TriAL for RDF: Adapting Graph Query Languages for RDF Data

Leonid Libkin
University of Edinburgh
libkin@ed.ac.uk

Juan Reutter
University of Edinburgh and
Catholic University of Chile
juan.reutter@ed.ac.uk

Domagoj Vrgoč
University of Edinburgh
domagoj.vrgoc@ed.ac.uk

ABSTRACT

Querying RDF data is viewed as one of the main applications of graph query languages, and yet the standard model of graph databases – essentially labeled graphs – is different from the triples-based model of RDF. While encodings of RDF databases into graph data exist, we show that even the most natural ones are bound to lose some functionality when used in conjunction with graph query languages. The solution is to work directly with triples, but then many properties taken for granted in the graph database context (e.g., reachability) lose their natural meaning.

Our goal is to introduce languages that work directly over triples and are closed, i.e., they produce sets of triples, rather than graphs. Our basic language is called TriAL, or Triple Algebra: it guarantees closure properties by replacing the product with a family of join operations. We extend TriAL with recursion, and explain why such an extension is more intricate for triples than for graphs. We present a declarative language, namely a fragment of datalog, capturing the recursive algebra. For both languages, the combined complexity of query evaluation is given by low-degree polynomials. We compare our languages with relational languages, such as finite-variable logics, and previously studied graph query languages such as adaptations of XPath, regular path queries, and nested regular expressions; many of these languages are subsumed by the recursive triple algebra. We also provide examples of the usefulness of TriAL in querying graph, RDF, and social networks data.

Categories and Subject Descriptors. F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic; H.2.1 [**Database Management**]: Logical Design—*Data Models*; H.2.3 [**Database management**]: Languages—*Query Languages*

General Terms. Theory, Languages, Algorithms

Keywords. RDF, Triple Algebra, Query evaluation

1. INTRODUCTION

Graph data management is currently one of the most active research topics in the database community, fueled by the adoption of graph models in new application domains, such as social networks, bioinformatics and astronomic databases, and projects such as the Web of Data and the Semantic Web. There are many proposals for graph query languages; we now understand many issues related to query evaluation over graphs, and there are multiple vendors offering graph database products, see [2, 3, 14, 37] for surveys.

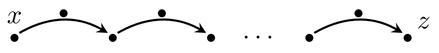
The Semantic Web and its underlying data model, RDF, are usually cited as one of the key applications of graph databases, but there is some mismatch between them. The standard model of graph databases [2, 37] that dates back to [12, 13], is that of directed edge-labeled graphs, i.e., pairs $G = (V, E)$, where V is a set of vertices (objects), and E is a set of labeled edges. Each labeled edge is of the form (v, a, v'), where v, v' are nodes in V, and a is a label from some finite labeling alphabet Σ. As such, they are the same as labeled transition systems used as a basic model in both hardware and software verification. Graph databases of course can store data associated with their nodes (e.g., information about each person in a social network).

The model of RDF data is very similar, yet slightly different. The basic concept is a *triple* (s, p, o), that consists of the subject s, the predicate p, and the object o, drawn from a domain of uniform resource identifiers (URI'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.

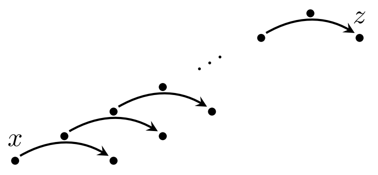
To understand why this mismatch is a problem, consider querying graph data. Since graph databases and RDF are represented as relations, relational queries can be applied to them. But crucially, we may also query the *topology* of a graph. For instance, many graph query languages have, as their basic building block, *regular path queries*, or RPQs [13], that find nodes reachable by a path whose label belongs to a regular language.

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:



But can such patterns be defined by existing RDF query languages? Or can they be defined by existing graph query languages under some graph encoding of RDF?

To answer these, we need to understand which navigational facilities are available for RDF data. A recent survey of graph database systems [3] shows that, by and large, they either offer support for triples, or they do graphs and then can express proper reachability queries. An attempt to add navigation to RDF languages was made in [33], where a language called nSPARQL was defined by taking SPARQL [22, 32], the standard query language for RDF, and extending it with a navigational mechanism provided by *nested regular expressions.* These are essentially regular path queries with XPath-inspired node tests. The evaluation of those uses essentially a graph encoding of RDF. As the starting point of our investigation, we show that there are natural reachability patterns for triples, similar to those shown above, that *cannot* be defined in graph encodings of RDF [5] using nested regular expressions, nor in nSPARQL itself.

Thus, navigational patterns over triples are beyond reach of both RDF languages and graph query languages that work on encodings of RDF. The solution is then to design languages that work directly on RDF triples, and have both relational and navigational querying facilities, just like graph query languages. Our goal, therefore, is to adapt graph database techniques for direct RDF querying.

A crucial property of a query language is *closure*: queries should return objects of the same kind as their input. Closed languages, therefore, are compositional: their operators can be applied to results of queries. Using graph languages for RDF suffers from non-compositionality: for instance, RPQs return graphs rather than triples. So we start by defining a closed language for triples. To understand its basic operations, we first look at a language that has essentially first-order expressivity, and then add navigational features.

We take relational algebra as the basic language. Clearly projection violates closure so we throw it away. Selection and set operations, on the other hand, are fine. The problematic operation is Cartesian product: if T, T' are sets of triples, then $T \times T'$ is not a set of triples but rather a set of 6-tuples. What do we do then? We shall need reachability in the language, and for graphs, reachability is computed by iterating *composition* of relations. The composition operation for binary relations preserves closure: a pair (x, y) is in the composition $R \circ R'$ of R and R' iff $(x, z) \in R$ and $(z, y) \in R'$ for some z. So this is a join of R and R' and it seems that what we need is it analog for triples.

But queries $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$ demonstrate that there is no such thing as *the reachability* for triples. In fact, we shall see that there is not even a nice analog of composition for triples. So instead, we add *all* possible joins that keep the algebra closed. The resulting language is called *Triple Algebra*, denoted by TriAL. We then add an iteration mechanism to it, to enable it to express reachability queries based on different joins, and obtain *Recursive Triple Algebra* TriAL^*.

The algebra TriAL^* can express both reachability patterns above, as well as queries we prove to be inexpressible in nSPARQL. It has a declarative language associated with it, a fragment of Datalog. It has good query evaluation bounds: combined complexity is (low-degree) polynomial. Moreover, we exhibit a fragment with complexity of the order $O(|e| \cdot |O| \cdot |T|)$, where e is the query, O is the set of objects in the database, and T is the set of triples. This is a very natural fragment, as it restricts arbitrary recursive definitions to those essentially defining reachability properties.

