A

TriAL: A navigational algebra for RDF triplestores

Navigational queries over RDF data are viewed as one of the main applications of graph query languages, and yet the standard model of graph databases – essentially labelled 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 an implementation of TriAL* on top of a relational query engine, and show its usefulness by running a wide array of navigational queries over real world RDF data, while at the same time testing how our implementation compares to existing RDF systems.

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

Additional Key Words and Phrases: 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 [Angles and Gutierrez 2008; Angles 2012; Cudré-Mauroux and Elnikety 2011; Wood 2012] 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 [Angles and Gutierrez 2008; Wood 2012] that dates back to [Consens and Mendelzon 1990; Cruz et al. 1987], 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.

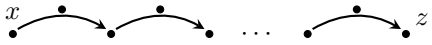
The model of RDF data is very similar, yet slightly different. The basic concept is a *triple* (s, p, o), that consists of the subject s, the predicate p, and the object o, drawn from a domain of uniform resource identifiers (URI's). Thus, the middle element need not come from a finite alphabet, and 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

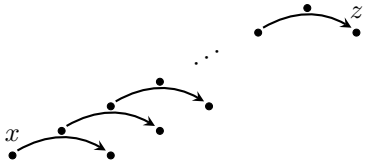
query languages have, as their basic building block, *regular path queries*, or RPQs [Cruz et al. 1987], that find nodes reachable by a path whose label belongs to a regular language.

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 questions, we need to understand which navigational facilities are available for RDF data. First, one needs to consider *property paths*, a feature added to SPARQL [Harris and Seaborne 2013], the standard query language for RDF data, in order to allow navigational queries. However, one can easily see that property paths are nothing more than regular path queries in disguise [Kostylev et al. 2015] and thus do not account for patterns such as e.g. $\mathsf{Reach}_{\rightarrow}$ above, since they view RDF triples as a graph database. A similar attempt to add navigation to RDF languages was made in [Pérez et al. 2010], where nSPARQL, an extension to SPARQL using a generalisation of property paths known as *nested regular expressions* in order to express navigational queries, was introduced. Just as in the case of property paths, nested regular expressions are also evaluated using 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 [Arenas and Pérez 2011] using nested regular expressions (and therefore also property paths), nor in nSPARQL itself.

Thus, many natural 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. In order to design such a language we need to see what types of properties are important for any querying mechanism.

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, or using SPARQL's property paths. 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.

Next, move onto the comparison of TriAL 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 result 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 [Libkin et al. 2013]. We also compare it with conjunctive RPQs [Consens and Mendelzon 1990] and some of their extensions studied in [Calvanese et al. 2000; 2002].

Of course, showing that a language has nice theoretical properties does not mean it is feasible to have it implemented in practice. For this reason in Section 7 we describe a proof of concept implementation of TriAL* on top of an existing relational database system, and test how it performs over real world RDF datasets and over synthetic data. We also compare this implementation with state of the art SPARQL query engines and show that our implementation either outperforms them or is at least competitive when evaluating navigational queries, while at the same time allowing more expressive power.

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, and that has good query evaluation properties. Furthermore, TriAL* permits an efficient implementation that performs at the level of modern RDF query engines, while at the same time being capable of expressing many queries that lie outside of their scope.

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. Section 7 shows how an implementation of TriAL* performs over real world data and when compared to modern SPARQL engines. We conclude in Section 8.

2. PRELIMINARIES AND MOTIVATION

In this section we formalise the graph and RDF data model and discuss some of the limitations of using graph query languages in the RDF setting. We also introduce the notion of a triplestore, generalising both graph and RDF databases.

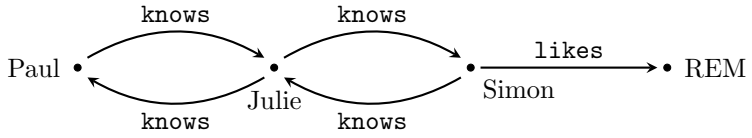


Fig. 1: A graph database showing part of a Social Network. Edge labels are on the edges, and node names next to the nodes.

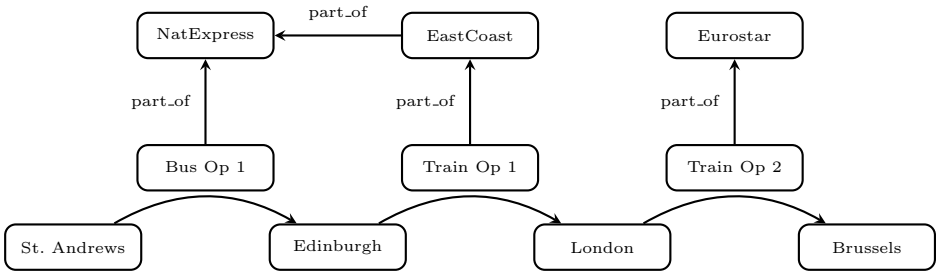


Fig. 2: RDF graph storing information about cities and transport services between them

2.1. Basic Definitions

Graph databases. Intuitively, a graph database is nothing but a finite edge labelled graph. Formally, 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. An example of a graph databases is shown in Figure 1. Here the nodes represent people or other entities in a Social Network setting, and the edges the connection between two nodes.

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 [Pérez et al. 2010], 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 2.1. *The RDF database D 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 D are (*`Edinburgh`*,* `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.*

2.2. Limitations of graph queries over RDF

Navigational properties (e.g., reachability patterns) are among the most important functionalities of RDF query languages. The current recommendation for navigational querying in RDF documents are property paths, a new addition to SPARQL, the standard query language for RDF graphs [Harris and Seaborne 2013]. However, property paths, as well as

their theoretical counterparts and extensions [Pérez et al. 2010; Anyanwu and Sheth 2003; Losemann and Martens 2012] are classes of queries inspired by classical graph query languages. As hinted in the Introduction, and as we show next, taking this approach can have certain limitations when it comes to navigational features of RDF databases.

Looking again at the database D in Figure 2, 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 it, the database from Figure 2 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:

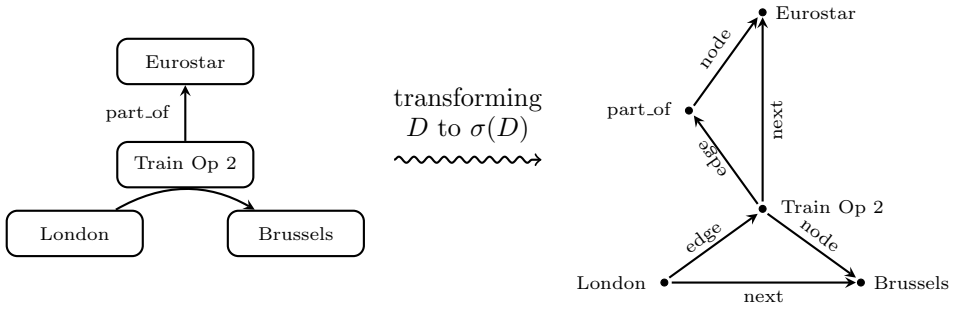
Q : *Find pairs of cities (x, y) such that one can travel from x to y using services operated by the same company.*

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.

So how do graph query languages fare when faced with a query like Q? To answer this question we will consider the class of nested regular expressions introduced in [Pérez et al. 2010], which extend SPARQL property paths with several extra functionalities. The idea behing nested regular expressions is to combine the usual reachability patterns of graph query languages with the XPath mechanism of node tests[1]. 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 [Arenas and Pérez 2011]; it is illustrated in Figure 3.

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 [Pérez et al. 2010]). At the same time, it is not sufficient for expressing simple reachability patterns like those in query Q:

[1]For a formal definition of nested regular expressions see [Pérez et al. 2010]. As the results we present do not depend on a specific syntax, but merely on the fact that the queries operate over graph databases, we omit this to keep the presentation concise.



RDF graph D Transformed graph $\sigma(D)$

Fig. 3: Transforming part of the RDF database from Figure 2 into a graph database

PROPOSITION 2.2. *The query Q is not expressible by NREs over graph transformations $\sigma(\cdot)$ of ternary relations.*

PROOF. Consider the RDF documents D_1 and D_2 consisting of the following triples:

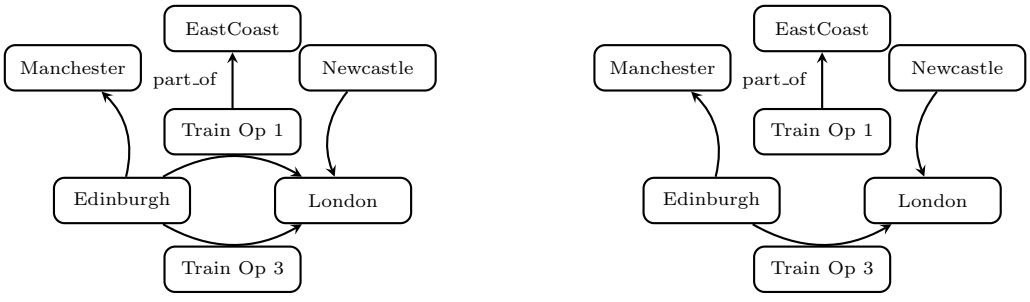
Graph D_1:

St Andrews	Bus Operator 1	Edinburgh
Edinburgh	Train Op 1	London
Edinburgh	Train Op 3	London
Edinburgh	Train Op 1	Manchester
Newcastle	Train Op 1	London
London	Train Op 2	Brussels
Bus Operator 1	part of	NatExpress
Train Op 1	part of	EastCoast
Train Op 2	part of	Eurostar
EastCoast	part of	NatExpress

Graph D_2:

St Andrews	Bus Operator 1	Edinburgh
Edinburgh	Train Op 3	London
Edinburgh	Train Op 1	Manchester
Newcastle	Train Op 1	London
London	Train Op 2	Brussels
Bus Operator 1	part of	NatExpress
Train Op 1	part of	EastCoast
Train Op 2	part of	Eurostar
EastCoast	part of	NatExpress

Essentially, graph D_1 is an extension of the RDF document D in Figure 2, while graph D_2 is the same as D_1 except that it does not contain the triple (`Edinburgh`, `Train Op 1` , `London`). The relevant parts of our databases are illustrated in the following image.



Part of RDF graph D_1 Part of RDF graph D_2

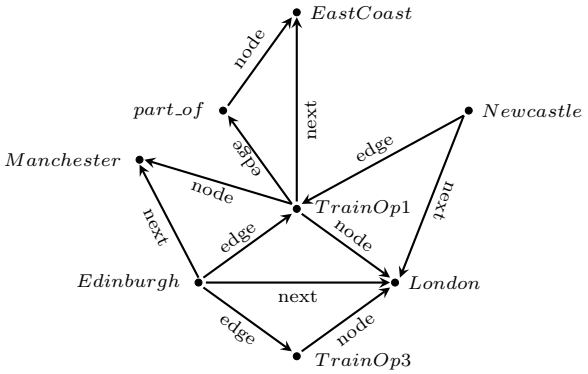


Fig. 4: Transforming part of the RDF databases D_1 and D_2

The absence of this triple has severe implications with respect to the query Q of the statement of the Proposition, since in particular the pair (`St Andrews`, `London`) belongs to the evaluation of Q over D_1, but it does not belong to the evaluation of Q over D_2.

However, it is not difficult to check that the graph translations of D_1 and D_2 are exactly the same graph database: $\sigma(D_1) = \sigma(D_2)$. We have included the relevant part of transformations $\sigma(D_1)$ and $\sigma(D_2)$ in Figure 4. It follows that Q is not expressible in nested regular expressions, since obviously the answer of all nested regular expressions is the same over $\sigma(D_1)$ and $\sigma(D_2)$ (they are the same graph). □

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 [Pérez et al. 2010], 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 2.3. *The query Q above cannot be expressed in nSPARQL.*

PROOF. The semantics of the nested regular expressions in the RDF context (in [Pérez et al. 2010]) is given as follows, assuming a triple representation of RDF documents. For next, it is the set $\{(v, v') \mid \exists z E(v, z, v')\}$, the semantics of edge is $\{(v, v') \mid \exists z E(v, v', z)\}$ and node is $\{(v, v') \mid \exists z E(z, v, v')\}$; for the rest of the operators it is the same as in the graph database case. Thus, even though stated in an RDF context, this semantics is essentially given according to the translation $\sigma(\cdot)$, in the sense that the semantics of an NRE e is the same for all RDF documents D and D' such that $\sigma(D) = \sigma(D')$ [2]. Hence the proof follows directly from Proposition 2.2 and the easy fact that Q cannot be expressed in SPARQL. □

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.

[2]The NREs defined in [Pérez et al. 2010] had additional primitives, such as next :: sp. These were added for the purpose of allowing RDFS inference with NREs, but play no role in the general expressivity of nSPARQL in our setting since we are dealing with arbitrary objects, whereas the constructs in [Pérez et al. 2010] are limited to RDFS predicates. Here we assume that primitives such as next :: [e], with e an arbitrary NRE, are not allowed. For a discussion on how the proof extends in the case when they are present see [Pérez et al. 2010]

2.3. Triplestore Databases

To introduce proper triple-based navigational languages, we first define a simple model of triplestores, which is more general than both RDF and graph databases. Let $\mathcal{O}$ be a countably infinite set of objects which can appear in our database[3].

Definition 2.4. A *triplestore database*, or just *triplestore* is a tuple $T = (O, E_1, \ldots, E_n)$, 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
— for each $o \in O$ there is $i \in \{1, \ldots, n\}$ and a triple $t \in E_i$ 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 one of the relations. This assumption will also allow us to work with the active domain of our triplestore, thus enabling us to construct an algebra that is complete in terms of first order operations.

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. Having multiple relations can be used to model richer scenarios (e.g. where we have more than one type of an edge), and can also be used to model graph databases: namely, if we have a graph database $G = (V, E)$ over an alphabet Σ, we can use a triplestore having the relation E_a for each $a \in \Sigma$ which simply stores all the triples $(v, a, v') \in E$. An alternative way of seeing graph databases as triplestores will be explored in Section 6.2.

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

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

[3]This is our analogue of URIs in RDF

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 equalities and inequalities between elements in $\{1, 1', 2, 2', 3, 3'\} \cup \mathcal{O}$.

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.

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 $\{1, 2, 3\} \cup \mathcal{O}$, then $\sigma_\theta(e)$ is a TriAL expression.
— If e_1, e_2 are TriAL expressions, then the following are TriAL expressions:
  — $e_1 \cup e_2$;
  — $e_1 - e_2$;
  — $e_1 \bowtie^{i,j,k}_\theta e_2$, 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 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 [Harris and Seaborne 2013].

Example 3.1. 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. 2, (`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 $e' = e \cup (e \bowtie^{1,3',3}_{2=1'} E)$.

Definable operations: intersection and complement. As usual, the intersection operation can be defined as $e_1 \cap e_2 = e_1 \bowtie^{1,2,3}_{1=1',2=2',3=3'} e_2$. 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 3.1 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 3.1 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 e, denoted as $(e \bowtie^{i,j,k}_{\theta})^*$ for right, and $(\bowtie^{i,j,k}_{\theta} e)^*$ for left. These are defined as

$$(e \bowtie)^* = \emptyset \cup e \cup e \bowtie e \cup (e \bowtie e) \bowtie e \cup \ldots,$$
$$(\bowtie e)^* = \emptyset \cup e \cup e \bowtie e \cup e \bowtie (e \bowtie e) \cup \ldots$$

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 3.2. 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

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

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

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

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

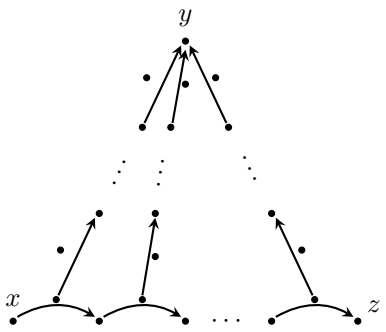
Now we present several examples of queries one can ask using the Triple Algebra.

Example 3.3. We refer now to reachability queries $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$ from the introduction. It can easily be checked that these are defined by

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

respectively.

Next consider the query from Theorem 2.2, which we denote by $\mathsf{reachTA}$. 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

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

Note that the interior join $(E \overset{1,3',3}{\underset{2=1'}{\bowtie}})^*$ 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.

Another useful application of such a nested query can be found in workflows tracking provenance of some document. Indeed, there we might be interested to find all versions of a document that contain an error, but originate from an error-free version. We might also ask if there is a path connecting those two documents where each of the versions referred to some particular document – the likely culprit for the mistake. In the image above z would represent version with an error, x a valid version it originates from, and y the document all of the versions that lead to the one with an error refer to.

Remark 3.4. Here we give some remarks about notation and implicit assumptions in the remainder of the paper.

— We will often denote conditions θ as conjunction of equalities or inequalities instead of sets. For example we will write $\theta = (1 \neq 3') \wedge (2 = 2')$ for $\theta = \{1 \neq 3', 2 = 2'\}$.
— In the proofs we will usually handle only the case of the right Kleene closure $(R \bowtie)^*$. The proofs for the left closure are completely symmetric.
— As usual in database theory, we only consider queries that are domain-independent, and therefore we loose no generality in assuming active domain semantics for FO formulas and other similar formalisms.

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 [Gottlob and Koch 2004] 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 represent triplestores as relational structures. This is done in a straightforward way; that is: we just define a schema for the triplestore T to contain a ternary relation symbol $E(\cdot,\cdot,\cdot)$ for each triplestore name in T, and a constant symbol o, for each $o \in \mathcal{O}$. Each triplestore database T can be represented as an instance I_T of this schema in the standard way: the universe of our instance is the set of all objects appearing in the relations of T, the interpretation of each relation name E in this instance corresponds to the triples in the relation E in T, and each constant is interpreted by itself. As is usually done in databases [Abiteboul et al. 1995], we deploy the *active domain semantics*, where the universe of our model is simply the set of all objects appearing in the database, and the constants are interpreted as themselves (even when they do not appear in the database – note that we have one uniform universe $\mathcal{O}$ for all objects, so this does not cause any issues). Whenever dealing with logical structures, or relational representation of a triplestore we will use this convention.

