TriAL: A Navigational Algebra for RDF Triplestores

LEONID LIBKIN, University of Edinburgh
JUAN L. REUTTER, Pontifica Universidad Católica de Chile and Center for Semantic Web Research
ADRIÁN SOTO, Pontifica Universidad Católica de Chile and Center for Semantic Web Research
DOMAGOJ VRGOČ, Pontifica Universidad Católica de Chile and Center for Semantic Web Research

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

Our goal is to introduce languages that work directly over triples and are closed, i.e., they produce sets of triples, rather than graphs. Our basic language is called TriAL, or Triple Algebra: it guarantees closure properties by replacing the product with a family of join operations. We extend TriAL with recursion and explain why such an extension is more intricate for triples than for graphs. We present a declarative language, namely a fragment of datalog, capturing the recursive algebra. For both languages, the combined complexity of query evaluation is given by low-degree polynomials. We compare our language with 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 recursive TriAL on top of a relational query engine, and we 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.

CCS Concepts: • **Information systems → Query languages**; *Resource Description Framework (RDF)*; • **Theory of computation → Database query languages (principles)**;

Additional Key Words and Phrases: RDF, Triple Algebra, Query evaluation

ACM Reference format:
Leonid Libkin, Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. 2018. TriAL: A Navigational Algebra for RDF Triplestores. *ACM Trans. Database Syst.* 43, 1, Article 5 (March 2018), 46 pages.
https://doi.org/10.1145/3154385

5

Libkin was supported by the EPSRC Grants No. EP/N023056/1 and No. EP/M025268/1. Reutter, Soto, and Vrgoč were supported by the Nucleus Millenium Center for Semantic Web Research under Grant No. NC12004. Soto was also supported by CONICYT-PCHA Doctorado Nacional Grant No. 2017-21171731. Vrgoč was also supported by the FONDECYT Project No. 11160383.

Authors' addresses: L. Libkin, LFCS, School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, UK; email: libkin@inf.ed.ac.uk; J. L. Reutter, A. Soto, and D. Vrgoč, Department of Computer Science, School of Engineering, Pontifica Universidad Católica de Chile, Vicuña Mackenna 4860, Macul 7820436, Santiago, Chile; emails: jreutter@ing.puc.cl, assoto@uc.cl, dvrgoc@ing.puc.cl.

https://doi.org/10.1145/3154385

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 References [2–4, 20, 57] 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 simplest model of graph databases [4, 57], which dates back to References [18, 19], 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.

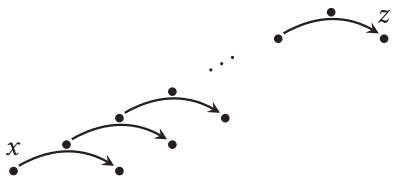
The model of RDF data is very similar, yet slightly different. The basic concept is a *triple* (s, p, o), which consists of the subject s, the predicate p, and the object o, drawn from a domain of uniform resource identifiers (URIs).[1] Thus, the middle element need not come from a finite alphabet and 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 one of their basic building blocks, a way of expressing *reachability* or other properties that are dependent on the *path* between two nodes. The most typical one is that of *regular path queries*, or RPQs [19], which 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 Reach$_{\rightarrow}$ looks for pairs (x, z) connected by paths of the following shape:

and 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 [32], the standard query language for RDF data, to allow navigational queries. However, one can easily see that property paths are essentially regular path queries in disguise [35] and thus do not account for

[1] As we explain in Section 2, for a simpler presentation we do not deal with literals or blank nodes.

patterns such as, e.g., Reach$_{\nearrow}$ above, since they view RDF triples as a graph database. A similar attempt to add navigation to RDF languages was made in Reference [46], where nSPARQL, an extension to SPARQL using a generalisation of property paths known as *nested regular expressions*, 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 [8] 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 navigational primitives and graph primitives 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 and develop a language for direct RDF querying.

Since in navigational queries we want to repeat a pattern an arbitrary number of times (such as in the reachability queries above), we would like each application to produce a result that is still described by our data model. Therefore, a crucial property of our language will be *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 nodes 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 we 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 its analog for triples.

But queries Reach$_{\rightarrow}$ and 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, we move on to the comparison of TriAL with graph navigational primitives. In particular, we show that the navigational power of TriAL* subsumes that of both regular path queries and nested regular expressions (and thus also SPARQL property paths). In fact, it subsumes a version of *XPath* recently proposed for graph databases [37]. We also compare it with conjunctive RPQs [18] and some of their extensions studied in References [15, 16].

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 we compare its performance against SPARQL systems at the time of computing property path queries. We 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. We also test more expressive recursive queries on synthetic data, showing how the language remains feasible in practice when looking for complex recursive patterns.

All together, this article shows that TriAL* is an expressive language that subsumes a number of well-known 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.