The model we use is slightly more general than just triples of objects and amounts to combining triplestores as in, e.g., [24] with the representation of objects used in the Neo4j database [14, 31]. Each object participating in a triple comes associated with a set of attributes. Of course this can be modeled via more triples, but the model we use is conceptually cleaner and leads to a more natural comparison with other query languages.

The first of those comparisons is with relational querying. We show that TriAL lives between FO^3 and FO^6 (recall that FO^k refers to the fragment of First-Order Logic using only k variables). In fact it contains FO^3, is contained in FO^6, and is incomparable with FO^4 and FO^5. A similar results holds for TriAL^* and transitive closure logic.

On the graph querying side, we show that the navigational power of TriAL^* subsumes that of both regular path queries and nested regular expressions. In fact it subsumes a version of *graph XPath* recently proposed for graph databases [27]. We also compare it with conjunctive RPQs [12] and some of their extensions studied in [10, 11]. When it comes to graphs with data held in their nodes, we show that TriAL* continues to subsume some of the formalisms proposed in that context, such as graph XPath expanded with node tests and some types of regular expressions with data values [28, 27].

This shows that TriAL* is an expressive language that subsumes a number of well known relational and graph formalisms, that permits navigational queries not expressible on graph encodings of RDF or in nSPARQL, and that has good query evaluation properties.

Organization In Section 2 we review graph and RDF databases, and describe our model. We also show that some natural navigational queries over triples cannot be expressed in languages such as nSPARQL. In Section 3 we define TriAL and TriAL* and study their expressiveness. In Section 4 we give a declarative language capturing TriAL*. In Section 5 we study query evaluation, and in Sections 6.1 and 6.2 we study our languages in connection with relational and graph querying.

2. GRAPH DATABASES AND RDF

Basic Definitions

Graph databases. We now review some standard definitions (see, e.g., [2, 11, 37]). A graph database is just a finite edge-labeled graph in which each node has a data value attached. Formally, let $\mathcal{N}$ be a countably infinite set of *node ids*, Σ a finite alphabet and $\mathcal{D}$ a countably infinite set of data values. Then a *graph database* over Σ is a triple $G = (V, E, \rho)$, where $V \subset \mathcal{N}$ is a finite set of nodes, $E \subseteq V \times \Sigma \times V$ is a set of labeled edges, and $\rho : V \to \mathcal{D}$ is a function assigning a data value to each node. Each edge is a triple (u, a, v), whose interpretation is an a-labeled edge from u to v. When Σ is clear from the context, we shall simply speak of a graph database. If we work with graph databases that make no use of data values, we write $G = (V, E)$ and disregard the function ρ.

A *path* π from u_0 to u_m in G is a sequence (u_0, a_0, u_1), $(u_1, a_1, u_2), \cdots, (u_{m-1}, a_{m-1}, u_m)$, where each (u_i, a_i, u_{i+1}), for $i < m$, is an edge in E. The *label* of π, denoted by $\lambda(\pi)$, is the word $a_0 \cdots a_{m-1} \in \Sigma^*$.

Regular path queries. Typical navigational languages for graph databases use *regular path queries*, or *RPQ*s [13] as the basic building block. An RPQ is an expression $x \xrightarrow{L} y$, where x and y are variables and L is a regular language over Σ. Given a graph database $G = (V, E)$ over Σ, it defines pairs of nodes (u, v) such that there is a path π from u to v with $\lambda(\pi) \in L$.

Nested regular expressions. These expressions, abbreviated as NRE, over a finite alphabet Σ, extend ordinary regular expressions with the nesting operator (essentially the node test of XPath) and inverses [8, 33]. Formally they are defined as follows:

$$e := \varepsilon \mid a \mid a^- \mid e \cdot e \mid e^* \mid e + e \mid [e], \quad a \in \Sigma.$$

An NRE defines, over a graph $G = (V, E)$, a binary relation on V. The semantics of ε is the diagonal $\{(u, u) \mid u \in V\}$; the semantics of a is the set $\{(u, v) \mid (u, a, v) \in E\}$ of a-labeled edges, and a^- defines $\{(u, v) \mid (v, a, u) \in E\}$. Operations $\cdot$, $+$, and $*$ denote composition, union, and transitive closure of binary relations. Finally, the node test $[e]$ defines pairs (u, u) so that (u, v) is in the result of e for some $v \in V$.

RDF databases. RDF databases contain triples in which, unlike in graph databases, the middle component need not come from a fixed set of labels. Formally, if $\mathbf{U}$ is a countably infinite domain of uniform resource identifiers (URI's), then an RDF triple is $(s, p, o) \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$, 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. Here we deal with *ground* RDF documents [33], i.e., we do not consider blank nodes or literals in RDF documents (otherwise we need to deal with disjoint domains, which complicates the presentation).

Example 1. The RDF database D in Figure 1 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 D are (`Edinburgh`, `Train Op 1`, `London`) and (`Train Op 1`, `part_of`, `EastCoast`). For simplicity we assume that we can determine when and object is a city or an operator implicitly. This can of course be modeled by adding an additional outgoing edge labeled `city` from each city and `operator` from each service operator.

Graph Queries for RDF

Navigational properties (e.g., reachability patterns), are among the most important functionalities of RDF query languages. However, typical RDF query languages, such as SPARQL, are in spirit relational languages. To extend them with navigation, as in [33, 4, 30], one typically uses features inspired by graph query languages, surveyed briefly earlier. Nonetheless, such approaches have their inherent limitations, as we explain here.

Looking again at the database D in Figure 1, we see the main difference between graphs and RDF: the majority of the edge labels in D are also used as subjects or objects (i.e., nodes) of other triples of D. For instance, one can travel from Edinburgh to London by using a train service Train Op 1, but in this case the label itself is viewed as a node when we express the fact that this operator is actually a part of EastCoast trains.

For RDF, one normally uses a model of *triplestores* that is different from graph databases. According to

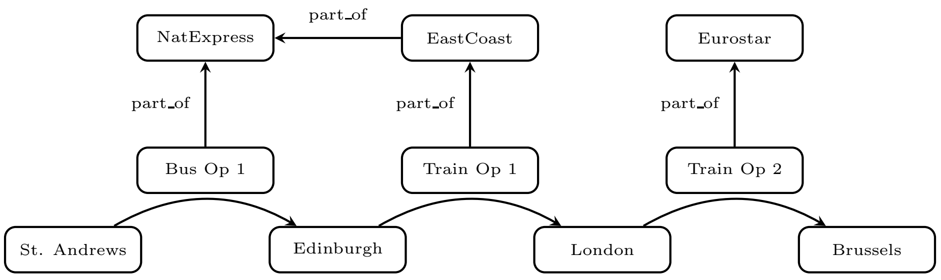


Figure 1: RDF graph storing information about cities and transport services between them

it, the database from Figure 1 is viewed as a ternary relation:

`St. Andrews`	`Bus Op 1`	`Edinburgh`
`Edinburgh`	`Train Op 1`	`London`
`London`	`Train Op 2`	`Brussels`
`Bus Op 1`	`part_of`	`NatExpress`
`Train Op 1`	`part_of`	`EastCoast`
`Train Op 2`	`part_of`	`Eurostar`
`EastCoast`	`part_of`	`NatExpress`

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. In our example, the pair (`Edinburgh`, `London`) belongs to $Q(D)$, and one can also check that (`St. Andrews`, `London`) is in $Q(D)$, since recursively both operators are part of NatExpress (using the transitivity of part_of). However, the pair (`St. Andrews`, `Brussels`) does not belong to $Q(D)$, since we can only travel that route if we change companies, from NatExpress to Eurostar.