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

$$S(\overline{x}) \;\leftarrow\; S_1(\overline{x_1}), S_2(\overline{x_2}), u_1 = v_1, \ldots, u_m = v_m \tag{1}$$

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 objects in $\mathcal{O}$;
(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 TripleDatalog$^\neg$ rule is like the rule (1) but all equalities and predicates, except the head predicate S, can appear negated. A TripleDatalog$^\neg$ *program* Π is a finite set of 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 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 [Abiteboul et al. 1995] for the precise definition). We are now ready to present the first capturing result.

PROPOSITION 4.1. TriAL *is equivalent to nonrecursive* TripleDatalog$^\neg$ *programs.*

PROOF. Let us first show the containment of TriAL in non-recursive TripleDatalog$^\neg$. We show that for every expression e one can construct a non-recursive TripleDatalog$^\neg$ program Π_e such that, $e(T) = \Pi_e(I_T)$, for all triplestore databases T.

We define the translation by the following inductive construction, assuming Ans, Ans_1 and Ans_2 are special symbols that define the output of non-recursive TripleDatalog$^\neg$ programs.

— If e is just a triplestore name E, then Π_e consists of the single rule $Ans(x,y,z) \leftarrow E(x,y,z)$.

— If e is $e_1 \cup e_2$, then Π_e consists of the union of the rules of the programs Π_{e_1} and Π_{e_2}, together with the rules $Ans(\overline{x}) \leftarrow Ans_1(\overline{x})$ and $Ans(\overline{x}) \leftarrow Ans_2(\overline{x})$, where we assume that Ans_1 and Ans_2 are the predicates that define the output of Π_{e_1} and Π_{e_2}, respectively.
— If e is $e_1 - e_2$, then Π_e consists of the union of the rules of the programs Π_{e_1} and Π_{e_2}, together with the rule $Ans(\overline{x}) \leftarrow Ans_1(\overline{x}), \neg Ans_2(\overline{x})$, where we assume that Ans_1 and Ans_2 are the predicates that define the output of Π_{e_1} and Π_{e_2}, respectively.
— If e is $e_1 \bowtie_\theta^{i,j,k} e_2$, assume that θ consists of m conditions. Then Π_e consists of the union of the rules of the programs Π_{e_1} and Π_{e_2}, together with the rule

$$Ans(x_i, x_j, x_k) \leftarrow Ans_1(x_1, x_2, x_3), Ans_2(x_4, x_5, x_6), u_1(=) \neq v_1, \ldots, u_m(=) \neq v_m, \quad (2)$$

where for each p-th condition in θ of form $a = b$ or $a \neq b$, we have that $u_p = x_a$ and $v_p = x_b$ (or $u_p = o$ if a is an object o in $\mathcal{O}$, and likewise for b); and where we assume that Ans_1 and Ans_2 are the predicates that define the output of Π_{e_1} and Π_{e_2}, respectively.
— The case of selection goes along the same lines as the join case.

Clearly, this program is nonrecursive. Moreover, it is trivial to prove that this transition satisfies our desired property.

Next we show the containment of non-recursive TripleDatalog$^\neg$ in TriAL. We show that for every non-recursive TripleDatalog$^\neg$ program Π one can construct an expression e_Π such that, $e_\Pi(T) = \Pi(I_T)$, for all triplestore databases T.

We assume that Π contains a single predicate Ans that represents the answer of the query. Also, without loss of generality we can assume that no rule uses predicate E, for some triplestore name E, other than a rule of form $P(x, y, z) \leftarrow E(x, y, z)$, for a predicate P in the predicates of Π that does not appear in the head of any other rule in Π.

We need some notation. The dependence graph of Π is a directed graph whose nodes are the predicates of π, and the edges capture the dependence relation of the predicates of Π, i.e., there is an edge from predicate R to predicate S if there is a rule in Π with R in its head and S in its body. Since Π is non-recursive, its dependency graph is acyclic. We now define the TriAL expression in a recursive fashion, following its dependency graph:

— Assume that all the rules in Π that have predicate S in the head are of form

$$S(x_{a^j}, x_{b^j}, x_{c^j}) \leftarrow S_1^j(x_1^j, x_2^j, x_3^j), S_2^j(x_4^j, x_5^j, x_6^j), u_1^j(\neq) = v_1^j, \ldots, u_m^j(\neq) = v_m^j \quad (3)$$

for $1 \leq j \leq m$, and where S_1^j and S_2^j are (not necessarily distinct) predicate symbols of arity 3 and all variables in $x_{a^j}, x_{b^j}, x_{c^j}$ and each of u_k^j, v_k^j are contained in $\{x_1^j, x_2^j, x_3^j, x_4^j, x_5^j, x_6^j\}$.
Then the TriAL expression e_S is

$$\bigcup_{1 \leq j \leq m} e_{S_1^j} \bowtie_{\theta^j}^{a^j, b^j, c^j} e_{S_2^j},$$

where θ^j contains an (in)equality $a = b$ for each (in)equality $x_a = x_b$ in the rule. If either of S_1^j or S_2^j appear negated in the rule, then just replace $e_{S_1^j}$ for $(e_{S_1^j})^c$ or $(e_{S_2^j})^c$.
— The TriAL expression e_P (for predicate P in rule $P(x, y, z) \leftarrow E(x, y, z)$) is just E; if these variables appear in different order in the rule, we permute them via the selection operator σ.

It is now straightforward to verify that for every non-recursive TripleDatalog$^\neg$ program Π whose answer predicate is Ans the expression e_{Ans} is such that, $e_{Ans}(T) = \Pi(I_T)$, for all triplestore databases T. □

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 [Consens and Mendelzon 1990]. 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:

$$\begin{array}{lcl} S(\overline{x}) & \leftarrow & R(\overline{x}) \\ S(\overline{x}) & \leftarrow & S(\bar{x}_1), R(\bar{x}_2), u_1(=) \neq v_1, \ldots, u_m(=) \neq v_m, \end{array} \tag{4}$$

where each u_i and v_i is contained in $\overline{x_1} \cup \overline{x_2}$, 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 reachability rules (for binary relations) in Datalog, and in addition one can impose equality and inequality constraints.

Note that the negation in $\mathsf{ReachTripleDatalog}^\neg$ programs is *stratified*. The semantics of these programs is the standard least-fixpoint semantics [Abiteboul et al. 1995]. 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 [Consens and Mendelzon 1990]. In our case, we have a capturing result for TriAL^*.

THEOREM 4.2. *The expressive power of* TriAL^* *and* $\mathsf{ReachTripleDatalog}^\neg$ *programs is the same.*

PROOF. Let us first show the containment of TriAL^* in $\mathsf{ReachTripleDatalog}^\neg$. The proof goes along the same lines as the proof of containment of TriAL in $\mathsf{TripleDatalog}^\neg$. We have to show that for every TriAL^* expression e there is a $\mathsf{ReachTripleDatalog}^\neg$ program Π_e such that $e(T) = \Pi_e(I_T)$, for all triplestores T.

The only difference from the construction in the proof of TriAL in $\mathsf{TripleDatalog}^\neg$ is the treatment of the constructs $e = (e_1 \bowtie_\theta^{i,j,k})^*$ and $e = (\bowtie_\theta^{i,j,k} e_1)^*$. For the former construct (the other one is symmetrical), assume that $\theta = (\bigwedge_{1 \leq i \leq m} p_i(\neq) = q_i)$. We let Π_e be the union of all rules of Π_{e_1}, plus rules

$$\begin{array}{rcl} Ans(x,y,z) & \leftarrow & Ans_1(x,y,z) \\ Ans(x_i,x_j,x_k) & \leftarrow & Ans(x_1,x_2,x_3), Ans_1(x_4,x_5,x_6), x_{p_1}(\neq) = x_{q_1}, \ldots, x_{p_m}(\neq) = x_{q_m}, \end{array}$$

where Ans_1 is the answer predicate of Π_{e_1}. Notice that we have assumed for simplicity there are no comparison with constants; these can be included in our translation in a straightforward way. The proof that $e(T) = \Pi_e(I_T)$, for all triplestores T now follows easily.

The proof of containment of $\mathsf{ReachTripleDatalog}^\neg$ in TriAL^* also goes along the same lines as the proof that $\mathsf{TripleDatalog}^\neg$ is contained in TriAL. The only difference is when creating expression e_S, for some recursive predicate S. From the properties of $\mathsf{ReachTripleDatalog}^\neg$ programs, we know S is the head of exactly two rules of form

$$\begin{array}{rcl} S(\overline{x}) & \leftarrow & R(\overline{x}) \\ S(x_a,x_b,x_c) & \leftarrow & S(x_1,x_2,x_3), R(x_4,x_5,x_6), u_1(\neq) = v_1, \ldots, u_m(\neq) = v_m, \end{array}$$

(1) R is a nonrecursive predicate of arity 3,
(2) variables x_a, x_b, x_c and each u_j, v_j are contained in $\{x_1, \ldots, x_6\}$.

We then let e_S be $(e_R \bowtie_\theta^{a,b,c})^*$, where θ contains the inequality $p(\neq) = q$ for each predicate $x_p(\neq) = x_q$ in the rule above, or the respective comparison with constant if p or q belong to $\mathcal{O}$.

Once again, it is straightforward to verify that e_{Ans} is such that, $e_{Ans}(T) = \Pi(I_T)$, for all triplestores T. □

We now give an example of a simple datalog program computing the query from Theorem 2.3.

Example 4.3. *The following* ReachTripleDatalog$^\neg$ *program is equivalent to query Q from Theorem 2.3. 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 $Q = ((E \bowtie^{1,3',3}_{2=1'})^* \bowtie^{1,2,3'}_{3=1',2=2'})^*$. 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. We start with query evaluation, redefined here for TriAL* queries.

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, GXPath) 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 5.1. *The problem* QueryEvaluation *is* Ptime*-complete, and in* Nlogspace *if the algebra expression e is fixed.*

Proof. The Ptime upper bound follows immediately from Theorem 5.2 below. Ptime-hardness follows from the fact that every FO^3 query can be expressed in TriAL (Theorem **??**) and the known result that evaluating FO^k queries is Ptime-hard already when $k = 3$ [Vardi 1995].

For the Nlogspace upper bound, the idea is to divide the expression e into all its subexpression, corresponding to subtrees of the parsing tree of φ. Starting from the leaves until the root of the parse tree of e, one can guess the relevant triples that will be witnessing the presence of the query triple in the answer set $e(T)$.

Note that for this we only need to remember $O(|e|)$ triples – a number of fixed length. After we have guessed a triple for each node in the parse tree for e we simply check that they belong to the result of applying the subexpression defined by that node of the tree to our triplestore T. Thus to check that the desired complexity bound holds we need to show that each of the operations can be performed in Nlogspace, given any of the triples. This follows by an easy inductive argument.

For example, if $e = E_i$ is one of the initial relations in T, we simply check that the guessed triple is present in its table. Note that this can be done in Nlogspace.

This is done in an analogous way for the expressions of the form $e = e_1 \cup e_2$ and $e = e_1 - e_2$. To see that the claim also holds for joins, note that one only has to check that join conditions can be verified in Nlogspace. But this is a straightforward consequence of the observation that for conditions we use only comparisons of objects.

Finally, to see that the star operator $(R \bowtie^{i,j,k}_{\theta})^*$ can be implemented in Nlogspace we simply do a standard reachability argument for graphs. That is, since we are trying to

verify that a specific triple (a, b, c) is in the answer to the star-join operator, we guess the sequence that verifies this. We begin by a single triple in R (and we can check that it is there in NLOGSPACE by the induction hypothesis) and guess each new triple in R, joining it with the previous one, until we have performed at most $|T|$ steps. □

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. [Fan et al. 2011; 2010; Gottlob et al. 2002]), we consider an *array representation* for triplestores. That is, when representing a triplestore $T = (O, E_1, \ldots, E_m)$ with $O = \{o_1, \ldots, o_n\}$, 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. Using this representation we obtain the following bounds.

THEOREM 5.2. *The problem* QUERYCOMPUTATION *can be solved in time*

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

PROOF. The basic outline of the algorithm is as follows:

(1) Build the parse tree for our expression.
(2) Evaluate the subexpressions bottom-up.

Now to see that the algorithm meets the desired time bounds we simply have to show that each step of evaluating a subexpression can be performed in time $O(|T|^2)$.

We prove this inductively on the structure of subexpression e.

As stated previously, we assume that the objects are sorted and that the triplestore is given by its adjacency matrix T with the property that $T[i, j, k] = 1$ if and only if $(o_i, o_j, o_k) \in T$. If we are dealing with a triplestore that has more than one relation we will assume that we have access to each of the $n \times n \times n$ matrices representing E_i. Our algorithm computes, given an expression e and a triplestore T the matrix R_e such that $(o_i, o_j, o_k) \in e(T)$ iff $R_e[i, j, k] = 1$.

If $e = E_i$, the name of one of the initial triplestore matrices, we already have our answer, so no computation is needed.

If $e = R_1 \cup R_2$ and we are given the matrix representation of R_1 and R_2 (that is the adjacency matrix of the answer of R_i on our triplestore T) we simply compute R_e as the union of these two matrices. Note that this takes time $O(|T|)$.

If $e = R_1 \cap R_2$ we compute R_e as the intersection of these two matrices. That is, for each triple (i, j, k) we check if $R_1[i, j, k] = R_2[i, j, k] = 1$. Note that this takes time $O(|T|)$.

If $e = R_1 - R_2$ we compute R_e as the difference of the two matrices. That is for each (i, j, k) we set $R_e[i, j, k] = 1$ if and only if $R_1[i, j, k] = 1$ and $R_2[i, j, k] = 0$. The time required is $O(|T|)$.

If $e = \sigma_\varphi R_1$ and we are given the matrix for R_1 we can compute R_e in time $O(|e||T|)$ by traversing each triple (i, j, k), checking that $R_1[i, j, k] = 1$ and that the objects o_i, o_j and o_k satisfy the conditions specified by φ. Notice that each of these checks can be done in $|e|$ time using T, since the number of comparisons in φ has a fixed upper bound, modulo comparison with constants. The comparison with constants can be done in time $|e|$ because we have to check (in)equality only with the constants that actually appear in e.

Finally, in the case that $e = R_1 \bowtie_\theta^{i',j',k'} R_2$ we can compute R_e using the following algorithm:

Procedure 1 Computing joins

Input: Matrix representation of R_1, R_2
Output: Matrix R_e representing e

```
1: Let θ' be the conditions obtained from θ by removing comparisons with constants
2: Let α be the conditions in θ using constants
3: Filter R_1 and R_2 according to α
4: for i = 1 → n do
5:     for j = 1 → n do
6:         for k = 1 → n do
7:             if R_1[i, j, k] = 1 then
8:                 for l = 1 → n do
9:                     for m = 1 → n do
10:                        for n = 1 → n do
11:                            if R_2[l, m, n] = 1 then
12:                                if (o_i, o_j, o_k) and (o_l, o_m, o_n) satisfy the conditions in θ'
    then R_e[i', j', k'] = 1
13:                                else R_e[i', j', k'] = 0
```

Note that lines 1–3 correspond to computing selections operator and can therefore be performed using the time $O(|e||T|)$ and reusing the matrices R_1 and R_2. It is straightforward to see that the remaining of the algorithm works as intended by joining the desirable triples. This is performed in $O(|T|^2)$. Thus the whole join computation can be done in time $O(|T|^2)$.

This concludes the first part of our theorem and we thus conclude that TriAL query computation problem can be solved in time $O(|e||T|^2)$.

For the second part of the theorem we only have to show that each star operation can be computed in time $O(|T|^3)$. To see this we consider the algorithm in Procedure 2, computing the answer set for $e = (R_1 \bowtie_\theta^{i',j',k'})^*$

Procedure 2 Computing stars

Input: Matrix representation of R_1
Output: Matrix R_e representing e

```
1: Initialize R_e := R_1
2: for i = 1 → n^3 do
3:     Compute R_e := R_e ∪ R_e ⋈_θ^{i',j',k'} R_1
```

First we note that the algorithm does indeed compute the correct answer set. This follows because the joining in our star process has to became saturated after n^3 steps, since this is

the maximum possible number of triples in a model with n elements. Note now that each join in step 3 can be computed in time $O(|T|^2)$, thus giving us the total running time of $O(n^3 \cdot |T|^2) = O(|T|^3)$.

Finally, left-joins can be computed in an analogous way. □

Note that this immediately gives the PTIME upper bound for Proposition 5.1.

One can examine the proofs of Proposition 4.1 and Theorem 4.2 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 5.3. *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.*

5.1. Low-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 it even further. We now look at fragments of TriAL^* for which this is possible.

Relational fragments of TriAL. In algorithms from Theorem 5.2, 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 the conditions θ 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 5.4. *The* QUERYCOMPUTATION *problem for* $\mathsf{TriAL}^=$ *expressions can be solved in time* $O(|e| \cdot |O| \cdot |T|)$.

