124–127, 2011.
48. A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.
49. A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory, College Park, Maryland, USA, October 25-27, 1972*, pages 125–129, 1972.
50. J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010.
51. J. L. Reutter. Containment of nested regular expressions. *arXiv preprint arXiv:1304.2637*, 2013.
52. J. L. Reutter, M. Romero, and M. Y. Vardi. Regular queries on graph databases. In *18th International Conference on Database Theory (ICDT 2015)*, volume 31, pages 177–194, 2015.
53. J. L. Reutter, A. Soto, and D. Vrgoč. Recursion in SPARQL. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, pages 19–35, 2015.
54. I. Robinson, J. Webber, and E. Eifrem. *Graph Databases: New Opportunities for Connected Data.* " O'Reilly Media, Inc.", 2015.
55. S. Rudolph and M. Krötzsch. Flag & check: Data access with monadically defined queries. In *Proceedings of the 32nd symposium on Principles of database systems*, pages 151–162. ACM, 2013.
56. J. D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.
57. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, August 27 - September 3, 1995, Proceedings)*, pages 238–266, 1995.
58. N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1875–1889, 2016.