To enhance SPARQL with navigational properties, [33] added nested regular expressions to it, resulting in a language called nSPARQL. The idea was to combine the usual reachability patterns of graph query languages with the XPath mechanism of node tests. However, nested regular expressions, which we saw earlier, are defined for graphs, and not for databases storing triples. Thus, they cannot be used directly over RDF databases; instead, one needs to transform an RDF database D into a graph first. An example of such transformation $D \to \sigma(D)$ was given in [5]; it is illustrated in Figure 2.

Formally, given an RDF document D, the graph $\sigma(D) = (V, E)$ is a graph database over alphabet $\Sigma = \{\text{next}, \text{node}, \text{edge}\}$, where V contains all resources from D, and for each triple (s, p, o) in D, the edge relation E contains edges (s, edge, p), (p, node, o) and (s, next, o). This transformation scheme is important in practical RDF applications (it was shown to be crucial for addressing the problem of interpreting RDFS features within SPARQL [33]). At the same time, it is not sufficient for expressing simple reachability patterns like those in query Q:

Proposition 1. *The query Q is not expressible by NREs over graph transformations $\sigma(\cdot)$ of ternary relations.*

Thus, the most common RDF navigational mechanism cannot express a very natural property, essentially due to the need to do so via a graph transformation.

One might argue that this result is due to the shortcomings of a specific transformation (however relevant to practical tasks it might be). So we ask what happens in the native RDF scenario. In particular, we would like to see what happens with the language nSPARQL [33], which is a proper RDF query language extending SPARQL with navigation based on nested regular expressions. But this language falls short too, as it fails to express the simple reachability query Q.

Theorem 1. *The query Q above cannot be expressed in nSPARQL.*

The key reason for these limitations is that the navigation mechanisms used in RDF languages are graph-based, when one really needs them to be triple-based.

Triplestore Databases

To introduce proper triple-based navigational languages, we first define a simple model of triplestores, and show its usefulness in another application area of graph databases, namely social networks.

Let $\mathcal{O}$ be a countably infinite set of objects, and $\mathcal{D}$ be a countably infinite set of data values.

Definition 1. *A* triplestore database, *or just* triplestore *over $\mathcal{D}$ is a tuple $T = (O, E_1, \ldots, E_n, \rho)$, where:*

- *$O \subset \mathcal{O}$ is a finite set of objects,*
- *each $E_i \subseteq O \times O \times O$ is a set of triples, and*
- *$\rho : O \to \mathcal{D}$ is a function that assigns a data value to each object.*

Often we have just a single ternary relation E in a triplestore database (e.g., in the previously seen examples of representing RDF databases), but all the languages and results we state here apply to multiple relations. The function ρ could also map $\mathcal{O}$ to tuples

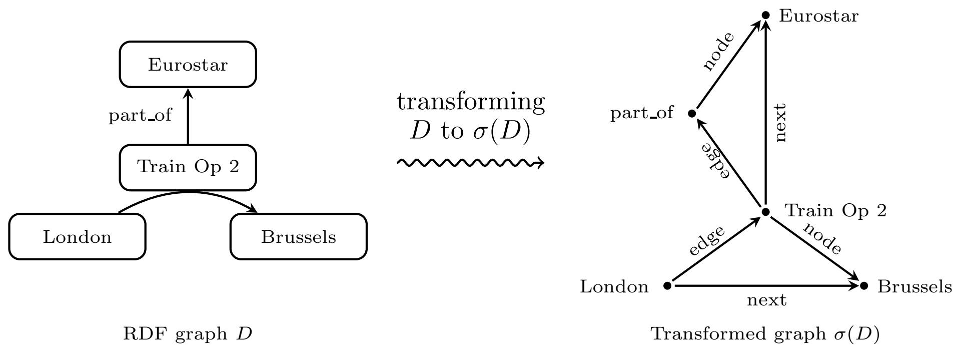


Figure 2: Transforming part of the RDF database from Figure 1 into a graph database

over $\mathcal{D}$, and all results remain true (one just uses $\mathcal{D}^k$ as the range of ρ, as in the example below). We use the function $\rho : O \to \mathcal{D}$ just to simplify notations.

Triplestores easily model RDF, and we will see later that they model graph databases. Furthermore, they can be used to model several other applications relying on semistructured data, such as e.g. social networks.

3. AN ALGEBRA FOR RDF

We saw that problems encountered while adapting graph languages to RDF are related to the inherent limitations of the graph data model for representing RDF data. Thus, one should work directly with triples. But existing languages are either based on binary relations and fall short of the power necessary for RDF querying, or are general relational languages which are not closed when it comes to querying RDF triples. Hence, we need a language that works directly on triples, is closed, and has good query evaluation properties.

We now present such a language, based on relational algebra for triples. We start with a plain version and then add recursive primitives that provide the crucial functionality for handling reachability properties.

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 valid triplestore. This clearly rules out projection. Selection and Boolean operations are fine. Cartesian product, however, would create a relation of arity six, but instead we use *joins* that only keep three positions in the result.

Triple joins. To see 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. Reachability with relation S is defined by recursively applying composition: $S \cup S \circ S \cup S \circ S \circ S \cup \ldots$. 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 tertiary 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 tertiary relations R and R', *join* operations are of the form

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

where

- $i, j, k \in \{1, 1', 2, 2', 3, 3'\}$,
- θ is a set of equalities and inequalities between elements in $\{1, 1', 2, 2', 3, 3'\} \cup \mathcal{O}$,
- η is a set of equalities and inequalities between elements in $\{\rho(1), \rho(1'), \rho(2), \rho(2'), \rho(3), \rho(3')\} \cup \mathcal{D}$.

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 object, is in θ, then $o_l = o$, and likewise for inequalities;
- each condition from η holds; that is, if $\rho(l) = \rho(m)$ is in η, then $\rho(o_l) = \rho(o_m)$, and if $\rho(l) = d$, where d is a data value, is in η, then $\rho(o_l) = d$, and likewise for inequalities.