PROOF. To prove this we will use the close connection of positive fragment of $\mathsf{TriAL}^=$ with FO^4. We establish this as follows. To each triplestore $T = (O, E_1, \ldots, E_n)$ we associate an FO structure $\mathcal{M}_T = (O, E_1, \ldots, E_n)$, where O is the set of objects appearing in T, and $E_1, \ldots, E_n$ are just the representation of the triple relations. As before (see e.g. Section 4), we assume that the constants $o \in \mathcal{O}$ are present in our vocabulary and interpreted as themselves (note that we always have that $O \subset \mathcal{O}$), and we deploy the active domain semantics, therefore executing all the logical operations with O as the operative domain. In Lemma 5.5 we will then show that for each $\mathsf{TriAL}^=$ expression e one can compute, in time $O(|e|)$, an equivalent FO formula φ_e true precisely for the triples in $\mathcal{M}_T$ which satisfy e over T.

Note that we can compute $\mathcal{M}_T$ from T in linear time. To finish the proof we show in Lemma 5.6 that each FO^4 formula φ using relations that are at most ternary (in fact this holds for relations of arity four as well, but is not relevant for our analysis) can be evaluated in time $O(|\varphi| \cdot |O|^4)$.

The result of Proposition 5.4 now follows, since we can take our expression e, transform it into a formula φ_e of FO^4 and evaluate it in time $O(|\varphi_e| \cdot |O|^4) = O(|e| \cdot |O| \cdot |T|)$, since $|T| = |O|^3$ and $|\varphi_e| = O(|e|)$.

The proof of the two lemmas follows below. □

First we show that over triplestores $\mathsf{TriAL}^=$ is contained in FO^4.

LEMMA 5.5. *For every* $\mathsf{TriAL}^=$ *expression* e *one can construct an* FO^4 *formula* φ_e *such that a triple* (a, b, c) *belongs to* $e(T)$ *if and only if* $\mathcal{M}_T \models \varphi_e(a, b, c)$.

PROOF. The proof is done by induction. The base case when $e = E_i$ for some $1 \leq i \leq n$ is trivial, and so are the cases when $e = e_1 \cup e_2$, $e = e_1 - e_2$ and $e = \sigma_\theta e_1$. The only interesting case is when $e = e_1 \bowtie_\theta^{i,j,k} e_2$.

As usual, we assume that e is $e_1 \bowtie_\theta^{i,j,k} e_2$, where θ is a conjunction of equalities between elements in $\{1, 1', 2, 2', 3, 3'\} \cup \mathcal{O}$. We need some terminology.

Let $\theta = \theta_\ell \wedge \theta_r \wedge \theta_\bowtie \wedge \theta_\ell^c \wedge \theta_r^c$, where

— θ_ℓ and θ_r contain only equalities between indexes in $\{1, 2, 3\}$ and $\{1', 2', 3'\}$, respectively.
— θ_ℓ^c and θ_r^c contain only equalities where one element is in $\mathcal{O}$ and the other is in $\{1, 2, 3\}$ and $\{1', 2', 3'\}$, respectively.
— $\theta_\bowtie$ contains all the remaining equalities, i.e. those equalities in which one index is in $\{1, 2, 3\}$ and the other in $\{1', 2', 3'\}$.

Notice that any two equalities of form $i = j'$ and $i = k'$, for $i \in \{1, 2, 3\}$ and $j', k' \in \{1', 2', 3'\}$ can be replaced with $i = j'$ and $j' = k'$, and likewise we can replace $i = k'$ and $j = k'$ with $i = j$ and $j = k'$. For this reason we assume that $\theta_\bowtie$ contains at most 3 equalities, and no two equalities in them can mention the same element. Furthermore, if $\theta_\bowtie$ has two or more equalities, then the join can be straightforwardly expressed in FO^4, since now instead of the six possible positions we only care about four -or three-of them. For this reason we only show how to construct the formula when $\theta_\bowtie$ has one or no equalities.

Finally, for a conjunction θ of equalities between elements in $\{1, 1', 2, 2', 3, 3'\}$, we let $\alpha(\theta)$ be the formula $\bigwedge_{i=j \in \theta} x_i = x_j$, and for a conjunction θ^c of equalities between an object in $\mathcal{O}$ and an element in $\{1, 1', 2, 2', 3, 3'\}$ we let $\alpha(\theta^c) = \bigwedge_{o=i \in \theta^c} o = x_i$.

In order to construct formula φ_e, we distinguish 2 types of joins:

— Joins of form $e = e_1 \bowtie_\theta^{i,j,k} e_2$ where all of i, j, k belong to either $\{1, 2, 3\}$ or $\{1', 2', 3'\}$. Assume that i, j, k belong to $\{1, 2, 3\}$ (the other case is of course symmetrical). We first consider the case in which $\theta_\bowtie$ has no equalities. We then let

$$\varphi_e(x_i, x_j, x_k) = \varphi_{e_1}(x_1, x_2, x_3) \wedge \alpha(\theta_\ell) \wedge \alpha(\theta_\ell^c) \wedge \exists w \Big(\exists x_1 \big(\exists x_2 (\varphi_{e_2}(w, x_1, x_2) \wedge \alpha(\theta_r)[x_{1'}, x_{2'}, x_{3'} \to w, x_1, x_2] \wedge \alpha(\theta_r^c)[x_{1'}, x_{2'}, x_{3'} \to w, x_1, x_2]) \big) \Big)$$

Where a formula $\psi[x, y, z \to x', y', z']$ is just the formula ψ in which we replace each occurrence of variables x, y, z for x', y', z', respectively. For the case when $\theta_\bowtie$ is nonempty, notice here than any equality in $\theta_\bowtie$ only makes our life easier, since it eliminates one of the existential guesses we need in the above formula. This cover all other possible cases of $\theta_\bowtie$.

Let us illustrate this construction with an example.

Consider the expression $e = e_1 \bowtie_{1=2}^{1,2,3} e_2$. Then θ_ℓ is $1 = 2$, and all of the remaining formulas are empty. Then we have:

$$\varphi_e(x_1, x_2, x_3) = \varphi_{e_1}(x_1, x_2, x_3) \wedge x_1 = x_2 \wedge \exists w \exists x_1 \exists x_2 \varphi_{e_2}(w, x_1, x_2)$$

— Joins of form $e = e_1 \bowtie_\theta^{i,j,k} e_2$ where not all of i, j, k belong to either $\{1, 2, 3\}$ or $\{1', 2', 3'\}$. Assume for the sake of readability that $i = 1$, $j = 2$ and $k = 3'$ (all of other cases are completely symmetrical). We have again two possibilities:
  — There are no equalities in $\theta_\bowtie$. We then let

$$\varphi_e(x_1, x_2, x_{3'}) = \big(\exists x_3(\varphi_{e_1}(x_1, x_2, x_3) \wedge \alpha(\theta_\ell) \wedge \alpha(\theta_\ell^c))\big) \wedge$$
$$\exists x_3 \exists x_1 \big(\varphi_{e_2}(x_3, x_1, x_{3'}) \wedge \alpha(\theta_r)[x_{1'}, x_{2'} \to x_3, x_1] \wedge \alpha(\theta_r^c)[x_{1'}, x_{2'} \to x_3, x_1]\big)\Big)$$

— There is a single equality in $\theta_{\bowtie}$. Assume for the sake of readability that $i = 1$, $j = 2$ and $k = 3'$ (all of other cases are completely symmetrical). Notice that if $\theta_{\bowtie}$ has the equality $3 = 3'$, then this is equivalent to the previous case with one equality in $\theta_{\bowtie}$, but with $k = 3$. Moreover, equalities in $\theta_{\bowtie}$ involving 1 or 2 just make our life easier, so we will also not take them into account here. We are thus left with the assumption that $\theta_{\bowtie}$ contains the equality $3 = 1'$ (the case where it contains instead $3 = 2'$ is symmetrical). We then let

$$\varphi_e(x_1, x_2, x_{3'}) =$$
$$\exists x_{1'}\Big(\varphi_{e_1}(x_1, x_2, x_{1'}) \wedge \alpha(\theta_\ell)[x_3 \to x_{1'}] \wedge \alpha(\theta_\ell^c)[x_3 \to x_{1'}] \wedge$$
$$\exists x_1 \big(\varphi_{e_2}(x_{1'}, x_1, x_{3'}) \wedge x_{1'} = x_{3'} \wedge \alpha(\theta_r)[x_{2'} \to x_1] \wedge \alpha(\theta_r^c)[x_{2'} \to x_1]\big)\Big)$$

Having established how to construct φ_e, it is now straightforward to show that it satisfies the property of Lemma 5.5. It is also readily observed that the size of formula φ_e corresponding to e is $O(|e|)$. □

To finish the proof of Proposition 5.4 we show that FO^4 formulas can be evaluated efficiently.

LEMMA 5.6. *Let φ be an arbitrary formula using at most four variables. Then the set of all tuples that make φ true in $\mathcal{M}$, with $\mathcal{M}$ as above (we omit the subscript T for the sake of readability, since it is now clear), can be computed in time $O(|\varphi| \cdot |O|^4)$.*

PROOF. To see that this holds note that we can assume that our formulas only use the connectives $\neg, \vee$ and the quantifier $\exists$. Indeed, we can assume this since any formula using other quantifiers can be rewritten using the ones above with a constant blow-up in the size of formula. In particular, our formulas in Lemma 5.5 use only $\wedge$ in addition to these three logical connectives, and $\wedge$ can be rewritten in terms of $\vee$ and $\neg$.

The desired algorithm works as follows.

(1) Build a parse tree for the formula φ.
(2) Compute the output relation(s) bottom-up using the tree.

To see that the algorithm works within the desired time bound we only have to make sure that each of the computation steps in 2 can be performed in time $O(|O|^4)$. We have three cases to consider, based on whether we are using negation, disjunction, or existential quantification. Here we assume that we compute a matrix $\psi(\mathcal{M})$, for each subformula ψ of φ. Note that, since we use formulas with at most four free variables each matrix can be of size at most $|O|^4$ (i.e. we are working with a four dimensional matrix). If the (sub)formula has only two free variables the resulting matrix will, of course, be two dimensional.

First we consider the case of negation. That is, assume that we have a matrix $\psi(\mathcal{M})$ and we are evaluating a formula $\varphi = \neg\psi$. Then we simply build a matrix for the $\varphi(\mathcal{M})$ by flipping each bit in the matrix for $\psi(\mathcal{M})$. This can clearly be done in time $O(|O|^4)$ by traversing the entire matrix.

Next, consider the case when $\varphi = \exists x \psi(x, y, z, w)$ and assume that we have the matrix for $\psi(x, y, z, w)$. The existing matrix is now reduced to a three dimensional matrix with the

value 1 in position i, j, k if and only if there is an l such that $\psi(\mathcal{M})[l, i, j, k] = 1$. Note that computing this amounts to scanning the entire matrix for ψ. In the case when ψ case only three free variables we will need only $O(|O|^3)$ time to compute $\varphi(\mathcal{M})$.

Finally, let $\varphi = \psi_1(x, y, w) \vee \psi_2(x, y, z, w)$. The cases when ψ_1 and ψ_2 have a different number of free variables follows by symmetry. What we do first is to compute a 4-D matrix $\psi_1'(\mathcal{M})$ by setting $\psi_1'(\mathcal{M})[i, j, k, l] = 1$ iff $\psi_1(\mathcal{M})[i, j, l] = 1$. Note that this matrix can be computed in time $O(|O|^4)$. Next we compute the output matrix by putting 1 in each cell where either $\psi_1'(\mathcal{M})$ or $\psi_2(\mathcal{M})$ have 1. All the other cases can be performed symmetrically by using the appropriate matrices and their projections.

This completes the proof of Lemma 5.6. □

Navigational fragments. 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 5.2 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 5.7. *The problem* QUERYCOMPUTATION *for* $\mathsf{reachTA}^=$ *can be solved in time* $O(|e| \cdot |O| \cdot |T|)$.

PROOF. To show this we will use the algorithm presented in Proposition 5.4. All of the operations except the evaluation of Kleene star will be preformed in a same way as there. Note that we can assume this since the algorithm in Lemma 5.6 computes the subexpressions bottom up using the matrices representing the output. Thus we can use it to compute answers to subformulas, compose it with the method presented here to evaluate Kleene stars and proceed with the algorithm from Lemma 5.6. To obtain the desired complexity bound we only have to show how to compute navigational operations in time $O(|O| \cdot |T|)$.

That is, we show how to, given a matrix representation for a relation R we compute matrix representation for $(R \bowtie^{1,2,3'}_{3=1'})^*$ and $(R \bowtie^{1,2,3'}_{3=1',2=2'})^*$, respectively.

Let $O = \{o_1, \ldots, o_n\}$ be the set of object appearing in our triplestore T. (The assumption that they are ordered is standard when considering matrix representations). As input, we are given a three dimensional matrix R representing the output of relation R when evaluated over T. That is we have $(o_i, o_j, o_k) \in R(T)$ if and only if $R[i, j, k] = 1$. (Here we use R both to denote the relation R and its matrix representation).

First, in Procedure 3, we give a procedure that computes the matrix M_e for the expression

$$e = (R \bowtie^{1,2,3'}_{3=1'})^*.$$

To show that the algorithm works correctly notice that steps 1 to 6 precompute the matrix R_{reach} such that $R_{reach}[i, j] = 1$ if and only if o_i has and out edge ending in o_j (or equivalently $(o_i, o, o_k) \in T$ for some o). After this in step 7 we compute the transitive closure R^*_{reach} thus obtaining all pairs of nodes reachable one from another using path of arbitrary label in the graph representing T. Next in steps 8 to 15 we simply compute all the triples in the output matrix M_e. To do so we observe that a pair (o_i, o_k) will belong to

Procedure 3 Computing $e = (R \bowtie^{1,2,3'}_{3=1'})^*$

Input: Matrix representation of R
Output: Matrix M_e representing e

```
1: Precomputing the reachability matrix R_reach:
2: for i = 1 → n do
3:     for j = 1 → n do
4:         for k = 1 → n do
5:             if R[i,k,j] = 1 then
6:                 R_reach[i,j] = 1
7: Compute the transitive closure R*_reach
8: Compute the output matrix M_e:
9: for i = 1 → n do
10:    for j = 1 → n do
11:        for k = 1 → n do
12:            if R[i,k,j] = 1 then
13:                for l = 1 → n do
14:                    if R*_reach[j,l] = 1 then
15:                        M_e[i,k,l] = 1
```

Procedure 4 Computing $e = (R \bowtie^{1,2,3'}_{3=1',2=2'})^*$

Input: Matrix representation of R
Output: Matrix M_e representing e

```
1: for k = 1 → n do
2:     Precomputing the reachability matrix R^k_reach:
3:     for i = 1 → n do
4:         for j = 1 → n do
5:             if R[i,k,j] = 1 then
6:                 R_reach[i,j] = 1
7:     Compute the transitive closure (R^k_reach)*
8:     Compute the output matrix M_e:
9:     for i = 1 → n do
10:        for j = 1 → n do
11:            if R[i,k,j] = 1 then
12:                for l = 1 → n do
13:                    if (R^k_reach)*[j,l] = 1 then
14:                        M_e[i,k,l] = 1
```

some triple (o_i, o_k, o_l) of the output, if there is j such that $(o_i, o_k, o_j) \in T$ (line 12) and o_l is reachable from o_j (line 14).

To determine the complexity of the algorithm notice that steps 1 to 6 take time $O(|O|^3) = O(|T|)$, while computing the transitive closure in step 7 can be done using Warshall's algorithm (see T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, The MIT Press, 2003.) in time $O(|O|^3) = O(|T|)$. Finally steps 8 to 15 take time $O(|O| \cdot |T|)$, thus giving us the desired time bound.

Next, in Procedure 4, we show how to compute joins of the form $(R \bowtie^{1,2,3'}_{3=1',2=2'})^*$ using a slight modification of the algorithm above.

It is straightforward to see that the algorithm uses the same time to compute the output as the algorithm in Procedure 3.

To show that it works correctly observe that we precompute matrix R^k_{reach} for each k, thus checking reachability only for triples whose second node is o_k. Since the rest of the algorithm works in the same way as the one in Procedure 3, we conclude that the computed answer M_e represents e correctly. □

6. TRIPLE ALGEBRA AND OTHER LANGUAGES

We have provided a declarative specification for TriAL and TriAL*, but so far we do not know how our algebras compare with other popular languages for relational databases, RDF or graph databases.

The goal of this section is to compare the expressive power of TriAL and TriAL* with other popular languages for relational databases, RDF or graph databases. In order to be able to perform this comparison we first need a common yardstick for all of these formalisms. We use First–Order Logic (FO) to gauge TriAL in terms of relational query languages and the infinitary logic $\mathcal{L}^w_{\infty,w}$ to compare TriAL* with graph query languages that are also equipped with a star operator. 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.

Since TriAL is a restriction of relational algebra, of course it is contained in FO, and similarly TriAL* is contained in $\mathcal{L}^w_{\infty,w}$. But several querying formalisms for graph or semistructured data have been shown to be equivalent to fragments of these logics with restricted variables, such as XPath and FO with only 3 variables (FO^3) over trees [Marx 2005] or navigational graph query languages and FO^3 over graphs [Fletcher et al. 2011]. Thus, to begin our study, we show that the expressive power of TriAL is strictly between that of the fragments of FO using 3 and 6 variables, and that TriAL* lies strictly between the fragments of $\mathcal{L}^w_{\infty,w}$ using 3 and 6 variables. To show these results we need to develop a good deal of technical machinery, including a notion of games characterizing expressivity in TriAL and TriAL*. We then use our machinery to compare the expressive power of our algebras with several established formalisms. Most notably, we show that TriAL* strictly contains Graph XPath, the graph database adaptation of navigational XPath for XML databases [Libkin et al. 2013], and that it is incomparable with register automata and other graph query languages supporting data values from [Libkin and Vrgoč 2012].

6.1. Triple Algebra and First–Order Logic

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 $\langle E_1, \ldots, E_n, \sim \rangle$.