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, i.e., the result of each expression is a triplestore.

- Every relation name in a triplestore is a TriAL expression.
- If e is a TriAL expression, θ a set of equalities and inequalities over $\{1,2,3\} \cup \mathcal{O}$, and η is a set of equalities and inequalities over $\{\rho(1), \rho(2), \rho(3)\} \cup \mathcal{D}$, then $\sigma_{\theta,\eta}(e)$ is a TriAL expression.
- If e_1, e_2 are TriAL expressions, then the following are TriAL expressions:
  - $e_1 \cup e_2$;
  - $e_1 - e_2$;
  - $e_1 \bowtie_{\theta,\eta}^{i,j,k} e_2$, where i, j, k, θ, η 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 both θ and η.

Given a triplestore database T, we write $e(T)$ for the result of expression e on T.

Note that $e(T)$ is again a triplestore, and thus TriAL defines closed operations on triplestores. This is important, for instance, when we require RDF queries to produce RDF graphs as their result (instead of arbitrary tuples of objects), as it is done in SPARQL via the CONSTRUCT operator [34].

Example 2. To get some intuition about the Triple Algebra consider the following TriAL expression:

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

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 e outputs the triple $(x_1, x_{3'}, x_3)$. E.g., in the triplestore of Fig. 1, (`London`, `Train Op 2`, `Brussels`) is joined with (`Train Op 2`, `part_of`, `Eurostar`), producing (`London`, `Eurostar`, `Brussels`); the full result is

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

Thus, e computes travel information for pairs of European cities together with companies one can use. It 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 $e' = e \cup (e \bowtie_{2=1'}^{1,3',3} E)$.

Definable operations: intersection and complement. As usual, the intersection operation can be defined as $e_1 \cap e_2 = e_1 \bowtie_{1=1',2=2',3=3'}^{1,2,3} e_2$. Note that using join and union, we can define the set U of all triples (o_1, o_2, o_3) so that each o_i occurs in our triplestore database T. For instance, to collect all such triples so that o_1 occurs in the first position of R, and o_2, o_3 occur in the 2nd and 3rd position of R' respectively, we would use the expression $(R \bowtie^{1,2',3} R') \bowtie^{1,2,3'} R'$. Taking the union of all such expressions, gives us the relation U.

Using such U, we can define e^c, the complement of e with respect to the active domain, as $U - e$. In what follows, we regularly use intersection and complement in our examples.

Adding Recursion. One problem with Example 2 above is that it does not include triples ($\texttt{city}_1$,`service`,$\texttt{city}_2$) so that relation R contains a triple ($\texttt{city}_1$,$\texttt{service}_0$,$\texttt{city}_2$), and there is a chain, of some length, indicating that $\texttt{service}_0$ is a part of `service`. The second expression in Example 2 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 $\bowtie_{\theta,\eta}^{i,j,k}$ over an expression e, denoted as $(e \bowtie_{\theta,\eta}^{i,j,k})^*$ for right, and $(\bowtie_{\theta,\eta}^{i,j,k} e)^*$ for left. These are defined as

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

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 3. Consider a triplestore database $T = (O, E)$, with $E = \{(a,b,c), (c,d,e), (d,e,f)\}$. The function ρ is not relevant for this example. The expression

$$e_1 = (E \underset{3=1'}{\overset{1,2,2'}{\bowtie}})^*$$

computes $e_1(T) = E \cup \{(a,b,d), (a,b,e)\}$, while

$$e_2 = (\underset{3=1'}{\overset{1,2,2'}{\bowtie}} E)^*$$

computes $e_2(T) = E \cup \{(a,b,d)\}$.

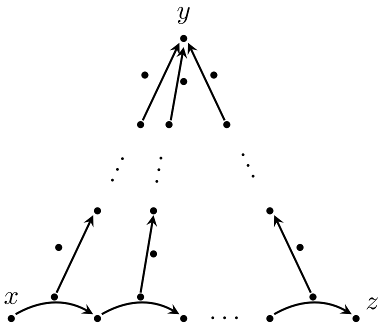
Now we present several examples of queries one can ask using the Triple Algebra.

Example 4. We refer now to reachability queries $\mathsf{Reach}_{\to}$ and $\mathsf{Reach}_{\nearrow}$ from the introduction. It can easily be checked that these are defined by

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

respectively.

Next consider the query from Theorem 1. Graphically, it can be represented as follows:



That is, we are looking for pairs of cities such that one can travel from one to the other using services operated by the same company. This query is expressed by

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

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.

4. A DECLARATIVE LANGUAGE

Triple Algebra and its recursive versions are *procedural* languages. In databases, we are used to dealing with declarative languages. The most common one for expressing queries that need recursion is Datalog. It is one of the most studied database query languages, and it has reappeared recently in numerous applications. One instance of this is its well documented success in Web information extraction [19] and there are numerous others. So it seems natural to look for Datalog fragments to capture TriAL and its recursive version.

Since Datalog works over relational vocabularies, we need to explain how to represent triplestores T. The schema of these representations consists of a ternary relation symbol $E(\cdot,\cdot,\cdot)$ for each triplestore name in T, plus a binary relation symbol $\sim(\cdot,\cdot)$. Each triplestore database T can be represented as an instance I_T of this schema in the standard way: the interpretation of each relation name E in this instance corresponds to the triples in the triplestore E in T, and the interpretation of $\sim$ contains all pairs (x, y) of objects such that $\rho(x) = \rho(y)$, i.e. x and y have the same data value. If the values of ρ are tuples, we just use $\sim_i$ relations testing that the ith components of tuples are the same, for each i; this does not affect the results here at all.

We start with a Datalog fragment capturing TriAL. A TripleDatalog rule is of the form

$$\begin{aligned} S(\bar{x}) \;\leftarrow\; & S_1(\bar{x}_1), S_2(\bar{x}_2), \\ & \sim(y_1, z_1), \ldots, \sim(y_n, z_n), u_1 = v_1, \ldots, u_m = v_m \quad (1) \end{aligned}$$

where

1. S, S_1 and S_2 are (not necessarily distinct) predicate symbols of arity at most 3;
2. all variables in $\bar{x}$ and each of y_i, z_i and u_j, v_j are contained in $\bar{x}_1$ or $\bar{x}_2$.

A $\mathsf{TripleDatalog}^{\neg}$ rule is like the rule (1) 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 *nonrecursive* 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 [1] for the precise definition). We are now ready to present the first capturing result.

Proposition 2. TriAL *is equivalent to nonrecursive* $\mathsf{TripleDatalog}^{\neg}$ *programs.*

We next turn to the expressive power of recursive Triple Algebra TriAL^*. To capture it, we of course add recursion to Datalog rules, and impose a restriction that was previously used in [12]. A $\mathsf{ReachTripleDatalog}^{\neg}$ *program* is a $\mathsf{TripleDatalog}^{\neg}$ program in which each recursive predicate S is the head of exactly two rules of the form:

$$\begin{aligned} S(\bar{x}) \;&\leftarrow\; R(\bar{x}) \\ S(\bar{x}) \;&\leftarrow\; S(\bar{x}_1), R(\bar{x}_2), V(y_1, z_1), \ldots, V(y_k, z_k) \end{aligned}$$

where each $V(y_i, z_i)$ is one of the following: $y_i = z_i$, or $y_i \neq z_i$, or $\sim(y_i, z_i)$, or $\neg\sim(y_i, z_i)$, and R is a nonrecursive predicate of arity at most 3. These rules essentially mimic the standard reachability rules (for binary relation) in Datalog, and in addition one can impose equality and inequality constraints, as well as data 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 [1]. A similarly defined syntactic class, but over graph databases, rather than triplestores, was shown to capture the expressive power of FO with the transitive closure operator [12]. In our case, we have a capturing result for TriAL^*.

Theorem 2. *The expressive power of* TriAL^* *and* $\mathsf{ReachTripleDatalog}^\neg$ *programs is the same.*

Next we give an example of a simple datalog program computing the query from Theorem 1.

Example 5. The following $\mathsf{ReachTripleDatalog}^\neg$ program is equivalent to query Q from Theorem 1. Note that the answer is computed in the predicate Ans.

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

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

5. QUERY EVALUATION

In this section we analyze two versions of the query evaluation problems related to Triple Algebra. The *query evaluation* problem is to check if a given tuple is in the result of a query (as is standard in the study of complexity of database queries, especially when one wants to know which complexity classes they belong to). The query *computation* problem is to produce the output $e(T)$ for an expression e and a triplestore database T. We start with query evaluation.

Problem:	QUERYEVALUATION
Input:	A TriAL^* expression e, a triplestore T and a tuple (x_1, x_2, x_3) of objects.
Question:	Is $(x_1, x_2, x_3) \in e(T)$?

Many graph query languages (e.g., RPQs) have PTIME upper bounds for this problem, and the data complexity (i.e., when e is assumed to be fixed) is generally in NLOGSPACE (which cannot be improved, since the simplest reachability problem over graphs is already NLOGSPACE-hard). We now show that the same upper bounds hold for our algebra, even with recursion.

Proposition 3. *The problem* QUERYEVALUATION *is* PTIME*-complete, and in* NLOGSPACE *if the algebra expression e is fixed.*

Tractable evaluation (even with respect to combined complexity) is practically a must when dealing with very large and dynamic semi-structured databases. However, in order to make a case for the practical applicability of our algebra, we need to give more precise bounds for query evaluation, rather than describe complexity classes the problem belongs to. We now show that TriAL^* expressions can be evaluated in what is essentially cubic time with respect to the data. Thus, in the rest of the section we focus on the problem of actually computing the whole relation $e(T)$:

Problem:	QUERYCOMPUTATION
Input:	A TriAL^* expression e and a triplestore database T.
Output:	The relation $e(T)$

We now analyze the complexity of QUERYCOMPUTATION. Following an assumption frequently made in papers on graph database query evaluation (in particular, graph pattern matching algorithms) as well as bounded variable relational languages (cf. [16, 15, 20]), we consider an *array representation* for triplestores. That is, when representing a triplestore $T = (O, E_1, \dots, E_m, \rho)$ with $O = \{o_1, \dots, o_n\}$, we assume that each relation E_l is given by a three-dimensional $n \times n \times n$ matrix, so that the ijkth entry is set to 1 iff (o_i, o_j, o_k) is in E_l. Alternatively we can have a single matrix, where entries include sets of indexes of relations E_l that triples belong to. Furthermore we have a one-dimensional array of size n whose ith entry contains $\rho(o_i)$. Using this representation we obtain the following bounds.

Theorem 3. *The problem* QUERYCOMPUTATION *can be solved in time*

- $O(|e| \cdot |T|^2)$ *for* TriAL *expressions,*
- $O(|e| \cdot |T|^3)$ *for* TriAL^* *expressions.*

Note that this immediately gives the PTIME upper bound for Proposition 3.

One can examine the proofs of Proposition 2 and Theorem 2 and see that translations from Datalog into algebra are linear-time. Thus, we have the same bound for the query computation problem, when we evaluate a Datalog program Π in place of an algebra expression.

Corollary 1. *The problem* QUERYCOMPUTATION *for Datalog programs* Π *can be solved in time*

- $O(|\Pi| \cdot |T|^2)$ *for* $\mathsf{TripleDatalog}^\neg$ *programs,*
- $O(|\Pi| \cdot |T|^3)$ *for* $\mathsf{ReachTripleDatalog}^\neg$ *programs.*

Lower-complexity fragments. Even though we have acceptable combined complexity of query computation, if the size of T is very large, one may prefer to lower the it even further. We now look at fragments of TriAL^* for which this is possible.

In algorithms from Theorem 3, the main difficulty arises from the presence of inequalities in join conditions. A natural restriction then is to look at a fragment $\mathsf{TriAL}^=$ of TriAL in which all conditions θ and η used in joins can only use equalities. This fragment allows us to lower the $|T|^2$ complexity, by replacing one of the $|T|$ factors by $|O|$, the number of distinct objects.

Proposition 4. *The* QUERYCOMPUTATION *problem for* $\mathsf{TriAL}^=$ *expressions can be solved in time* $O(|e| \cdot |O| \cdot |T|)$.

To pose navigational queries, one needs the recursive algebra, so the question is whether similar bounds can be obtained for meaningful fragments of TriAL^*. Using the ideas from the proof of Theorem 3 we immediately get an $O(|e| \cdot |O| \cdot |T|^2)$ upper bound for $\mathsf{TriAL}^=$ with recursion. However, we can improve this result for the fragment $\mathsf{reachTA}^=$ that extends $\mathsf{TriAL}^=$ with essentially *reachability* properties, such as those used in RPQs and similar query languages for graph databases.

To define it, we restrict the star operator to mimic the following graph database reachability queries:

- the query "reachable by an arbitrary path", expressed by $(R \bowtie^{1,2,3'}_{3=1'})^*$; and
- the query "reachable by a path labeled with the same element", expressed by $(R \bowtie^{1,2,3'}_{3=1',2=2'})^*$.

These are the only applications of the Kleene star permitted in $\mathsf{reachTA}^=$. For this fragment, we have the same lower complexity bound.

Proposition 5. *The problem* QUERYCOMPUTATION *for* $\mathsf{reachTA}^=$ *can be solved in time* $O(|e| \cdot |O| \cdot |T|)$.