As we have mentioned, TriAL is trivially contained in FO, and 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.

PROPOSITION 6.1. TriAL *is contained in* FO^6.

PROOF. Let e be a TriAL expression. We construct an FO^6 formula φ_e such that $e(T) = \varphi_e(I_T)$, for each triplestore T. The proof is by induction.

— For the base case, if e corresponds to a triplestore name E, then φ_e is $E(x, y, z)$.
— If $e = e_1 \cup e_2$, then $\varphi_e(x, y, z) = \varphi_{e_1}(x, y, z) \vee \varphi_{e_2}(x, y, z)$, which clearly is in FO^6 since existential variables within φ_{e_1} and φ_{e_2} can be renamed and reused.
— If $e = e_1 - e_2$, then $\varphi_e(x, y, z) = \varphi_{e_1}(x, y, z) \wedge \neg\varphi_{e_2}(x, y, z)$
— If $e = e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$, then $\varphi_e(x_i, x_j, x_k) = \exists x_u \exists x_v \exists x_w \varphi_{e_1}(x_1, x_2, x_3) \wedge \varphi_{e_2}(x_{1'}, x_{2'}, x_{3'}) \wedge \alpha(\theta) \wedge \beta(\eta)$, where u, v, w are the remaining elements that together with i, j, k complete $\{1, 1', 2, 2', 3, 3'\}$, $\alpha(\theta)$ is a conjunction of one equality $x_p = x_q$ or $x_p = o$ for each equality $p = q$ or $p = o$ in θ, and one inequality $x_p \neq x_q$ or $x_p \neq o$ for each inequality $p \neq q$ or $p \neq o$ in θ, for $o \in \mathcal{O}$ and $p, q \in \{1, 1', 2, 2', 3, 3'\}$; and $\beta(\eta)$ is a conjuntion of atoms $\sim(x_p, x_q)$ for each equality $\rho(p) = \rho(q)$ in η, and atoms $\neg\sim(x_p, x_q)$ for each inequality $\rho(p) \neq \rho(q)$ in η.
— Similarly, if $e = \sigma_{\theta,\eta} e_1$ then $\varphi_e(x, y, z) = \varphi_{e_1}(x, y, z) \wedge \alpha(\theta) \wedge \beta(\eta)$, where $\alpha(\theta)$ and $\beta(\eta)$ are defined as in the previous bullet.

It is now straightforward to check the desired properties for e and φ_e. □

All the subformulas used in this construction have just three free variables, so it seems plausible to believe that the containment shown above is strict. At the same time, one could ask what happens with fragments of FO using fewer 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. As it turns our, the expressive power of TriAL lies strictly between FO^3 and FO^6.

THEOREM 6.2.

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

We prove this theorem by showing that TriAL is equivalent to a fragment of FO whose expressive power is also as in the above theorem. We denote this fragment as FO^3-join, and we define it next.

Consider first the following extension of FO with a *join* operator, that we denote by FO-join. If φ_1 and φ_2 are formulas in FO-join formulas with free variables x_1, x_2, x_3 and $x_{1'}, x_{2'}, x_{3'}$ respectively, θ is a conjunction of equalities and inequalities between indexes in $\{1, 1', 2, 2', 3, 3'\}$ and η is a conjunction of equalities and inequalities between indexes in $\rho(1), \ldots, \rho(3')$, then the formula $\varphi(x_i, x_j, x_k) = \varphi_1(x_1, x_2, x_3) \bowtie^{i,j,k}_{\theta,\eta} \varphi_2(x_{1'}, x_{2'}, x_{3'})$ is also an FO-join formula. The semantics of the join construct is defined in the same way as for Triple Algebra: a structure I and an assignment $\sigma : \{x_i, x_j, x_k\} \to \mathbf{U}$ satisfy $\varphi_1(x_1, x_2, x_3) \bowtie^{i,j,k}_{\theta,\eta} \varphi_2(x_{1'}, x_{2'}, x_{3'})$ if σ can be extended to an assignment $\sigma' : \{x_1, x_2, x_3, x_{1'}, x_{2'}, x_{3'}\} \to \mathbf{U}$ such that (1) $(I, \sigma') \models \varphi_1$ and $(I, \sigma') \models \varphi_2$; (2) $\sigma(x_p) = \sigma(x_q)$ for every $p = q$ in θ and $\sigma(x_p) \neq \sigma(x_q)$ for every $p \neq q$ in θ; and (3) $\rho(\sigma(x_p)) \sim \rho(\sigma(x_q))$ for every $\rho(p) = \rho(q)$ in η and $\rho(\sigma(x_p)) \not\sim \rho(\sigma(x_q))$ for every $\rho(p) \neq \rho(q)$ in η.

To define the language FO^3-join we restrict the variables used in the formulas of this language; the intuition is that we want to restrict formulas to use just three variables in all cases except when a join is defined. To formalize this we define the set $\mathrm{var}(\varphi)$ of variables used by φ by induction: the variables used by $E(x, y, z)$ are $\{x, y, z\}$ and the variables used by $x = y$ are $\{x, y\}$; the variables used by $\varphi_1 \vee \varphi_2$ or $\varphi_1 \wedge \varphi_2$ is the union of the set of variables used in φ_1 and the set of variables used in φ_2; the variables used in $\forall x\varphi$, $\exists x\varphi$ and $\neg\varphi$ are the same as the variables used in φ, and finally the variables used in

$\varphi(x_i, x_j, x_k) = \varphi_1(x_1, x_2, x_3) \bowtie^{i,j,k}_{\theta,\eta} \varphi_2(x_{1'}, x_{2'}, x_{3'})$ are just $\{x_i, x_j, x_k\}$. Then FO^3-join is defined as the fragment of all FO-join formulas that use only three variables.

As promised, we show that FO^3-join captures Triple Algebra.

LEMMA 6.3. TriAL *is equivalent to* FO^3*-join.*

PROOF. The proof that TriAL is contained in FO^3-join resembles the proof of Proposition 6.1, as most of the translation from TriAL to FO^6 in this proof uses just three variables. The only exception is the join operator, that we can instead simulate with the join operator of our logic. Let e be a TriAL expression. We construct an FO^3-join formula φ_e such that $e(T) = \varphi_e(I_T)$, for each triplestore T, exactly as in the proof of Proposition 6.1, with the exemption of the inductive case when $e = e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$. In this case, $\varphi_e(x_i, x_j, x_k)$ corresponds to $\varphi_{e_1}(x_1, x_2, x_3) \bowtie^{i,j,k}_{\theta,\eta} \varphi_{e_2}(x_{1'}, x_{2'}, x_3)$, where $i, j, k \in \{x_1, x_2, x_3, x_{1'}, x_{2'}, x_{3'}\}$ using the appropriate renaming of variables for φ_{e_1} and φ_{e_2}. It is once again straightforward to check the desired properties for e and φ_e.

To show that FO^3-join is contained in TriAL, one needs to show how to construct, for every FO^3-join formula φ, an equivalent TriAL expression e_φ such that $e_\varphi(T) = \varphi(I_T)$, for all triplestores T. Once again, the construction is done by induction on the formula. During the induction case we assume that the variables used in e are x_1, x_2 and x_3. Furthermore, recall that U is just a shorthand for the relation that contains O^3.

— For the base case, if $\varphi = E(x_1, x_2, x_3)$ for some triplestore name E, then e_φ is just E. However, in general case when $\varphi = E(x_i, x_j, x_k)$, with each of x_i, x_j, x_k are (not necessarily distinct) variables in $\{x_1, x_2, x_3\}$, we let $e_\varphi = E \bowtie^{i,j,k} E$. For the other base case when φ is $x_i = x_j$, then $e_\varphi = U \bowtie^{1,2,3}_{i=j} U$.
— If $\varphi = \neg\varphi_1$, then $e_\varphi = U - e_{\varphi_1}$ (recall that we assume active domain semantics for FO formulas in general).
— If $\varphi = \exists x_i \varphi_1$, then $e_\varphi = e_{\varphi_1} \bowtie^{\bar{d}} U$, where $\bar{d}$ depends x_i: $\bar{d}$ is $1', 2, 3$ if $i = 1$, $1, 2', 3$ if $i = 2$ and $1, 2, 3'$ if $i = 3$. Intuitively, when quantifying x_i we do a projection on the i-th attribute of the computed relation. Since we cannot deal with relations of less arity, we put instead all possible values from O.
— If $\varphi = \varphi_1 \vee \varphi_2$, then $e_\varphi = e_{\varphi_1} \cup e_{\varphi_2}$ (again, this works because we assume active domain semantics for FO formulas).
— If $\varphi = \varphi_1(y_1, y_2, y_3) \bowtie^{i,j,k}_{\theta,\eta} \varphi_2(y_{1'}, y_{2'}, y_{3'})$, then $e_\varphi = e_{\varphi_1} \bowtie^{i,j,k}_{\theta,\eta} e_{\varphi_2}$.

It is also a straightforward task to check that φ and e_φ satisfy our desired properties. □

Clearly FO^3-join lies between FO^3 and FO^6: by definition it can express all FO^3 queries, and as we saw in Proposition 6.1 any FO^3-join query can be expressed in FO^6. The next theorem shows that TriAL is incomparable with these fragments, which concludes the proof of Theorem 6.2.

PROPOSITION 6.4. FO^3*-join is incomparable with* FO^4 *and* FO^5*.*

PROOF. We divide this proof into two parts: first we construct an FO^3-join formula that cannot be expressed in FO^5 and then we construct an FO^4 formula that cannot be expressed in FO^3-join. We focus on the simplest

For the first part we show that the following FO^3-join formula cannot be expressed in FO^5 (and thus not in FO^4):

$$e_6 := \exists x_1 \exists x_2 \exists x_3 \big(E(x_1, x_2, x_3) \overset{1,2,3}{\underset{\theta}{\bowtie}} E(x_{1'}, x_{2'}, x_{3'})\big), \text{ with } \theta = \bigwedge_{i,j \in \{1,2,3,1',2',3'\}, i \neq j} i \neq j,$$

This query states that our triplestore has two tuples in E where all objects are different. Now take $T_5 = (O_5, E_5, \rho_5)$ with $O_5 = \{a, b, c, d, e\}$, and $E_5 = O_5 \times O_5 \times O_5$, where ρ_5 assigns the same data value to all elements of O_5, and define T_6 in an analogous way, but with six elements. It is a well known fact [Libkin 2004] that the duplicator has a winning strategy in a 5-pebble game on these two structures, so they can not be distinguished by an FO^5 formula. On the other hand our expression e_6 does distinguish them and is thus not expressible in FO^5.

Before showing the FO^4 formula for the first part we need to develop some technical machinery that allows us to show inexpressibility over FO^3-join.

Let $\mathcal{J}$ be the set of all the join symbols that we allow in TriAL, and $\neg\mathcal{J}$ be the set $\{\neg\bowtie | \bowtie \in \mathcal{J}\}$. A *recipe* p for FO^3-join is a tree where each node is labeled with symbols from the alphabet $\mathcal{J} \cup \neg\mathcal{J} \cup \{\exists, \forall\}$, each node can have at most two children, and such that the following holds: If a node n of p has two children, then it is labeled with a symbol in $\mathcal{J} \cup \neg\mathcal{J}$, and if a node n of p has one child, then it is labeled with $\exists$ or $\forall$.

For every such recipe p, define the *quantifier class* $L(p)$ inductively as follows:

— $L(\varepsilon)$ contains quantifier and join free formulae.
— If the root of p is labeled with $Q \in \{\exists, \forall\}$, then $L(p)$ is the closure under conjunctions and disjunctions of the class $L(p') \cup \{Qx\varphi \mid \varphi \in L(p')\}$, where p' is the subtree of p whose root is the only child of p.
— If the root of p is labeled with a symbol $\bowtie$ in $\mathcal{J}$, let p_1 and p_2 be the subtrees of p whose roots are the first and the second child of p, respectively. Then $L(p)$ is the closure under conjunctions and disjunctions and of the class of all formulae $\varphi \bowtie \psi$, where $\varphi \in L(p_1)$ and $\psi \in L(p_2)$.
— If the root of p is labeled with a symbol $\neg\bowtie$ in $\neg\mathcal{J}$, let p_1 and p_2 be the subtrees of p whose roots are the first and the second child of p, respectively. Then $L(p)$ is the closure under conjunctions and disjunctions and of the class of all formulae $\neg(\varphi \bowtie \psi)$, where $\varphi \in L(p_1)$ and $\psi \in L(p_2)$.

We now define the join game between two structures $\mathcal{A}$ and $\mathcal{B}$. The games is played between two players, the spoiler and the duplicator. It proceeds as in a typical 3-pebble game (see [Libkin 2004] for a precise explanation), except that now the spoiler has two extra moves avaliable. For ease of exposition we only deal with vocabularies without constants, although the game can be modified to allow for constants as well.

There are nine pairs of pebbles: three main pairs $(x_1, y_1), (x_2, y_2), (x_3, y_3)$, and size auxiliary pebbles $(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_{1'}, v_{1'}), (u_{2'}, v_{2'}), (u_3, v_{4'})$. In every round of the game the spoiler starts by choosing and performing one of the following moves, to which the duplicator must respond.

(1) The existential move: The spoiler picks a pebble x_i from the main three pairs and locates this pebble on an element of $\mathcal{A}$. Duplicator must respond by locating the corresponding pebble y_i on an element of $\mathcal{B}$.
(2) The universal move: The spoiler picks a pebble y_i from the main three pairs and locates this pebble on an element of $\mathcal{B}$. Duplicator must respond by locating the corresponding pebble x_i on an element of $\mathcal{A}$.
(3) The positive join $\bowtie^{i,j,k}_{\theta,\eta}$ move:
This move can only be performed when all 3 pebbles from the main set have already been placed on both structures. Assume that pebbles x_1, x_2 and x_3 are placed on elements a_1, a_2, a_3 in $\mathcal{A}$, respectively. The spoiler picks all six auxiliary pebbles $u_1, u_2, u_3, u_{1'}, u_{2'}, u_{3'}$, and locate all of them in elements $p_1, p_2, p_3, p_{1'}, p_{2'}, p_{3'}$ of

$\mathcal{A}$, respectively, with the condition that all pebbled elements *satisfy* $(a_1, a_2, a_3) = (p_1, p_2, p_3) \bowtie^{i,j,k}_{\theta,\eta} (p_{1'}, p_{2'}, p_{3'})$. The spoiler then removes pebbles (x_1, x_2, x_3) from $\mathcal{A}$.
The duplicator must then place the corresponding pebbles $v_1, v_2, v_3, v_{1'}, v_{2'}, v_{3'}$ $\mathcal{B}$, in the same fashion as the spoiler, also satisfying the join, and then remove from $\mathcal{B}$ pebbles y_1, y_2, y_3.
Spoiler then has two choices: either remove pebbles u_1, u_2, u_3 from $\mathcal{A}$ and replace pebbles $u_{1'}, u_{2'}, u_{3'}$ with x_1, x_2, x_3, respectively, in $\mathcal{A}$, or instead remove $u_{1'}, u_{2'}, u_{3'}$ and replace u_1, u_2, u_3 with x_1, x_2, x_3. In the first case the duplicator removes pebbles v_1, v_2, v_3 from $\mathcal{B}$ and replaces pebbles $v_{1'}, v_{2'}, v_{3'}$ with y_1, y_2, y_3, respectively, in $\mathcal{B}$; and in the second case the duplicator removes $v_{1'}, v_{2'}, v_{3'}$ and replaces v_1, v_2, v_3 with y_1, y_2, y_3

(4) The negative join $\bowtie^{i,j,k}_{\theta,\eta}$ move:
This move goes in the same fashion as the positive version, except that $\mathcal{A}$ and $\mathcal{B}$ are interchanged. Assume that pebbles y_1, y_2 and y_3 are placed on elements b_1, b_2, b_3 in $\mathcal{B}$, respectively. The spoiler picks all six auxiliary pebbles $v_1, v_2, v_3, v_{1'}, v_{2'}, v_{3'}$, and locate all of them in elements $p_1, p_2, p_3, p_{1'}, p_{2'}, p_{3'}$ of $\mathcal{B}$, respectively, with the condition that all pebbled elements *satisfy* $(b_1, b_2, b_3) = (p_1, p_2, p_3) \bowtie^{i,j,k}_{\theta,\eta} (p_{1'}, p_{2'}, p_{3'})$. The spoiler then removes pebbles (y_1, y_2, y_3) from $\mathcal{B}$.
The duplicator must then place the corresponding pebbles $u_1, u_2, u_3, u_{1'}, u_{2'}, u_{3'}$ $\mathcal{A}$, in the same fashion as the spoiler, also satisfying the join, and then remove from $\mathcal{B}$ pebbles x_1, x_2, x_3.
Spoiler then has two choices: either remove pebbles v_1, v_2, v_3 from $\mathcal{B}$ and replace pebbles $v_{1'}, v_{2'}, v_{3'}$ with y_1, y_2, y_3, respectively, in $\mathcal{B}$, or instead remove $v_{1'}, v_{2'}, v_{3'}$ and replace v_1, v_2, v_3 with y_1, y_2, y_3. duplicator must follow with the corresponding actions on $\mathcal{A}$.

After each round, another round is played if the main pebbles determine a local isomorphism from $\mathcal{A}$ to $\mathcal{B}$. Formally, let a_1, a_2, a_3 and b_1, b_2, b_3 be the elements carrying pebbles x_1, x_2, x_3 and y_1, y_2, y_3, respectively, and consider the mapping f that maps each a_i to b_i. Then we say that there is a local isomorphism from $\mathcal{A}$ to $\mathcal{B}$ if f defines an isomorphism on the substructures of $\mathcal{A}$ and $\mathcal{B}$ generated by the pebbled elements. The spoiler wins the game if at the end of any round the main pebbles do not determine a local isomorphism from $\mathcal{A}$ to $\mathcal{B}$. The duplicator wins the game if the spoiler is not capable of finding a winning strategy after any number of rounds. For every recipe p of FO^3-join we also define the set of $L(p)$-join games. It contains all join games in which the sequence of moves performed by the spoiler are described by a path from the root of p to one of its leaves.