6. TRIPLE ALGEBRA AND RELATIONAL LANGUAGES

In this section we compare the expressive power of our algebras with relational languages. As usual, we say that a language $\mathcal{L}_1$ is contained in 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 $\mathcal{L}_1$ is strictly contained in $\mathcal{L}_2$. The languages are equivalent if each is contained in the other. They are incomparable if none is contained in the other.

6.1 Triple Algebra as a Relational language

To compare TriAL with relational languages, we use exactly the same relational representation of triplestores as we did when we found Datalog fragments capturing TriAL and TriAL^*. That is, we compare the expressive power of TriAL with that of First–Order Logic (FO) over vocabulary $\langle E_1, \ldots, E_n, \sim\rangle$. Given an FO formula $\varphi(\bar{x})$ and an instance I over such a vocabulary, we write $\varphi(I)$ for the result of evaluation of $\varphi(\bar{x})$ over I, i.e., the set of all tuples of objects $\bar{a}$ of size $|\bar{x}|$ such that $I \models \varphi(\bar{a})$.

Since TriAL is a restriction of relational algebra, of course it is contained in FO. We do a more detailed analysis based on the number of variables. Recall that FO^k stands for FO restricted to k variables only. To give an intuition why such restrictions are relevant for us, consider, for instance, the join operation $e = E \bowtie^{1,3',3}_{2=2'} E$. It can be expressed by the following FO^6 formula: $\varphi(x_1, x_{3'}, x_3) = \exists x_2 \exists x_{1'} \exists x_{2'} \big(E(x_1, x_2, x_3) \wedge E(x_{1'}, x_{2'}, x_{3'}) \wedge x_2 = x_{2'}\big)$. This suggests that we can simulate joins using only six variables, and this extends rather easily to the whole algebra. One can furthermore show that the containment is proper in this case.

What about fragments of FO using fewer variables? Clearly we cannot go below three variables. It is not difficult to show that TriAL simulates FO^3, but the relationship with the 4 and 5 variable formalisms appears much more intricate, and its study requires more involved techniques. We can show the following.

Theorem 4.

- TriAL *is strictly contained in* FO^6.
- FO^3 *is strictly contained in* TriAL.
- TriAL *is incomparable with* FO^4 *and* FO^5.

The containment of FO^3 in TriAL is proved by induction, and we use pebble games to show that such containment is proper. For the last, more involved part of the theorem, we first show that TriAL is not contained in FO^5. Notice that the expression e given by

$$U \underset{\theta}{\overset{1,2,3}{\bowtie}} U, \text{ with } \theta = \{i \neq j \mid i, j \in \{1, 1', 2, 2', 3, 3'\}, i < j\},$$

is such that $e(T)$ is not empty if and only if T has six different objects (recall that U is the set of all triples (o_1, o_2, o_3) so that each o_i occurs in a triple in T). It then follows that TriAL is not contained in FO^5 (nor FO^4), cf. [26]. To show that FO^4 is not contained in TriAL, we devise a game that characterizes expressibility of TriAL, and use this game to show that TriAL cannot express the following FO^4 query $\varphi(x, y, z)$:

$$\exists w \big(\psi(x, y, w) \wedge \psi(x, w, z) \wedge \psi(w, y, z) \wedge \psi(x, y, z)\big),$$

where

$$\psi(x, y, z) = \exists w \big(E(x, w, y) \wedge E(y, w, z) \wedge E(z, w, x)\big).$$

The above result also shows that TriAL cannot express all conjunctive queries, since in particular the query $\varphi(x, y, z)$ is a conjunctive query. This is of course expected; the intuition is that TriAL queries have limited memory and thus cannot express queries such as the existence of a k-clique, for large values of k.

Expressivity of $\mathsf{TriAL}^=$. The TriAL queries we used to separate it from FO^5 or FO^4 make use of inequalities in the join conditions. Thus, it is natural to ask what happens when we restrict our attention to $\mathsf{TriAL}^=$, the fragment that disallows inequalities in selections and joins. We saw in Section 5 that this fragment appears to be more manageable in terms of query answering. This suggests that fewer variables may be enough, as the number of variables is often indicative of the complexity of query evaluation [23, 36]. This is indeed the case.

Theorem 5.

- FO^3 *is strictly contained in* $\mathsf{TriAL}^=$.
- $\mathsf{TriAL}^=$ *is strictly contained in* FO^4.

Next, we turn to the expressive power of TriAL*. Since the Kleene star essentially defines the transitive closure of join operators, it seems natural for our study to compare TriAL* with Transitive Closure Logic, or TrCl.

Formally, TrCl is defined as an extension of FO with the following operator. If $\varphi(\bar{x}, \bar{y}, \bar{z})$ is a formula, where $|\bar{x}| = |\bar{y}| = n$, and $\bar{u}, \bar{v}$ are tuples of variables of the same length n, then $[\mathbf{trcl}_{\bar{x},\bar{y}}\varphi(\bar{x}, \bar{y}, \bar{z})](\bar{u}, \bar{v})$ is a formula whose free variables are those in $\bar{z}$, $\bar{u}$ and $\bar{v}$. For an instance I and instantiations $\bar{a}$, $\bar{b}$ and $\bar{c}$ for $\bar{u}$, $\bar{v}$ and $\bar{c}$, construct a graph G whose nodes are elements of I^n and edges contain pairs $(\bar{u}_1, \bar{u}_2)$ so that $\varphi(\bar{u}_1, \bar{u}_2, \bar{c})$ holds in I. Then $I \models [\mathbf{trcl}_{\bar{x},\bar{y}}\varphi(\bar{x}, \bar{y}, \bar{c})](\bar{a}, \bar{b})$ iff $(\bar{a}, \bar{b})$ is in the transitive closure of this graph G.

It is fairly easy to show that TriAL* is contained in TrCl; the question is whether one can find analogs of Theorem 4 for fragments of TrCl using a limited number of variables. We denote by TrCl^k the restriction of TrCl to k variables. Note that constructs of form $[\mathbf{trcl}_{\bar{x},\bar{y}}\varphi(\bar{x}, \bar{y}, \bar{z})](\bar{t}_1, \bar{t}_2)$ can be defined using $|\bar{t}_1| + |\bar{t}_2| + |\bar{z}|$ variables, by reusing $\bar{t}_1$ and $\bar{t}_2$ in φ.

Then we can show that the relationship between TriAL* and TrCl mimics the results of Theorem 4 for the case of TriAL and FO.

Theorem 6.

- TriAL* *is strictly contained in* TrCl^6.
- TrCl^3 *is strictly contained in* TriAL*.
- TriAL* *is incomparable with* TrCl^4 *and* TrCl^5.

6.2 Triple Algebra as a Graph Language

The goal of this section is to demonstrate the usefulness of TriAL* in the context of graph databases. In particular we show how to use TriAL* for querying graph databases, both with and without data values, and compare it in terms of expressiveness with several well established graph database query languages.

6.2.1 Navigational graph query languages

We compare TriAL* with a number of established formalisms for graph databases such as NREs, RPQs and CRPQs. As our yardstick language for comparison we use a recently proposed version of XPath, adapted for graph querying [27]. Its navigational fragment, used now, is essentially Propositional Dynamic Logic (PDL) [21] with negation on paths; below we also expand it with data tests when we deal with graphs whose nodes hold data values. These languages are designed to query the topology of a graph database and specify various reachability patterns between nodes. As such, they are naturally equipped with the star operator and to make our comparison fair we will compare them with TriAL*.

The navigational language used now is called GXPath; its formulae are split into node tests, returning sets of nodes and path expressions, returning sets of pairs of nodes. Node tests are given by the following grammar:

$$\varphi, \psi := \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle\alpha\rangle$$

where α is a path expression.

The path formulae of GXPath are given below. Here a ranges over the labeling alphabet Σ.

$$\alpha, \beta := \varepsilon \mid a \mid a^- \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha} \mid \alpha^*.$$

The semantics is standard, and follows the usual semantics of PDL or XPath languages. Given a graph $G = (V, E)$, $\top$ returns V, and $\langle\alpha\rangle$ returns $v \in V$ so that (v, v') is in the semantics of α for some $v' \in V$. The semantics of Boolean operators is standard. For path formulae, ε returns $\{(v, v) \mid v \in V\}$, a returns $\{(v, v') \mid (v, a, v') \in E\}$ and a^- returns $\{(v', v) \mid (v, a, v') \in E\}$. Expressions $\alpha \cdot \beta, \alpha \cup \beta, \overline{\alpha}$, and α^* denote relation composition, union, complement, and transitive closure. Finally $[\varphi]$ denotes the set of pairs (v, v) so that v is in the semantics of φ.

Since TriAL* is designed to query triplestores, we need to explain how to compare its power with that of graph query languages. Given a graph database $G = (V, E)$ over the alphabet Σ, we define a triplestore $T_G = (O, E)$, with $O = V \cup \Sigma$. Note that for now we deal with navigation; later we shall also look at data values.

To compare TriAL* with binary graph queries in a graph query language $\mathcal{L}$, we turn TriAL* ternary queries Q into binary by applying the $\pi_{1,3}(Q)$, i.e., keeping (s, o) from every triple (s, p, o) returned by Q. Under these conventions, we say that a graph query language $\mathcal{L}$ is contained in TriAL* if for every binary query $\alpha \in \mathcal{L}$ there is a TriAL* expression e_α so that $\pi_{1,3}(e_\alpha)$ and α are equivalent, and likewise, TriAL* is contained in a graph query language $\mathcal{L}$ if for every expression e in TriAL* there is a binary query $\alpha_e \in \mathcal{L}$ that is equivalent to $\pi_{1,3}(e)$. The notions of being strictly contained and incomparable extend in the same way.

Alternatively, one can do comparisons using triplestores represented as graph databases, as in Proposition 1. Since here we study the ability of TriAL* to serve as a graph query language, the comparison explained above looks more natural, but in fact all the results remain true even if we do the comparison over triplestores represented as graph databases, as described in Section 2.

We now show that all GXPath queries can be defined in TriAL*, but that there are certain properties that TriAL* can define that lie beyond the reach of GXPath.

Theorem 7. *GXPath is strictly contained in* TriAL*.

We prove this by using the equivalence of GXPath with the 3-variable fragment of reachability logic FO* [35], shown in [27].

Note that this also implies a strict containment of languages presented in [17, 18] in TriAL^*, since it is easy to show that they are subsumed by GXPath.

To compare TriAL^* with common graph languages such as NREs and RPQs we observe that NREs can be thought of as path expressions of GXPath that do not use complement and where nesting is replaced with $[\langle\alpha\rangle]$. RPQs do not even have nesting. Thus:

Corollary 2.

- *NREs are strictly contained in* TriAL^*.
- *RPQs are strictly contained in* TriAL^*.

It is common in graph databases to consider queries that are closed under conjunction and existential quantification, such as CRPQs [13, 37], C2RPQs [10] and CNREs [9]. The latter are expressions $\varphi(\bar{x}) = \exists \bar{y} \bigwedge_{i=1}^{n} (x_i \xrightarrow{e_i} y_i)$, where all variables x_i, y_i come from $\bar{x}, \bar{y}$ and each e_i is a NRE. The semantics extends that of NREs, with each $x_i \xrightarrow{e_i} y_i$ interpreted as the existence of a path between them that is denoted by e_i. We compare TriAL^* with these queries, and also with *unions* of CNREs that use bounded number of variables.

Theorem 8.

- *CNREs and* TriAL^* *are incomparable in terms of expressive power.*
- *Unions of CNREs that use only three variables are strictly contained in* TriAL^*.

By observing that the expressions separating CNREs from TriAL^* are CRPQs, and that CNREs are more expressive than CRPQs and C2RPQS [8] we obtain:

Corollary 3.

- *CRPQs and* TriAL^* *are incomparable in terms of expressive power.*
- *Unions of C2RPQs and CRPQs that use only three variables are strictly contained in* TriAL^*.

There are further extensions, such as *extended* CRPQs, where paths witnessing RPQs can be named and compared for relationships between them, defined as regular or even rational relations [6, 7]. We leave the comparison with these languages as future work.

6.2.2 Query languages for graphs with data

Until now we have compared our algebra with purely navigational formalisms. Triple stores do have data values, however, and can thus model any graph database. That is, for any graph database $G = (V, E, \rho)$ we can define a triplestore $T_G = (O, E, \rho)$ with $O = V \cup \Sigma$. Note that nodes corresponding to labels have no data values assigned in our model. This is not an obstacle and can in fact be used to model graph databases that have data values on both the nodes and the edges.

We provide a comparison to an extension of GXPath with data value comparisons. The language, denoted by $\mathsf{GXPath}(\sim)$, presented first in [27], is given by the following grammars for node and path formulae:

$$\varphi, \psi := \top | \langle \alpha = \beta \rangle \mid \langle \alpha \neq \beta \rangle \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha \rangle$$

$$\alpha, \beta := \varepsilon \mid a \mid a^- \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha} \mid \alpha^* \mid \alpha_= \mid \alpha_{\neq}.$$

The semantics of additional expressions is as follows: α_θ returns those pairs (v, v') returned by α for which $\rho(v)\ \theta\ \rho(v')$, for $\theta \in \{=, \neq\}$, and $\langle \alpha\ \theta\ \beta \rangle$ returns nodes v such that there are pairs (v, v_α) and (v, v_β) returned by α and β and $\rho(v_\alpha)\ \theta\ \rho(v_\beta)$. The former addition corresponds to the notion of regular expressions with equality [28], and the latter to standard XPath data-value comparisons.