Let L be a class of FO^3-join formulae and $\mathcal{A}$ and $\mathcal{B}$ structures of vocabulary $\langle E_1, \ldots, E_n, \sim, \bar{a} \rangle$. We write $\mathcal{A} \preceq_L \mathcal{B}$ if $\mathcal{A} \models \varphi$ implies $\mathcal{B} \models \varphi$, for every sentence $\varphi \in L$.

LEMMA 6.5. *If the duplicator wins on all $L(p)$-join games, then $\mathcal{A} \preceq_{L(p)} \mathcal{B}$.*

Before we prove this Lemma, we make two observations. We have included a tuple of constants in our vocabulary. This is to allow the use of sentences that start with a join operator, since we need this for the inductive case. Moreover, note that If, in a join game a pebble has already been placed on element $a \in \mathcal{A}$, then the remainder of the game can be considered as a game with two pebbles on $(\mathcal{A}, a)$, and the same holds when a greater number of pebbles has already been placed.

PROOF. We prove the contrary: If there is a sentence φ of class $L(p)$ such that $\mathcal{A} \models \varphi$ but $\mathcal{B} \not\models \varphi$, then the spoiler has a winning strategy for the $L(p)$-join game. We prove this by induction on the height of p.

The case when p is empty is trivial.

For the inductive case, assume that Lemma holds for all recipes of height k, and let p be a recipe of height $k+1$. Furthermore, assume that there is a sentence φ such that $\mathcal{A} \models \varphi$, but $\mathcal{B} \not\models \varphi$. We will construct a winning strategy for the spoiler. If φ is a boolean combinations

of formulas, then the two structures are distinguished by at least one of them. We are thus left with the following cases:

— The case when φ is of form $\exists\psi(\bar{x})$, where $\bar{x}$ is a tuple of at most two variables, and ψ has depth at most $k-1$ and belongs to $L(q)$, where q is the subtree whose root is the single child of p. In this case the spoiler can win as follows. In his first move he places one main pebble in an element a such that $(\mathcal{A}, a) \models \psi$. Since by hypothesis we have that $\mathcal{B} \not\models \varphi$, then no matter in which element $b \in \mathcal{B}$ the duplicator places its pebble, we know that $(\mathcal{B}, b) \not\models \psi$, and thus the spoiler has a winning strategy for the remainder of the truncated game.
— φ is of form $\forall\psi(\bar{x})$, in which case the strategy is analogous to the previous one, but with $\mathcal{A}$ and $\mathcal{B}$ interchanged.
— $\varphi(a,b,c)$ is of form $\varphi_1 \bowtie \varphi_2$, for some $\bowtie$ in $\mathcal{J}$, and where a, b and c are constants in our vocabulary. Then p has two children p_1 and p_2, both of height $\leq k$, and $\varphi_1 \in L(p_1)$, $\varphi_2 \in L(p_2)$. Since $\mathcal{A} \models \varphi(a,b,c)$, yet $\mathcal{B} \not\models \varphi(a,b,c)$, spoiler can win by first placing pebbles on the interpretation of elements a, b, c over $\mathcal{A}$, and splitting pebbles, placing them into sets (a_1, b_1, c_1) and (a_2, b_2, c_2) of elements in $\mathcal{A}$ such that $(a,b,c) = (a_1, b_1, c_1) \bowtie (a_2, b_2, c_2)$. Given that $\mathcal{B} \not\models \varphi(a,b,c)$, then for every pair (d_1, e_1, f_1) and (d_2, e_2, f_2) of elements in $\mathcal{B}$ such that $(a,b,c) = (d_1, e_1, f_1) \bowtie (d_2, e_2, f_2)$, it must be the case that either $(\mathcal{B} \not\models \varphi_1(d_1, e_1, f_1)$ or $(\mathcal{B} \not\models \varphi(d_2, e_2, f_2)$. Depending on the move of the duplicator, spoiler chooses the set accordingly, and continues to win the truncated game on $(\mathcal{A}, a_i, b_i, c_i)$ and $(\mathcal{B}, d_i, e_i, f_i)$, for $i = 1$ or $i = 2$.
— $\varphi(a,b,c)$ is of form $\neg(\varphi_1 \bowtie \varphi_2)$, for some $\bowtie$ in $\mathcal{J}$, and where a, b and c are constants in our vocabulary. Then p has two children p_1 and p_2, both of height $\leq k$, and $\varphi_1 \in L(p_1)$, $\varphi_2 \in L(p_2)$. In this case, since $\mathcal{A} \models \varphi(a,b,c)$, yet $\mathcal{B} \not\models \varphi(a,b,c)$, we have that $\mathcal{A} \not\models (\varphi_1 \bowtie \varphi_2)$, but $\mathcal{B} \models (\varphi_1 \bowtie \varphi_2)$. the spoiler can then use the same strategy outlined in the previous case, except with $\mathcal{A}$ and $\mathcal{B}$ interchanged.

□

We now continue with the proof of the Theorem. Due to Lemma 6.5, all that is left to do is to show structures $\mathcal{A}$ and $\mathcal{B}$ such that the duplicator can win any join game, and yet they are distinguished by an FO^4 formula. For simplicity we use the simpler vocabulary $\langle E \rangle$. Structures $\mathcal{A}$ and $\mathcal{B}$ are defined over the same domain of elements $D = \{a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3, d_1, d_2, d_3, e\}$.

++++++ falta la nueva parte con las nuevas estructuras, hay que mejorar y hay que mostrar este lemma! +++++++++

LEMMA 6.6. *Consider a set $S \subseteq D$ of three elements and a partial isomorphism $f : S \to D$ from $\mathcal{A}$ to $\mathcal{B}$. Then for every pair (P, P') of subsets of D of three elements one can find subsets R and R' of D, of three elements each and functions $h : R \cup S \to D$ and $h' : R' \cup S \to D$ that extend f, and such that both h and h' are partial isomorphisms from $\mathcal{A}$ to $\mathcal{B}$. The same holds when interchanging $\mathcal{A}$ for $\mathcal{B}$.*

Note that the above lema guarantees that every join move can be continued: every time the spoiler places the additional pebbles, the duplicator respond by selecting the elements that correspond to the image of h and h'.

From Lemma 6.5 we obtain that $\mathcal{A}$ and $\mathcal{B}$ agree on all FO^3-join formulas. However, it is not difficult to see that they do not agree to the following FO^4 formula (which asks for a 4-clique and thus is only true in $\mathcal{B}$).

+++ revisar +++

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

where $\psi(x,u,y) = E(x,u,y) \wedge E(y,u,x)$ This finishes the proof of the Theorem.

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 [Immerman and Kozen 1989; Vardi 1995]. This is indeed the case.

THEOREM 6.7.

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

PROOF. The containment of $\mathsf{TriAL}^=$ in FO^4 was shown in the proof of Proposition 5.4, and that $\mathsf{TriAL}^=$ contains FO^3 was already showed in the second part of the proof of Theorem **??**, since the translation used there does not make use of inequalities in joins.

That the containments are strict follows from the proof of Theorem **??**. □

Expressivity of the recursive algebra. 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 $\bar{v}$. The semantics is as follows. For an instance I and an assignment $\bar{c}$ for variables $\bar{z}$, 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 **??** 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 **??** for the case of TriAL and FO.

THEOREM 6.8.

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

PROOF. We split the proof into three parts, one for each of the claims.

Part 1. We begin by proving that TriAL^* is strictly contained in TrCl^6. To see that TriAL^* is contained in TrCl^6 we use induction on the structure of TriAL^* expressions. Note that all the cases, except for the Kleene closure of various joins we use, are precisely the same translation as in the proof of Theorem **??**. What remains to prove is that expressions of the

form

$$e' := (e \overset{i,j,k}{\underset{\theta,\eta}{\bowtie}})^*$$

can be translated into TrCl^6 expressions (the other join being completely symmetrical).

To see this, let $\psi_e(x,y,z)$ be a TrCl^6 formula equivalent to e. That is we have that $I_T \models \psi_e(a,b,c)$ if and only if $(a,b,c) \in R(T)$, for any triplestore T, with I_T the FO-structure representing T. We define the following formula $\psi_{e'}(x',y',z')$ in TrCl^6:

$$\psi_e(x',y',z') \vee \exists x,y,z\ \big(\psi_e(x,y,z) \wedge [\mathbf{trcl}_{x,y,z,x',y',z'}\varphi(x,y,z,x',y',z')](x,y,z,x',y',z')\big)$$

Where $\varphi(x,y,z,x',y',z')$ is a formula such that $\varphi(a,b,c,a',b',c')$ holds in I_T iff there exists a triple (a'',b'',c'') such that $\psi_e(a'',b'',c'')$ holds and the join of (a,b,c) and (a'',b'',c'') produces triple (a',b',c'). The definition of this formula in TrCl^6 is rather cumbersome, since it depends on the positions i,j,k of the join in question. We just give two examples, the rest are treated in the same way: For the expression $e' = (e \bowtie^{1,2,3'})^*$, we have that $\varphi(x,y,z,x',y',z')$ is $x = x' \wedge y = y' \wedge \exists x' \exists y' \big(\psi_e(x,y,z) \wedge \psi_e(x',y',z')\big)$. As another example, if $e' = (e \bowtie^{1',2',3'})^*$, then φ is just $\psi_e(x,y,z) \wedge \psi_e(x',y',z')$.

Next we prove that $\psi_{e'}$ is equivalent to expression e' over all triplestores. For one direction, let T be a triplestore database using a set O of objects, and assume that triple (a,b,c) belong to $e'(T)$. Then from the semantics of the recursive operator, there are sequences $t_1,\ldots,t_m$ of triples in O^3 and $p_1,\ldots,p_m$ of triples in $e(T)$ such that $t_1 \in e(T)$, and $t_{m+1} = t_m \overset{i,j,k}{\underset{\theta,\eta}{\bowtie}} p_m$. If $m = 1$ this follows from the first part of $\psi_{e'}$. If $m > 1$, notice that, by definition, $I_T \models \varphi(t_j, t_{j+1})$ for each $1 \leq j < m$. It follows that $I_T \models \psi_{e'}$. The other direction is analogous.

The fact that the containment is strict follows from Part 3 of the proof.

Part 2. Next we prove that TrCl^3 is contained in TriAL^*. We do this by induction on TrCl^3 formulas. Note that all the cases, except for the case of transitive closure operator, are exactly the same as in the proof of Theorem **??**. Next we show how to translate formulas of the form

$$\psi(x,y,z) := [\mathbf{trcl}_{x,y}\varphi(x,y,z)](u_1,u_2).$$

By the induction hypothesis there exists a TriAL^* expression R_φ such that for any triplestore T we have $I_T \models \varphi(a,b,c)$ iff $(a,b,c) \in R_\varphi(T)$.

Consider now the following expression R_ψ:

$$R := (R_\varphi \overset{1,2',3}{\underset{3=3' \wedge 2=1'}{\bowtie}})^*.$$

Observe now that a triple (a,b,c) will be contained in $R(T)$ iff there is a sequence of triples $(a,b_1,c),(b_1,b_2,c),(b_2,b_3,c),\ldots(b_k,b,c)$ with the property that they all belong to $R_\varphi(T)$. But this then means that the pair (a,b) belongs to the transitive closure of the relation defined by $\varphi(x,y,c)$. That is we have that $(a,b,c) \in R(T)$ iff b is reachable from a using only edges defined by $\varphi(x,y,c)$.

We now proceed case by case, depending on the structure of terms u_1 and u_2. Since our terms are only variables we have a total of nine cases.

— If $u_1 = x$ and $u_2 = y$ we define $R_\psi := R$. It is straightforward to see that $(a,b,c) \in R_\psi(T)$ iff $I_T \models \psi(a,b,c)$.

— If $u_1 = y$ and $u_2 = x$ we define $R_\psi := R$.
— If $u_1 = x$ and $u_2 = z$ we define $R_\psi := \sigma_{2=3}R$.
— If $u_1 = z$ and $u_2 = x$ we define $R_\psi := \sigma_{1=3}R$.
— If $u_1 = x$ and $u_2 = x$ we define $R_\psi := \sigma_{1=2}R$.
— All of the other cases are symmetric.

This concludes the proof in the case when φ above uses x, y, z as variables. All of the other cases are similar, e.g. when we have the formula $[\mathbf{trcl}_{x,y}\varphi(x, y, x)](x, y)$ the expression $(\sigma_{1=3}R_\varphi \bowtie^{1,2',3}_{2=1'})^*$ in place of R will suffice (note that now we have only two free variables).

That the containment is strict follows from the comments at the beginning of the proof of Part 3 below.

Part 3. We begin by showing that TriAL^* is not contained in TrCl^4 or TrCl^5. In the proof of Theorem **??** we show that TriAL, and thus TriAL^* contain an expression e such that $e(T)$ is nonempty if and only if T has 6 different objects. The proof then follows by two classical results in finite model theory [Libkin 2004]: (1) e cannot be expressed by neither $\mathcal{L}^4_{\infty\omega}$ not $\mathcal{L}^5_{\infty\omega}$, the infinitary logic restricted to 4 and 5 variables, respectively, and (2) TrCl^k is contained in $\mathcal{L}^k_{\infty\omega}$

To see that TrCl^4 is not contained in TriAL (and thus that neither TrCl^5 not TrCl^6 are contained in TriAL), we define an analog of the logic FO^3-join used in the proof of Theorem **??**. The logic FO$^3_\infty$-join extends FO^3-join with countably infinite disjunctions and conjunctions of formulas in FO^3-join (of course the restriction on the variables still holds). Formally, every FO^3-join formula is in FO$^3_\infty$-join, and if all φ_i are formulas in FO$^3_\infty$-join using the same set of at most 3 variables, for $i \in S$, where S is not necessarily finite, then $\bigwedge_{i\in S}\varphi_i$ and $\bigvee_{i\in S}\varphi_i$ are formulas in FO$^3_\infty$-join.

Notice that, by using these disjunctions, it is trivial to express the recursive star operator of TriAL^* with FO$^3_\infty$-join. Thus, if two structures $\mathcal{A}$ and $\mathcal{B}$ are indistinguishable by FO$^3_\infty$-join, then so are they by TriAL^*.

On the other hand, using the techniques in [Libkin 2004] it is not difficult to see that, if two structures $\mathcal{A}$ and $\mathcal{B}$ are indistinguishable by FO$^3_\infty$-join iff they are indistinguishable by FO^3-join (if the spoiler can win the join game on $\mathcal{A}$ and $\mathcal{B}$, then it can win the infinitary join game that characterizes FO$^3_\infty$-join).

It follows from the above observations, and the proof of Theorem **??**, that TriAL^* cannot express the query

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

where

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

used in the proof of Theorem **??**. □

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, and compare it in terms of expressiveness with several well established graph database query languages such as NREs, RPQs and *conjunctive* regular path queries (CRPQs). As our yardstick language for comparison we use a recently proposed version of XPath, adapted for graph querying [Libkin et al. 2013] which subsumes both NREs and RPQs. Its navigational fragment, presented next, is essentially Propositional Dynamic Logic (PDL) [Harel et al.

2000] with negation on paths. 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 that we use 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.

Path formulae of GXPath are given below. Here a ranges over a finite 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 an edge labelled graph $G = (V, E)$ over the labelling alphabet Σ, $\top$ returns V, and $\langle\alpha\rangle$ returns all $v \in V$ 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 (with respect to V^2), and transitive closure. Finally $[\varphi]$ denotes the set of pairs (v, v) such 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 as follows. First, let $\texttt{no\_edge}$ be a new label not appearing in Σ and define $\Sigma' = \Sigma \cup \{\texttt{no\_edge}\}$. Next, let $E' = E \cup \{(v, \texttt{no\_edge}, v') | (v, v') \notin \pi_{1,3}(E)\}$, where $\pi_{1,3}$ denotes the projecting the first and the third element of each triple in $E \subseteq V \times \Sigma \times V$. Finally, we define $T_G = (O, E')$, with $O = V \cup \Sigma'$. Note that the extra label is used in order allow each element of the triplestore domain to appear in the relation E, as per Definition A.1. This does not cause any issues, since the extended edge relation can be computed in polynomial time.

When dealing with the triplestore representation T_G of a graph database G we will often use relation E that contains only the edges of the original graph G. In order to avoid confusing where the relation is coming from, when working inside T_G we will use the notation E_G for E. Note that E_G can be defined as $E' \bowtie^{1,2,3}_{2 \neq \texttt{no\_edge}} E'$. Similarly, instead of working with the universal relation over T_G which contains all the triples of elements in O, we will make use of the relation $U_G = \{(s, p, o) | s, p, o \in V\}$ which contains all the triples of elements of V. The relation U_G is easily defined as $U \bowtie^{1,2,3}_{\varphi} U$, with φ containing conjuncts $1 \neq a \wedge 2 \neq a \wedge 3 \neq a$, for each $a \in \Sigma'$ and U being the universal relation over O as defined in Section 3.

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 such that for every graph database G we get the same answer when applying e_α to G as when applying $\pi_{1,3}(e_\alpha)$ to T_G (we say that the two queries are *equivalent over graph databases*). 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 $\pi_{1,3}(e)$ over graph databases. 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 2.2. 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.

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 6.9. *GXPath is strictly contained in* TriAL*.