To compare $\mathsf{GXPath}(\sim)$ with TriAL^*, we use the same convention as for data value-free languages. Connections of $\mathsf{GXPath}(\sim)$ with a 3-variable reachability logic and the proof of Theorem 4 show:

Corollary 4. *$\mathsf{GXPath}(\sim)$ is strictly contained in* TriAL^*.

This also implies that TriAL^* subsumes an extension of RPQs based on regular expressions with equality [28], which can test for (in)equality of data values at the beginning and the end of paths.

Another formalism proposed for querying graph databases with data values is that of *register automata* [25]. In general, these work over data words, i.e., words over both a finite alphabet and an infinite set of data values. RPQs defined by register automata find pairs of nodes connected by a path accepted by such automata. We refer to [28, 25] for precise definitions, and state the comparison result below.

Proposition 6. TriAL^* *is incomparable in terms of expressive power with register automata.*

This follows since register automata can define properties not expressible with six variables, but on the other hand are not closed under complement.

7. CONCLUSIONS AND FUTURE WORK

While graph database query mechanisms have been promoted as a useful tool for querying RDF data, most of these approaches view RDF as a graph database. Although inherently similar, the two models do have significant differences. We showed that some very natural navigational queries for RDF cannot be expressed with graph-based navigational mechanisms. The solution is then to use proper triple-based models and languages.

We introduced such a model, that combines the usual idea of triplestores used in many RDF implementations, with that of graphs with data, and proposed an algebra for that model. It comes in two flavors, a non-recursive algebra TriAL and a recursive one TriAL^*. We also provided Datalog-based declarative languages capturing these. We studied the query evaluation problem, as well as the expressivity of the languages, comparing them with both relational and graph query languages.

There are several future directions we would like to pursue. One relates to understanding connections with another way restriction guaranteeing closure, namely using semi-joins. Although some of the properties crucial for our goals cannot be expressed solely with semi-joins, such restrictions are closely related to the guarded fragment of FO [29], which enjoys better properties than the full FO. Another theoretical question that arises from our investigation is studying connections between languages for tuples of arbitrary arity, not just triples.

On the more practical side, we want to provide a deeper insight into the connection of our languages and nSPARQL, which seems to be the current choice for navigational RDF queries. For instance, we would like to see whether TriAL^* functionalities can be included into SPARQL, resulting in a language provably more expressive than nSPARQL, that provides recursive functionalities needed to compute most navigational queries required in RDF, including property paths. Another direction is to see how possible implementations of TriAL^* stack up against currently used systems. In this respect we would like to test if commercial RDBMSs can scalably implement the type of recursion we require, or whether augmenting one of the existing open-source triplestore systems will result in a more efficient evaluation when recursion is added.

Acknowledgments Work partly supported by EPSRC grants G049165 and J015377. We thank anonymous referees for their helpful suggestions.

8. References

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

[2] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1), 2008.

[3] R. Angles. A comparison of current graph database models. In *ICDE Workshops*, pages 171–177, 2012.

[4] K. Anyanwu and A. Sheth. ρ-Queries: Enabling querying for semantic associations on the Semantic Web. In *WWW'03*, pages 690–699.

[5] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *PODS*, pages 305–316, 2011.

[6] P. Barceló, L. Libkin, A.W. Lin, and P. Wood. Expressive languages for path queries over graph-structured data. *ACM TODS* 38(4) (2012).

[7] P. Barceló, D. Figueira, and L. Libkin. Graph logics with rational relations and the generalized intersection problem. In *LICS'12*, pages 115–124.

[8] P. Barceló, J. Pérez, and J. L. Reutter. Relative expressiveness of nested regular expressions. In *AMW'12*, pages 180–195.

[9] P. Barceló, J. Pérez, and J. L. Reutter. Schema mappings and data exchange for graph databases. In *ICDT'13.*

[10] D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR'2000*, pages 176–185.

[11] D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. Rewriting of regular expressions and regular path queries. *JCSS*, 64(3):443–465, 2002.

[12] M. Consens, A. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS'90*, pages 404–416.

[13] I. Cruz, A.O. Mendelzon, and P. Wood. A graphical query language supporting recursion. In *SIGMOD'87*, pages 323–330.

[14] P. Cudré-Mauroux and S. Elnikety. Graph data management systems for new application domains. *PVLDB*, 4(12):1510–1511, 2011.

[15] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.

[16] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Graph pattern matching: from intractable to polynomial time. *PVLDB*, 3(1):264–275, 2010.

[17] G. Fletcher et al. Relative expressive power of navigational querying on graphs. *ICDT 2011*, 197-207

[18] G. Fletcher et al. The impact of transitive closure on the boolean expressiveness of navigational query languages on graphs. *FoIKS 2012*, 124-143

[19] G. Gottlob and C. Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.

[20] G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: a deductive query language with linear time model checking. *ACM TOCL*, 3(1):42–79, 2002.

[21] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[22] S. Harris et al. *SPARQL 1.1 Query Language.* `http://www.w3.org/TR/sparql11-query`.

[23] N. Immerman, D. Kozen. Definability with Bounded Number of Bound Variables. *IANDC*, 83(2):121-139 (1989).

[24] The Apache Jena Manual. `http://jena.apache.org`.

[25] M. Kaminski and N. Francez. Finite memory automata. *TCS*, 134(2):329–363, 1994.

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

[27] L. Libkin, W. Martens, and D. Vrgoč. Querying graph databases with XPath. In *ICDT*, 2013.

[28] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. In *ICDT'12*, pages 74–85.

[29] D. Leinders, M. Marx, J. Tyszkiewicz and J. Van den Bussche. The semijoin algebra and the guarded fragment. *Logic, Language and Information*, 14(3), 331–343, 2009.

[30] K. Losemann, W. Martens. The complexity of evaluating path expressions in SPARQL. In *PODS'12*, pages 101–112.

[31] The Neo4j Manual. `http://docs.neo4j.org`.

[32] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.

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

[34] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation 15 January 2008, `http://www.w3.org/TR/rdf-sparql-query/`.

[35] B. ten Cate. The expressivity of XPath with transitive closure. In *PODS*, pages 328–337, 2006.

[36] M. Vardi. On the complexity of bounded-variable queries. In *PODS'95*, pages 266–276.

[37] P. Wood. Query languages for graph databases. *Sigmod Record*, 41(1):50–60, 2012.