PROOF. Assume that GXPath uses a finite alphabet Σ of labels. We show that GXPath is contained in TriAL* by simultaneous induction on the structure of GXPath expressions. If we are dealing with a path expression α we will denote the TriAL* expression equivalent to α by E_α. Similarly when dealing with a node expression φ, the corresponding TriAL* expression will be denoted E_φ. Note that for the node expression φ of GXPath we consider the TriAL* expression E_φ to be its equivalent if the answer set of φ is the same as the answer of $\pi_1(E_\varphi)$ over all graph databases and their triplestore representations, respectively.

Through the proof we will make use of the relation U_G defined above. We will also make use of the diagonal relation $D = U_G \bowtie^{1,1,1}_{1=1} U_G$ selecting all the triples (a, a, a) with $a \in V$.

Basis:

— $\alpha = a$ then $E_\alpha = E_G \bowtie^{1,1,3}_{2=a} E_G$
— $\alpha = a^-$ then $E_\alpha = E_G \bowtie^{3,3,1}_{2=a} E_G$
— $\alpha = \varepsilon$ then $E_\alpha = U_G \bowtie^{1,1,1}_{1=1} U_G$
— $\varphi = \top$ then $E_\varphi = U_G \bowtie^{1,1,1}_{1=1} U_G$

Inductive step:

— $\alpha' = [\varphi]$ then $E_{\alpha'} = E_\varphi \bowtie^{1,1,1}_{1=1} E_\varphi$
— $\alpha' = \alpha \cdot \beta$ then $E_{\alpha'} = E_\alpha \bowtie^{1,1,3'}_{3=1'} E_\beta$
— $\alpha' = \alpha \cup \beta$ then $E_{\alpha'} = E_\alpha \cup E_\beta$
— $\alpha' = \alpha^*$ then $E_{\alpha'} = (E_\alpha \bowtie^{1,1,3'}_{3=1'})^*$
— $\alpha' = \overline{\alpha}$ then $E_{\alpha'} = (U_G - E_\alpha) \bowtie^{1,1,3}_{1=2} U_G$
— $\varphi' = \neg\varphi$ then $E_{\varphi'} = D - E_\varphi$
— $\varphi' = \varphi \wedge \psi$ then $E_{\varphi'} = E_\varphi \cap E_\psi$
— $\varphi' = \langle\alpha\rangle$ then $E_{\varphi'} = E_\alpha \bowtie^{1,1,1}_{1=1} E_\alpha$.

In order to make complementation easier we make the answer set of all of our expressions consist only of elements of V. Furthermore, for path formulas the first element of the answer is repeated in the triple, while for node formulas the triple has the answer node repeated three times. It is straightforward to check that this translation works as intended. For illustration, consider the case when $\alpha' = \alpha \cdot \beta$. Our induction hypothesis is that we have two expressions, E_α and E_β such that (a, b) is in the answer to α on G iff $(a, a, b) \in E_\alpha(T_G)$ and similarly for β. Assume now that (a, b) is in the answer to α' on G. Then there is c such that (a, c) is in the answer to α and (c, b) in the answer to β. But then $(a, a, c) \in E_\alpha(T_G)$ and $(c, c, b) \in E_\beta(T_G)$. By the definition of join, we conclude that $(a, a, b) \in E_{\alpha'}(T_G)$. Note that all the implications above were in fact equivalences, so we get the opposite direction as well. All of the other cases follow similarly.

To show that the containment is strict we use the fact that GXPath is contained in $\mathcal{L}^3_{\infty,\omega}$ [Vrgoč 2014]. Consider now the following TriAL expression:

$$U_G \overset{1,2,3}{\underset{\varphi}{\bowtie}} U_G,$$

where $\varphi = (1 \neq 2) \wedge (1 \neq 3) \wedge (1 \neq 1') \wedge (2 \neq 3) \wedge (2 \neq 1') \wedge (3 \neq 1')$. It follows easily that this expression has an nonempty answer set if and only if the original graph database had at least four different nodes. It is well known that this query is not expressible in $\mathcal{L}^3_{\infty,\omega}$, thus implying that the containment is indeed strict. □

Note that this also implies a strict containment of languages presented in [Fletcher et al. 2011; 2012] 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 6.10.

— *NREs are strictly contained in* TriAL*.
— *RPQs are strictly contained in* TriAL*.

Noting that SPARQL property paths are just a syntactic variant o two-way RPQs[4] [Kostylev et al. 2015], Theorem 6.9 also gives us the following:

Corollary 6.11.

— *SPARQL property paths are strictly contained in* TriAL*.

Next we move to comparison with conjunctive queries. Here, instead of usual CRPQs we will consider slightly more expressive conjunctive NREs (CNREs) [Barceló et al. 2013]. Formally, these are expressions of the form $\varphi(\overline{x}) = \exists\overline{y} \bigwedge_{i=1}^{n}(x_i \xrightarrow{e_i} y_i)$, 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 pattern 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.

In order to do these comparisons we will rely on the fact that TriAL* is subsumed by infinitary logic with six variables.

Lemma 6.12. TriAL* *is contained in the infinitary logic* $\mathcal{L}^6_{\infty,\omega}$.

Proof. REPHRASE THIS PARAGRAPH ONCE THE REFERENCES SETTLE; THIS IS NOW LEMMA 6.4? What we mean by this is along the lines of the proof of Proposition 6.1, where we compare TriAL with FO^6 over the vocabulary $(E_1, \ldots, E_l, \{o \in \mathcal{O}\})$. As before we always interpret the constants $o \in \mathcal{O}$ as themselves, but deploy the active domain semantics, which considers only the objects appearing in the triplestore relations $E_1, \ldots E_n$ to form the operational domain for our model.

That is to prove the lemma, we only have to show that the * operator can be simulated in this logic. To see this consider an arbitrary star-join of the form

$$R' = (R \overset{i,j,k}{\underset{\theta}{\bowtie}})^*.$$

Assume that we have an $\mathcal{L}^6_{\infty,\omega}$ formula $F_R(x_i, x_j, x_k)$ such that $T \models F_R(a, b, c)$ if and only if $(a, b, c) \in R(T)$ (we obtain this from Proposition 6.1, possibly renaming the variables). We first define a formula α based on θ. We let α be the conjunctions of formulas $x_i = x_j$, whenever $i = j$ is a conjunct in θ and $x_i \neq x_j$, whenever $i \neq j$ is a conjunct in θ. Constants are treated analogously, e.g. a comparison of the form $2 = a$ would be handled by adding the clause $x_2 = a$.

We now define the following formulas:

— $R_1(x_i, x_j, x_k) := F_R(x_i, x_j, x_k)$
— $R_{n+1}(x_i, x_j, x_k) := \exists x_u, x_v, x_w(R_n(x_1, x_2, x_3) \wedge \alpha \wedge \exists x_i, x_j, x_k(x_i = x_{1'} \wedge x_j = x_{2'} \wedge x_k = x_{3'} \wedge F_R(x'_1, x'_2, x'_3))$

[4]There are some subtle differences with respect to negated property sets though, however, these can be easily expressed using TriAL*, since we allow comparison with an arbitrary constant.

Here we have $\{i, j, k, u, v, w\} = \{1, 1', 2, 2', 3, 3'\}$.
Finally set $F_{R'}(x_i, x_j, x_k) := \bigvee_{n\in\omega} R_n(x_i, x_j, x_k)$.
It is straightforward to check that this formula defines the desired relation over T. A similar formula can be defined for left-joins. □

When comparing TriAL* with CNREs we obtain the following.

THEOREM 6.13.

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

PROOF. We begin by proving that full CNREs and TriAL* are incomparable in terms of expressive power.

The existence of a CNRE query not expressible by TriAL* simply follows from the fact that TriAL* is contained in $\mathcal{L}^6_{\infty,\omega}$. The reason for this is that CNREs can ask for a 7-clique, a property not expressible in $\mathcal{L}^6_{\infty,\omega}$.

To see the reverse we will use a well know fact that CNREs are a monotonic class of queries. That is for any two graph databases G and G' such that $G \subseteq G'$ (that is G' contains all the nodes and edges of G) and any CNRE q we have that (u, v) is in the answer to q on G implies that (u, v) is in the answer to q on G' as well.

Next consider the following TriAL expressions

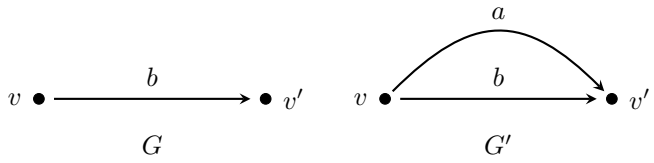
$$E_a := E_G \overset{1,2,3}{\underset{2=a}{\bowtie}} E_G \qquad E_{aux} := E_G \overset{1,2,3}{\underset{1=1',3=3'}{\bowtie}} E_a.$$

The expression E_a, when evaluated over T_G (for some graph database $G = (V, E)$), returns all triples (v, a, v') such that $(v, a, v') \in E$. On the other hand E_{aux} finds all $(v, b, v') \in E$, for some $b \in \Sigma$, such that (v, a, v') also belongs to E. We now define

$$e := E' - (E_a \cup E_{aux}).$$

It is straightforward to see that, when interpreted over T_G, this expression returns all pairs of nodes that are *not* connected by an a-labelled edge. (Formally we will return all the triples (v, b, v') such that v and v' are not connected by an a-labelled edge in G and b is either from Σ, in which case $(v, b, v') \in E$, or $b =$ `no_edge`.) Suppose now that there is a CNRE q defining the aforementioned query.

Consider the following two graphs.



The nodes (v, v') will be in the answer to our query over the graph G. Using the monotonicity of CNREs and the fact that G is contained in G' we conclude that (v, v') is also in the answer to our query over G'. Note that this is a contradiction since we assumed that q extracts all pairs of nodes not connected by an a-labeled path.

This concludes the proof of part one of our Theorem.

Next we show that UCNREs using only three distinct variables are contained in TriAL*. Observe first that for any NRE e there is a TriAL* expression E_e equivalent to e over all data graphs (Corollary 6.10). We will now show that any CNRE that uses precisely three variables is definable using TriAL*. To see this, consider the following example. Let Q be the following CNRE:

$$Q(x, y, z) := (x, e_1, y) \wedge (z, e_2, y) \wedge (y, e_3, y) \wedge (y, e_4, x).$$

Assume now that for each NRE e_i we construct an equivalent TriAL* expression T_{e_i} as in the proof of Theorem 6.9. In particular this means that (a, b) belongs to the answer of e_i over $G = (V, E)$ if and only if (a, a, b) belongs to the answer of T_{e_i} over T_G. In the expression equivalent to Q we will keep the variables x, y and z in that precise order in our triples; that is, x will appear only in the first place, y in the second and z in the third. Using this convention for each conjunct we define a TriAL* expression that will keep the variables used in this conjunct in the correct place, while the other values in the triple are arbitrary nodes from V. In particular, for (x, e_1, y) we use $T_1 = T_{e_1} \bowtie^{1,3,1'}_{1=1} U_G$. Note that the evaluation of T_1 will contain all the triples (a, b, o), where (a, b) belong to the answer of e_1 over G and $o \in V$. Similarly we define $T_2 = T_{e_2} \bowtie^{1',3,1}_{1=1} U_G$ for the conjunct (z, e_2, y). In the case of (y, e_3, y) we use $T_3 = T_{e_3} \bowtie^{1',1,2'}_{1=3} U_G$, which now contains all the triples (o, a, o') with (o, o) in the answer of e_3. Lastly, to cover the conjunct (y, e_4, x) we use $T_4 = T_{e_4} \bowtie^{3,1,1'}_{1=1} U_G$. We can now define $T_Q = T_1 \cap T_2 \cap T_3 \cap T_4$. It is easy to check that T_Q and Q are equivalent.

Extending this construction to the most general case of an arbitrary number of conjuncts with various arrangement of variables is straightforward.

Finally, since TriAL expressions are closed under union we get that UCNREs with only three variables are contained in TriAL*. That the containment is proper follows from the first part of the proof. □

One of the most fundamental classes of queries over graph databases appearing in the literature are conjunctive regular path queries, or CRPQs [Consens and Mendelzon 1990; Calvanese et al. 2003]. These can be seen as conjunctive NREs that do not use the nesting operator in their expressions. By observing that the expressions separating CNREs from TriAL* are CRPQs, and that CNREs are more expressive than CRPQs and C2RPQS [Barceló et al. 2012] we obtain:

COROLLARY 6.14.

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

Similarly, one can reason that full SPARQL is not comparable in terms of expressive power with TriAL*, since it allows an arbitrary number of variables. On the other hand, it is also easy to see that SELECT-FROM-WHERE SPARQL queries with property paths which use only three variables are strictly contained in TriAL*.

7. TRIAL IN PRACTICE

Thus far we have demonstrated that TriAL and TriAL* have good theoretical properties and are able to express a wide range of navigational queries over triplestores. However, this does still not guarantee that having such expressive languages is feasible in practice. Therefore, in this section we describe an implementation of TriAL* on top of an existing relational system and test its efficiency over real world and synthetic RDF data. To do so, we check how TriAL* copes with computationally expensive queries introduced in our examples, and also compare its performance on property path queries with two popular SPARQL engines.

We would like to state up front that our implementation is meant to serve as a proof of concept, and not as a standalone system. The main reason for this is the fact that the main focus of this paper is not the development of a working tool, but describing a plausible conceptual framework for expressing navigational queries over RDF triplestores. It is also for this reason that we opted to implement TriAL* on top of an existing system, and not provide

an independent implementation, as this would require us to deal with aspects outside of the scope of this work (e.g. data storage and retrieval, access methods, etc.).

To allow for reproducibility of the experiments we have made our implementation, together with a brief user guide, the queries used for testing, and the scripts used to generate the synthetic data available at an anonymous DropBox folder at https://bla.bla.bla.

Implementing TriAL*

Our implementation is designed to work on top of any relational database system which supports the `WITH RECURSIVE` operator. The main idea behind our implementation is to build a query tree for a TriAL* expression, which then uses the relational database as a "black-box" for evaluating joins and the Kleene star (it is for this reason that we use systems supporting recursion). In our experiments we used Postgre SQL v.9.5.3., so we will consider this engine as our reference point; however, the user is free to chose a relational database as desired. The triplestore is considered to be stored as one big ternary table.

The implementation consists of two modules. The first module is a parser which takes a TriAL* expression[5] and creates a query tree consisting of standard SQL commands. Here, all of the standard joins are rewritten as `SELECT-FROM-WHERE` SQL queries, where we assume that all selections (i.e. the operators $\sigma_\theta(e)$) are pushed into the joins. As far as the left and the right Kleene closure of a ternary relation is concerned, this is rewritten into a `WITH RECURSIVE` query which takes into account if the joins should be evaluated from the left or from the right. Once the query tree is constructed, it is passed to the second module, which connects to the database, and upon parsing the tree top-down, begins to evaluate the expression by executing SQL commands (therefore the evaluation is done bottom-up and needs to materialize all of the intermediate results). Once the root of the tree is evaluated, the result is returned to the user[6].

Although this type of implementation fully supports the algebra presented in the paper, one might be concerned about the efficiency of recursive operations, as these are the computationally most expensive part of our evaluation. Because of this we also extend the language in order to support specifying the starting point of the reachability pattern we want to compute, as is often the case in practice. Consider for example a pattern such as $\mathsf{Reach}_{\rightarrow}$ from the Introduction. Instead of knowing all possible pairs of nodes connected by such reachability pattern, in practice one is often only interested in finding all nodes reachable from one particular starting point (i.e. the leftmost point in the graphical depiction of that pattern in the Introduction). Indeed, when considering navigational RDF queries in practice, this is almost always the case [Gubichev et al. 2013; Reutter et al. 2015], and in fact, many RDF engines such as e.g. Virtuoso, only consider path queries which have an explicit starting point. Furthermore, specifying a starting point is equivalent to providing a base for the linear recursion in the `WITH RECURSIVE` operator, and is known to significantly speed up the query evaluation (in fact, we will soon see that the savings can be almost an order of magnitude). For these reasons, we include recursive queries which can explicitly specify their starting point in our implementation.

Next, we move onto empirical evaluation of our implementation which is meant to showcase that TriAL* queries have the potential to be used in practice. We divide our experiments into two parts. First, we consider real world RDF data from YAGO [Suchanek et al. 2007], a huge knowledge base containing information from Wikipedia, WordNet and GeoNames. Here we design several property path queries and nested regular expressions which test various aspects of navigational querying over this dataset. In the second part we take the three

[5]For the implementation we provide a keyboard friendly syntax resembling SQL's `SELECT-FROM-WHERE` commands. More details can be found at the repository https://myanonurl.com.

[6]We would like to note that since our experiments do not use the difference operator, we currently do not support it in the implementation. However, adding it is simply a matter of extending the parser.

TriAL* specific queries introduced in this paper: $\mathsf{Reach}_{\rightarrow}$, $\mathsf{Reach}_{\nearrow}$ and the query reachTA from Example 3.3, and design several synthetic datasets intended to push their performance, and see how they scale.

Real world RDF data and a comparison with SPARQL engines

Here we test how our implementation copes with real world RDF data and queries which are used in practice. We also compare the evaluation times of TriAL* with that of two popular SPARQL query engines, and test some queries which lie outside the scope of SPARQL or even nested regular expressions.

The dataset. As our dataset we use YAGO [Suchanek et al. 2007], a huge RDF database which contains information extracted from Wikipedia, WordNet and GeoNames. More precisely, we use a piece of YAGO known as YAGOFacts[7], which contains the core information available in the YAGO database. The dataset contains information about famous people, movies, geographical entities, etc. and shows how such entities are connected. For instance, some triples state that a particular actor acted in some movie (e.g. `(Kevin Bacon, actsIn, Unforgiven)`), others that a city is located in a particular region or country (e.g. `(Berlin, isLocatedIn, Germany)`), etc. When loaded into Postgre SQL, the dataset was of size 2.4GB and it contains 5.6 million triples.

Queries. For testing our implementation we selected five navigational queries which appeared in previous literature on RDF querying [Gubichev et al. 2013; Reutter et al. 2015]. We would like to stress here that we only evaluate navigational queries (i.e. the ones using Kleene star *). Doing a comparison with purely relational queries (such as the ones available in TriAL) would only amount to testing the performance of Postgre SQL (and comparing it to SPARQL engines), which was previously done in e.g. [Angles et al. 2013; Hernández et al. 2016]. We therefore focus solely on queries which use non trivial reachability patterns under the Kleene star. The queries are as follows:

(1) **Q1:** This query finds all actors which have a finite Bacon number[8]. On can view this as needing to navigate from Kevin Bacon to a movie he starred in (i.e. traversing a triple `(Kevin Bacon, actedIn, movieX)`), obtaining the actors who starred in this movie (that is, examining all other triples `(actorY, actedIn, movieX)`), and remembering the triple `(Kevin Bacon, actedIn, actorY)`. The procedure is then repeated starting from `actorY` until no further answers are found. Formally, we return the triple `(Kevin Bacon, actedIn, actor)`, where `actor` has a finite Bacon number.
(2) **Q2:** This query retrieves all types of geographic entities that have something to do with Berlin, or some other entity that Berlin is a part of. Here we start from a triple `(Berlin, isLocatedIn, X)`, and perform a join with triples of the form `(X, isLocatedIn, Y)`, remembering `(Berlin, isLocatedIn, Y)`. The process is then repeated, but now starting in `Y`. Once this recursive query is completed, we do a join with triples of the form `(Y, dealsWith, Z)`.
(3) **Q3:** Here we look for all the people who are married to a person who owns a property which is located in the United Sates (here the "located in" part is taken transitively). The query is similar to Q2.
(4) **Q4:** In this query we return all people with a finite Bacon number, but such that in the witnessing path the director of the movie is also an actor (not necessarily in the same movie).

[7]See http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/downloads/ for more details.
[8]A person has Bacon number one if she or he co-starred with Kevin Bacon in the same movie. A person has Bacon number $n+1$ if she co-starred in the same movie with someone with a Bacon number n.

	Q1	Q2	Q3	Q4	Q5
TriAL*	60	0.025	739 (5)†	136	265
Virtuoso	M	0.015	3.5	N/S	N/S
Jena	M	0.9	T	N/S	N/S

Table I: Execution times for queries Q1 through Q5 over the YAGO dataset. The abbreviation N/S stands for not supported, M for a memory error, and T for timeout. All times are in seconds. Here † stands for the running time of the optimised version of the query.

(5) **Q5:** In this query we test a pattern similar to the one from Example 3.3. Namely, we look for all people with a finite Bacon number, but such that the connection is made through movies which all have the same director.

Please note that the queries Q1, Q2 and Q3 are in fact property path queries, and are therefore supported by the current SPARQL standard. The query Q4 is a nested regular expression, while Q5 is only expressible using TriAL*. The queries Q2 and Q3 were taken from [Gubichev et al. 2013], while Q1, Q4 and Q5 come from [Reutter et al. 2015].

Another interesting observation is that all of these queries do have an implicit anchor where the evaluation can start: in the case of Q3 this is the final element we are trying to reach (United States), while in the other queries this is the first element (Kevin Bacon, or Berlin). When thinking of RDF as a format for data on the Web this actually makes a lot of sense, because we do not want to search the entire Web graph, but we wish to start the search from some fixed location (e.g. Kevin Bacon, Berlin, etc.).

Experimental setup. The queries were executed three times and the average running time is reported. In the case of queries Q1, Q2 and Q3, we also ran them over the same dataset using two popular SPARQl engines: Open Link Virtuoso[9] version 7.2.1.3214-pthreads (open source edition) and Apache Jena[10] version 3.1.0. We would like to note that due to the internal storage mechanisms, the YAGOFacts dataset weighs only 1.1GB in both Virtuoso and Jena. Since queries Q4 and Q5 are not supported in SPARQL we ran them only on our implementation of TriAL*. All of the experiments were ran using a MacBook Pro with 8GB RAM and an Intel Core i5 2.6GHz processor running OS X El Capitan.

Results and discussion. We present the execution times of our queries in Table I. As we can see, our implementation of TriAL* manages to compute all the queries in a reasonable time. One can also notice the stark difference between Q2 and Q3. Although these queries are similar in structure, the execution times are very different. Upon analysing the actual computation one can see that this is due to the size of the base relation used in the recursive part of the query. In the case of Q2 we only want the triples stating that Berlin is located in some entity, which is a relatively small set of triples, while for Q3 we want all entities located in the United States, which is a very large set of triples that is then being used to perform joins repeatedly. However, one can further optimise Q3 by allowing arbitrary queries as the base relation in the recursive part (this is roughly equivalent to having a sub query as a base of linear recursion in SQL). Namely, if we start computing Q3 by taking as the base only people married to someone who owns a property (and the location of the property), and then compute the transitive closure of "located in" using this relation as a base (and checking that we reach United States), we get an execution time of only 5 seconds. This shows us that simply specifying the starting point is sometimes not the best speed-up technique for TriAL* queries, and that proper query planning (with estimating the size of intermediate results), seems to hold a lot of promise for optimising navigational queries in

[9]http://virtuoso.openlinksw.com/
[10]https://jena.apache.org/

practice. As the main focus of this paper is conceptual, rather than practical, we plan to address this issue in future work.

Comparing the execution times to SPARQL engines, one can see that TriAL* shows much more stable performance. In particular, for the query Q1 which requires a large part of the RDF dataset to be traversed repeatedly, it is the only engine capable of executing the query. On the other hand, in the case of Q2, Virtuoso shows slightly better performance than TriAL*, while Jena is slower. The query Q3 reveals the type of features Virtuoso is optimised to work with, as it recognizes automatically that the query has a goal to reach, and therefore executes it from this end. When the query is appropriately optimised in TriAL* we again see similar execution times. It is worthwhile noting that Virtuoso executes navigational queries in main memory, so it is prone to run into memory limits quite fast for more complex questions, while TriAL* runs all the computations on disk, thus making it more reliable. Note that the queries Q4 and Q5 are not supported in current SPARQL engines, so we could not test against them.

Overall, we can see that our implementation of TriAL* shows good performance over real world data, and is comparable with current state of the art systems when it comes to navigational queries over RDF, while at the same time being capable of expressing many more queries. As the results show, there is a lot of room for improvement when it comes to optimising query execution, but overall, TriAL* seems to cope well with property paths.

Synthetic data and TriAL* queries

The main objective of this subsection is to demonstrate how the navigational queries presented throughout the paper perform on triplestores of realistic size, and how the simple optimisation of specifying the starting point can speed-up the evaluation by an order of magnitude for these queries.

Queries. We take three queries illustrating the types of navigational patterns which can occur in RDF: Reach$_{\rightarrow}$ and Reach$_{\nearrow}$ presented in the Introduction and reachTA (the TriAL* expression specifying each one of these queries can be found in Example 3.3).

Datasets. The RDF datasets used for testing these queries were generated in such a way that the number of patterns which get returned as the query answer for each of the queries forms around 2% of the total data. For each of the three queries we created three triplestores of sizes 500MB, 1.2GB and 1.7GB respectively. The number of triples in these datasets was around 7 million, 15 million, and 21 million, respectively. For the query Reach$_{\rightarrow}$ each datasets contained paths of length up to 20, with the number of such paths being 7, 15 and 20 thousand, respectively. Analogously, for the query Reach$_{\nearrow}$ each datasets contained patterns of height up to 20, with the number of such patterns being 7, 15 and 20 thousand, respectively. Finally, the number of patterns for the query reachTA was set at 2.4, 5 and 7.5 thousands respectively, with the horizontal length being 20, and the height of the pattern 3. A summary is available in Table II, columns 1 through 5.

Experimental setup. Each query was ran against the appropriate datasets in two modes. First, we ran the unrestricted version of the query as specified in Example 3.3. Next, we fixed the starting point of the query (i.e. the leftmost point in the graphical representation of each query) and tested the running times with this modification. All of the experiments were ran using a MacBook Pro with 8GB RAM and an Intel Core i5 2.6GHz processor running OS X El Capitan. The timeout for the queries was set at 4 hours.

Results and discussion. The evaluation results are presented in Table II. As we can see, when the unrestricted version of the query is ran over larger datasets one can run into some issues. In particular, the queries Reach$_{\rightarrow}$ and Reach$_{\nearrow}$ time out on the largest dataset, although they perform reasonably well over the smaller ones. On the other hand, the computationally more expensive query reachTA times out on D8 and D9, since it is based on nested recursion, which requires computing joins with the entire database multiple times.

	dataset	size	triples	patterns	time-u (sec)	time-sp (sec)
$\mathsf{Reach}_{\rightarrow}$	D1	572MB	7.14mil	7000	181	43
	D2	1.2GB	15.3mil	15000	1061	96
	D3	1.7GB	21.4mil	20000	X	131
$\mathsf{Reach}_{\nearrow}$	D4	565MB	7.14mil	7000	228	43
	D5	1.2GB	15.3mil	15000	795	93
	D6	1.7GB	21.4mil	20000	X	131
reachTA	D7	567MB	7.19mil	2400	7097	564
	D8	1.2GB	15.4mil	5000	X	13354
	D9	1.7GB	21.6mil	7500	X	X

Table II: Datasets and running times for TriAL^* queries over synthetic data. All patterns have maximal length of 20, and the height of the pattern in datsest D7, D8 and D9 is 3. The abbreviation time-u stands for the running time of the unrestricted version of each query, and time-sp, for the versions which has the staring point specified. The symbol X marks an execution timeout.

It is important to note that all of the computation is done on disk and is not evaluated in main memory, and would eventually terminate if the result is really needed.

Although some of the results of the unrestricted version of the queries show that further improvements are needed, when we specify the starting point of a query (as discussed above, this is often the case one wants to consider in practice), the results are much faster. In particular, we witness almost an order of magnitude improvement in the running times, and in the case of $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$ all the runs execute efficiently. There are still some issues with reachTA on larger datasets, which suggests that in order to have a full scale implementation of TriAL^*, it might be better to build a stand alone system based on the algorithms from Section 5, rather than using the existing relational architecture.

Overall, the results here fall in line with the complexity analysis from Section 5, which shows that simple navigational patterns can be evaluated efficiently, but that the full language of TriAL^* might cause some issues when evaluating queries over large datasets. Furthermore, when we know the starting point of our reachability query, the evaluation times are really efficient if we do not nest the star operator.

Practical lessons

In conclusion, we can see that although some TriAL^* queries are intrinsically difficult to compute (as demonstrated by the theoretical part of the paper), it is still possible to execute many such queries over real world datasets. Indeed, our experimental results show that for queries used in e.g. RDF, this can often be the case, any that the queries can be answered within a reasonable time limit. Pushing the performance over synthetic data also shows that the "well-behaved" queries pose no significant evaluation problems when a starting point is known, although there are some queries which run slow (but given enough time do terminate). In particular, our results suggest that there is a lot of room for improvement when it comes to query planning, and we hope to address this issue in future work.

8. CONCLUSIONS AND FUTURE WORK

As the current approaches for extracting navigational patterns from RDF data are primarily based on graph query languages, in this paper we explore if this tells the whole story. In particular, we identify several types of reachability patterns supported by RDF which lie outside of the scope of graph-based approaches and discuss some fundamental issues when using graph languages for querying RDF: namely, that they do not allow the queries to

be composed, since the result of a graph query is no longer an RDF database. To remedy this issue, we propose a simple algebra which can be used for navigating RDF triples. The language we propose, called TriAL*, is designed with the RDF data model in mind; that is, it works with triples, and the result of each query in the language is a set of triples. We also provide a Datalog characterisation of TriAL*, thus making the language more user-friendly.

As our results show, the TriAL* algebra has good query evaluation properties: in particular, the answers to the queries can be computed in polynomial time. Next, we also compare TriAL* with other well established formalisms for querying navigational properties such as fragments of first order logic which support recursion, and also with established query languages for graph databases. Finally, we provide an implementation of our algebra on top of an existing open source relational database system and do extensive testing on real world RDF data. Here our empirical results show that the theoretical ideas we present indeed have the potential to be used in practice, since our non optimised implementation is competitive against popular RDF querying engines, while still being able to express many queries they do not support.

In future work we would like to see how TriAL* can be extended even further to support different scenarios such as e.g. data graphs [Libkin et al. 2016], or property graphs, which are the data model used in current graph database systems (see e.g. the popular Neo4j graph engine [Neo4j 2013]). We present some initial results on this in the on-line appendix in order to show that this is a viable direction for future research. Another important direction is also implementing TriAL* as a stand alone system using algorithms from Section 5. As we have seen in Section 7, although using a relation system as a base is viable, some issues do exist and a full scale implementation which includes data structures that support fast evaluation of navigational queries might be preferred.

REFERENCES

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

R. Angles. 2012. A Comparison of Current Graph Database Models. In *ICDE Workshops*. 171–177.

R. Angles and C. Gutierrez. 2008. Survey of graph database models. *Comput. Surveys* 40, 1 (2008).

Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. 2013. Benchmarking database systems for social network applications. 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*. 15.

K. Anyanwu and A.P. Sheth. 2003. ρ-Queries: enabling querying for semantic associations on the semantic web. In *12th International World Wide Web Conference (WWW)*. 690–699.

M. Arenas and J. Pérez. 2011. Querying semantic web data with SPARQL. In *PODS*. 305–316.

P. Barceló, J. Pérez, and J.L. Reutter. 2012. Relative Expressiveness of Nested Regular Expressions. In *AMW*. 180–195.

P. Barceló, J. Pérez, and J. L. Reutter. 2013. Schema Mappings and Data Exchange for Graph Databases. In *ICDT*.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. 2000. Containment of conjunctive regular path queries with inverse. In *7th International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 176–185.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. 2002. Rewriting of regular expressions and regular path queries. *J. Comput. System Sci.* 64, 3 (2002), 443–465.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. 2003. Reasoning on regular path queries. *ACM SIGMOD Record* 32, 4 (2003), 83–92.

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

I. Cruz, A.O. Mendelzon, and P. Wood. 1987. A graphical query language supporting recursion. In *ACM Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD)*. 323–330.

P. Cudré-Mauroux and S. Elnikety. 2011. Graph Data Management Systems for New Application Domains. *PVLDB* 4, 12 (2011), 1510–1511.

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

W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. 2011. Adding regular expressions to graph reachability and pattern queries. In *27th International Conference on Data Engineering (ICDE)*. 39–50.

G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. 2011. Relative expressive power of navigational querying on graphs. In *ICDT*. 197–207.

G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. 2012. The Impact of Transitive Closure on the Boolean Expressiveness of Navigational Query Languages on Graphs. In *FoIKS*. 124–143.

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

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

Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling kleene: fast property paths in RDF-3X. In *GRADES 2013*. 14.

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

S. Harris and A. Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. `http://www.w3.org/TR/sparql11-query/`. (March 2013).

Daniel Hernández, Aidan Hogan, Cristian Riveros, Carlos Rojas, and Enzo Zerega. 2016. Querying Wikidata: Comparing SPARQL, Relational and Graph Databases. In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*. 88–103.

N. Immerman and D. Kozen. 1989. Definability with Bounded Number of Bound Variables. *IANDC* 83, 2 (1989), 121–139.

M. Kaminski and N. Francez. 1994. Finite memory automata. *Theoretical Computer Science* 134, 2 (1994), 329–363.

Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. 3–18.

LDBC. 2015. LDBC Task Force: Property Graphs Data Model. http://www.ldbccouncil.org. (2015).

L. Libkin. 2004. *Elements of Finite Model Theory*. Springer.

L. Libkin, W. Martens, and D. Vrgoč. 2013. Querying Graph Databases with XPath. In *ICDT*.

Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14.

L. Libkin and D. Vrgoč. 2012. Regular Path Queries on Graphs with Data. In *ICDT*. 74–85.

K. Losemann and W. Martens. 2012. The complexity of evaluating path expressions in SPARQL. In *PODS*. 101–112.

M. Marx. 2005. Conditional XPath. *ACM Trans. Database Syst.* 30, 4 (2005), 929–959.

Neo4j 2013. Neo4j, The graph database. `http://www.neo4j.org/`. (2013).

J. Pérez, M. Arenas, and C. Gutierrez. 2010. nSPARQL: A navigational language for RDF. *Journal of Web Semantics* 8, 4 (2010), 255–270.

Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. 2015. Recursion in SPARQL. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*. 19–35.

Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: A Core of Semantic Knowledge. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*. 697–706.

M. Y. Vardi. 1995. On the Complexity of Bounded-Variable Queries. In *PODS*. 266–276.

D. Vrgoč. 2014. *Querying graphs with data*. Ph.D. Dissertation. School of Informatics, University of Edinburgh.

P.T. Wood. 2012. Query Languages for Graph Databases. *Sigmod Record* 41, 1 (2012), 50–60.

Online Appendix to: TriAL: A navigational algebra for RDF triplestores

A. TRIAL FOR PROPERTY GRAPHS

In this appendix we show how the TriAL algebra can be extended to work over more expressive models underlying modern commercial graph database engines. We begin by introducing the model of data graphs which are a theoretical precursor to property graphs, and explaining what property graphs are. We then extend the model of triplestores to cover both these models, as well as extending our algebra in order to fully exploit the data they contain. Finally, we discuss evaluation issues and connections with logic and established data graph languages.

Data graphs and property graphs

Note that by the definition from Section 2, graph databases capture only the navigational aspect of the graph data model and do not permit us to store attribute values. To remedy this issue, the model of *data graphs* has been introduced in e.g. [Libkin and Vrgoč 2012], allowing to store data values coming from a potentially infinite alphabet in graph nodes. Formally, for a finite labelling alphabet Σ and a countably infinite set of data values $\mathcal{D}$, a data graph is defined as a triple (V, E, ρ), where (V, E) is a graph database over Σ, and $\rho : V \to \mathcal{D}$ is a function assigning a data value from $\mathcal{D}$ to each node. For instance, in the graph database from Figure 1, we could use the function ρ to assign the age to each person, or some other value. Supporting multiple attributes is done by extending the range of the function ρ to $\mathcal{D}^k$, where k is the number of attributes that will be used.

Data graphs are a theoretical precursor to a formalisation of the actual data model used in practical graph database systems: *property graphs*. Indeed, property graphs abound in systems supporting the graph data model (see e.g. by the popular Neo4j graph engine [Neo4j 2013]), and have been recently standardized by a working group of the Linked Data Benchmark Council (LDBC) formed by members of academia and industry [LDBC 2015]. The main difference between data graph and property graphs is that the latter allow data values in the edges as well, and they also allow multiple data values per node/edge. We will give an example of a property graph latter in this section.

Triplestores with attribute values

Here we extend the notion of a triplestore to include RDF databases, graph databases, data graphs and property graphs at once. Let $\mathcal{O}$ be a countably infinite set of objects, and $\mathcal{D}$ be a countably infinite set of data values.

Definition A.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,
— $\rho : O \to \mathcal{D}$ is a function that assigns a data value to each object, and
— for each $o \in O$ there is $i \in \{1, \ldots, n\}$ and a triple $t \in E_i$ 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

DOI:http://dx.doi.org/10.1145/0000000.0000000

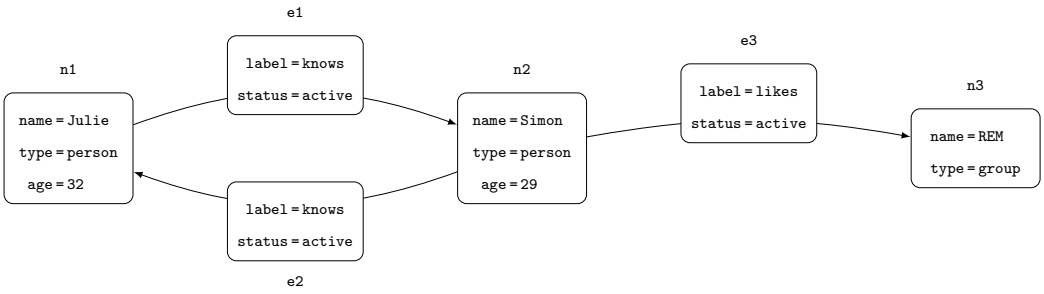


Fig. 5: A property graph storing Social Network data.

interested in actually appear in one of the relations. This assumption will also allow us to work with the active domain of our triplestore, thus enabling us to construct an algebra that is complete in terms of first order operations.

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 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.

The generality of this definition of triplestores allows us to cover more use scenarios than previously. In particular, the new model of triplestores subsumes the following:

(1) **RDF databases:** An RDF database fits into the triplestore model rather naturally: we simply use a single ternary relation E containing all the triples in the database.
(2) **Graph databases and data graphs:** A graph database $G = (V, E)$ can be seen as a triplestore where the set of objects are all the nodes and labels, while the set of edges E is used as the ternary relation. When dealing with a data graph $G = (V, E, \rho)$, the situation is analogous, but now we also include ρ in our triplestore (for completion we also define $\rho(a) = \perp$, for each label a appearing in G, and $\perp$ a designated data value).
(3) **Property graphs:** These are the model used in practical graph database systems. We illustrate how property graphs can be seen as triplestores in Example A.2.

Example A.2. In property graphs both the nodes and the edges are allowed to have attributes attached to them. To see why this might be useful consider again the graph database from Figure 1 presenting a part of a Social Network. Here we have the users and connections between them, but we can not specify any extra data, such as their age, their name, the time when the connection between two nodes was created, status of the connection, etc. To remedy this we use property graphs, and one such extension to our graph database is presented in Figure 5. Here we have three nodes and three edges in our property graph. The nodes are denoted `n1, n2, n3` and the edges `e1, e2, e3`. All of these objects then have attributes and their appropriate values. For instance, the node `n1` represents a person, the person's name is `Julie`, as denoted by the value of the attribute `name`. Similarly, the age of this person is `32`, and the type of the node (denoted by the attribute `type`) is `person`. Other nodes and edges have similar values.

To represent this property graph as a triplestore, we take the set of objects O to contain all the nodes `n1, n2, n3` and all the edges `e1, e2, e3`. Here each edge is identified with an object labelling it, and the endpoints of the edge are reflected in the triple relation defining our triplestore. In particular, for the property graph in Figure 5 the set of triples is: $\{(\texttt{n1}, \texttt{e1}, \texttt{n2}), (\texttt{n2}, \texttt{e2}, \texttt{n1}), (\texttt{n2}, \texttt{e3}, \texttt{n3})\}$, therefore telling us that e.g. `e1` has `n1` and `n2` as its endpoints.

To represent the attribute values our function ρ will assign to each object in O a quintuple $(name, type, age, label, status)$ of values. We use quintuples to represent data values and assume that each node will have null values for the last two attributes, while an edge will have nulls in the first three. Another way to go around this would be to have two different data value assignments to the object attributes, one for nodes and another for edges. To keep our language one sorted and compact we opt for the option presented here. Therefore in our example we will have that $\rho(\texttt{n1}) = (\texttt{Julie}, \texttt{person}, \texttt{32}, \perp, \perp)$, $\rho(\texttt{e1}) = (\perp, \perp, \perp, \texttt{likes}, \texttt{active})$, etc.

Extending TriAL to include data values

An extension of TriAL which also handles data values has to redefine the joins in order to make use of the attribute values defined by the function ρ. Formally, given two ternary relations R and R', *join* operations are now of the form

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

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 with data values. We now define the expressions of the *Triple Algebra with data values*, or d-TriAL for short. As before, d-TriAL 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 d-TriAL expression.
— If e is a d-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 d-TriAL expression.
— If e_1, e_2 are d-TriAL expressions, then the following are d-TriAL expressions:
  — $e_1 \cup e_2$;
  — $e_1 - e_2$;
  — $e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$, 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 as before. The semantics of the selection is defined in the same way as the semantics of the join: 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.

For the purposes of navigation we again define the *right* and *left Kleene closure* of any triple join $\bowtie^{i,j,k}_{\theta,\eta}$ over an expression e, denoted as $(e \bowtie^{i,j,k}_{\theta,\eta})^*$ for right, and $(\bowtie^{i,j,k}_{\theta,\eta} e)^*$ for left. The semantics are precisely the same as in the case of TriAL*. We refer to the resulting algebra as *Triple Algebra with data values and recursion* and denote it by d-TriAL*.

Example A.3. To see what type of properties the extended algebra can express consider again the property graph from Figure 5, but imagine now that we have many more people connected by `knows` links. In such a social network we might want to find all friends of friends of `Julie`; that is, people who she knows, or whom are known by another person she knows. This query can be easily expressed using TriAL* (without data values), but in some cases, we might want to also know Julie's friends-of-a-friend which are witnessed by `knows` relations which are currently active. For this we need to check that each link in the path connecting said person to `Julie` is active, for which we need to check the value of its status attribute: an operation not available in ordinary TriAL. However, in d-TriAL* we can express this query as follows:

$$(\sigma_{\rho(2)=\texttt{active}} E \bowtie^{1,2',3}_{1=3' \wedge \rho(2')=\texttt{active}})^*.$$

Datalog for d-TriAL

In order to deal with data values the relational vocabulary from Section 4 has to be extended with an additional binary symbol $\sim$, which holds true for two nodes n and n' if and only if $\rho(n) = \rho(n')$. Furthermore, since we want to allow comparing data values in the database with a constant data value, we also need to make our language two sorted, with the second sort having the set of all data values $\mathcal{D}$ as the domain. Since we are not doing any complementation with respect to this particular domain (we only work with the object present in the triplestore anyway), we can simply have the all data values present in the logical model of our triplestore and interpret constants as themselves. With this representation we can now define a fragment of Datalog capturing the triple algebra with data values.

Here we describe a Datalog fragment capturing d-TriAL. A $\mathsf{d-TripleDatalog}$ rule is of the form

$$S(\overline{x}) \ \leftarrow \ S_1(\overline{x_1}), S_2(\overline{x_2}), \sim(y_1, z_1), \ldots, \sim(y_n, z_n), u_1 = v_1, \ldots, u_m = v_m \tag{5}$$

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 objects in $\mathcal{O}$;
(4) y_is and z_is are either variables or data values in $\mathcal{D}$;
(5) all variables in $\overline{x}$ and all variables in u_j, v_j, y_j, z_j are contained in $\overline{x_1} \cup \overline{x_2}$.

The newly added predicate $\sim(y_i, z_i)$ compares the data values of the nodes y_i and z_i (or compares them to a constant data value), and can be seen as the equality relation over our second domain $\mathcal{D}$. As before, a $\mathsf{d-TripleDatalog}^\neg$ rule is like the rule (1) but all equalities and predicates, except the head predicate S, can appear negated. A $\mathsf{d-TripleDatalog}^\neg$ *program* Π is a finite set of $\mathsf{d-TripleDatalog}^\neg$ 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 $j \leq i$. An analogous proof to that of the Proposition 4.1 gives us the following.

PROPOSITION A.4. d-TriAL *is equivalent to nonrecursive* $\mathsf{d-TripleDatalog}^\neg$ *programs.*

To cover d-TriAL* we add the same type of recursion as with TriAL*. A $\mathsf{d-ReachTripleDatalog}^\neg$ *program* is a $\mathsf{d-TripleDatalog}^\neg$ program in which each recursive predicate S is the head of exactly two rules of the form:

$$\begin{array}{lcl} S(\overline{x}) & \leftarrow & R(\overline{x}) \\ S(\overline{x}) & \leftarrow & S(\bar{x}_1), R(\bar{x}_2), V(y_1, z_1), \ldots, V(y_k, z_k) \end{array} \tag{6}$$

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 3, or a recursive predicate defined by a rule of the

form 6 that appears before S. 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.

Again, the negation in $\mathsf{d}-\mathsf{ReachTripleDatalog}^\neg$ programs is *stratified* and the semantics is defined as before, but now taking into the account that we have data values as well. With this, we have the following d-TriAL^* capture result, whose proof is analogous to that of Theorem 4.2.

THEOREM A.5. *The expressive power of* d-TriAL^* *and* $\mathsf{d}-\mathsf{ReachTripleDatalog}^\neg$ *programs is the same.*

Complexity and comparison with logic

When it comes to complexity of evaluating TriAL with data values, there is no difference to the case where data values are not present. In particular, one can easily prove that:

THEOREM A.6. *The problem* QUERYCOMPUTATION *can be solved in time*

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

Similarly, when it comes to comparison with logic, one can again draw the same conclusions as in Section 6, but now using the variant of first order logic which includes data values (denoted here by $d-\mathrm{FO}$) in the same way that datalog programs of Subsection A. Namely, reusing the same techniques as in Section 6, we can prove the following:

THEOREM A.7.

— $d-\mathrm{FO}^3$ *is strictly contained in* d-TriAL.
— d-TriAL *is strictly contained in* $d-\mathrm{FO}^6$.
— d-TriAL *is incomparable with* $d-\mathrm{FO}^4$ *and* $d-\mathrm{FO}^5$*; and*
— *The same set of conclusions holds for* d-TriAL^* *and the transitive closure logic with data value comparisons.*

Comparison with languages for data graphs

Here we show how d-TriAL compares to languages for querying data graphs. Next we formalise how we treat data graphs as triplestores. Take any data graph $G = (V, E, \rho)$ over the alphabet Σ and with $\rho : V \to \mathcal{D}$ assigning a data value from an infinite domain $\mathcal{D}$ to each node of V. For this graph we define the corresponding triplestore $T_G = (O, E', \rho)$ over Σ', where Σ', the domain O and the relation E' are as in the case of graphs with no data values. We use the same function ρ to assign data values in G and T_G. 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 $\mathsf{GXPath}(=)$, presented first in [Libkin et al. 2013], is given by the following grammars for node and path formulae:

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

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

The interpretation of standard operators is as in Section 6.2, and the semantics of the equality 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, respectively, and $\rho(v_\alpha)\ \theta\ \rho(v_\beta)$. The former addition

corresponds to the notion of regular expressions with equality [Libkin and Vrgoč 2012], and the latter to standard XPath data-value comparisons.

To compare GXPath(=) with d-TriAL*, we use the same convention as for navigational languages.

PROPOSITION A.8. *GXPath(=) is strictly contained in* d-TriAL*.

PROOF. The proof here follows the same lines as the one of Theorem 6.9. Because of this we only have to show how to define an equivalent d-TriAL* expression for any of the newly added data operators in GXPath(=).

— For $\varphi = \langle \alpha = \beta \rangle$ we define $E_\varphi = E_\alpha \bowtie^{1,1,1}_{1=1',\rho(3)=\rho(3')} E_\beta$
— For $\varphi = \langle \alpha \neq \beta \rangle$ we define $E_\varphi = E_\alpha \bowtie^{1,1,1}_{1=1',\rho(3)\neq\rho(3')} E_\beta$
— For $\alpha' = \alpha_=$ we define $E_{\alpha'} = E_\alpha \bowtie^{1,1,3}_{\rho(1)=\rho(3)} E_\alpha$
— For $\alpha' = \alpha_\neq$ we define $E_{\alpha'} = E_\alpha \bowtie^{1,1,3}_{\rho(1)\neq\rho(3)} E_\alpha$

It is again straightforward to see that the described translations works as desired.

To show that the containment is strict we use a similar approach as when proving Theorem 6.9. We first notice that the fact that GXPath is contained in $\mathcal{L}^3_{\infty,\omega}$ [Vrgoč 2014] can easily be extended to show that GXPath(=) is subsumed by $\mathcal{L}^3_{\infty,\omega}(\sim)$, the infinitary three variable logic with data value tests. Here the only addition to the logic is the ability to use formulas of the form $x \sim y$ that are true if and only if x and y have the same data value (since we do not use comparisons with constants we do not have to worry about the language being two sorted).

More formally, we will represent a data graph $G = (V, E, \rho)$ as a FO structure $G = (V, (E_a : a \in \Sigma), \sim)$ with $E_a = \{(v, v') : (v, a, v') \in E\}$ and $v \sim v'$ if and only if $\rho(v) = \rho(v')$. It is straightforward to see that with this interpretation we have GXPath(=) $\subseteq \mathcal{L}^3_{\infty,\omega}(\sim)$.

It is also easy to see that the 3-pebble game [Libkin 2004] for $\mathcal{L}^3_{\infty,\omega}(\sim)$ follows the intended semantics when interpreted over data graphs. (Note that the game works over any class of structures, but over data graphs only relations are edge relations and the data value comparison.)

We can now play the 3-pebble game over the 3-clique graph and the 4-clique graph [Libkin 2004] where all data values are the same. The same winning strategy for the duplicator as in the game with no data values will still work, so we conclude that $\mathcal{L}^3_{\infty,\omega}(\sim)$ can not distinguish the two models.

Consider now the following TriAL expression:

$$U_G \overset{1,2,3}{\underset{\varphi}{\bowtie}} U_G,$$

where $\varphi = (1 \neq 2) \wedge (1 \neq 3) \wedge (1 \neq 1') \wedge (2 \neq 3) \wedge (2 \neq 1') \wedge (3 \neq 1')$ with U_G as defined previously. It follows easily that this expression has different answer on the two models (since it asks for four different nodes in the original graph database). This finishes our proof. □

This also implies that d-TriAL* subsumes an extension of RPQs based on regular expressions with equality [Libkin and Vrgoč 2012], 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* [Kaminski and Francez 1994]. 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 [Libkin and Vrgoč 2012; Kaminski and Francez 1994] for precise definitions, and state the comparison result below.

PROPOSITION A.9. *d-TriAL* is incomparable in terms of expressive power with register automata.*

PROOF. We begin by showing that register automata are not contained in d-TriAL*. To see this recall from Lemma 6.12 that TriAL* is subsumed by infinitary logic $\mathcal{L}^6_{\infty,\omega}$ (an extension to include data value comparisons is straightforward).

Next we observe that for any number n, register automata can define a property not expressible in $\mathcal{L}^n_{\infty,\omega}$. For this consider the following regular expression with memory, shown in [Libkin and Vrgoč 2012] to be equivalent to register automata:

$e_2 :=\downarrow x_1 a[x_1^{\neq}] \downarrow x_2$

$e_{n+1} := e_n \cdot a[x_1^{\neq} \wedge x_2^{\neq} \wedge \cdots \wedge x_n^{\neq}] \downarrow x_{n+1}$.

Since no node can have more than one data value attached it follows that the answer to the query posted by the expression e_n is nonempty if and only if the graph database has at least n different elements.

It is well known [Libkin 2004] that $\mathcal{L}^n_{\infty,\omega}$ can not define a query stating that the model has at least $n+1$ element. Since TriAL* is contained in $\mathcal{L}^6_{\infty,\omega}$ the desired result follows from the fact that e_7 is nonempty only on the graphs with at least 7 elements.

To show that there are d-TriAL* queries outside of reach of register automata, recall that d-TriAL* subsumes GXPath(=) (Theorem A.8) and the later already has the required property [Vrgoč 2014]. □