New contributions. The material presented in this article is based on a conference article [39], which introduces the query language presented here and discusses some of its basic theoretical properties. The main contributions added to this article not present in Reference [39] can be summarised as follows:

- *A working implementation of the proposed language.* While Reference [39] introduces a navigational algebra for RDF and shows its basic properties, the contributions there are purely theoretical and are not concerned with applicability of the framework. In contrast, we implemented the TriAL language and tested it both on real world and synthetic data using a wide range of queries. These result are presented in Section 7 and were not present in Reference [39].
- *Full proofs of all the theorems.* Reference [39] only announces and sketches the main theoretical results about the framework we propose. Here all technical details of the proofs are present and fully elaborated, as witnessed by the difference between the length of the two papers.
- *New theoretical results.* Some new results, such as the ability to eliminate the right-Kleene closure using the left one and vice versa have been added.
- *Simplification of the framework.* In Reference [39], the data model we used went beyond RDF, thus somewhat obfuscating its main purpose. Here we base the presentation on the model of RDF triples, thus simplifying the query language and making it directly applicable to this context. We then present all the extensions to the more general model of property graphs in the online appendix to the paper.

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 Section 6, we compare our language to different graph querying formalisms. Section 7 shows how an implementation of TriAL* performs over real-world data and when compared to modern SPARQL engines. We conclude in Section 9.

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.

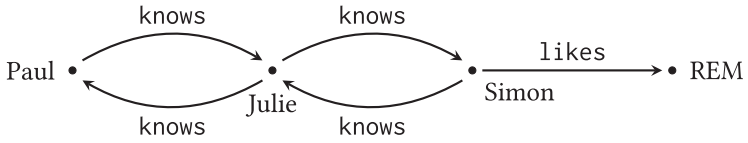


Fig. 1. A graph database showing part of a Social Network. Edge labels are on the edges, and node names next to the nodes.

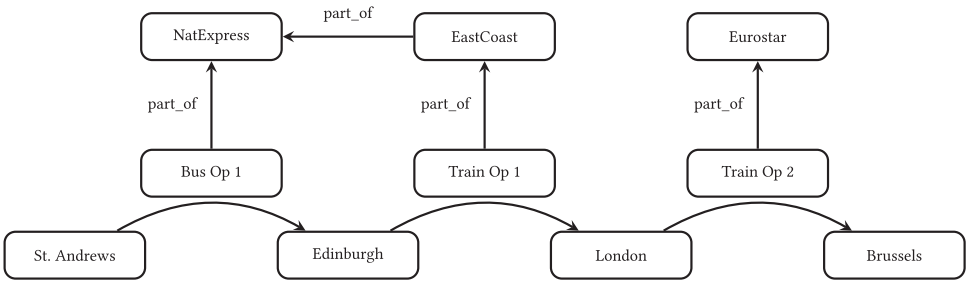


Fig. 2. RDF graph storing information about cities and transport services between them.

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 database is shown in Figure 1. Here, the nodes represent people or other entities in a Social Network setting and the edges connection between two entities.

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 [46], 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, transportation services 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 [32]. However, property paths, as well as their theoretical counterparts and extensions [6, 41, 46] 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 (NRE) introduced in Reference [46], which extend SPARQL property paths with several extra functionalities. The idea behind nested regular expressions is to combine the usual reachability patterns of graph query languages with the XPath mechanism of node tests.[2] However, nested regular expressions 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 Reference [8]; 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

[2]For a formal definition of nested regular expressions, see Reference [46]. 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.

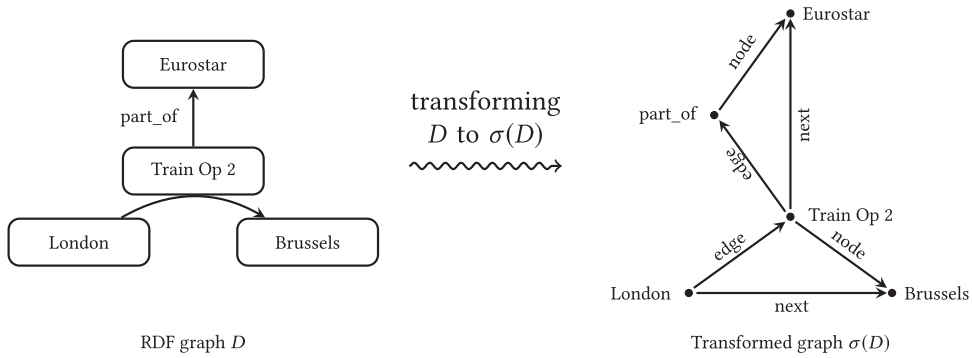


Fig. 3. Transforming part of the RDF database from Figure 2 into a graph database.

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

Fig. 4. Two triplestores D_1 and D_2.

problem of interpreting RDFS features within SPARQL [46]). At the same time, it is not sufficient for expressing simple reachability patterns like those in query Q:

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 from Figure 4. 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 Figure 5.

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 6. 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.

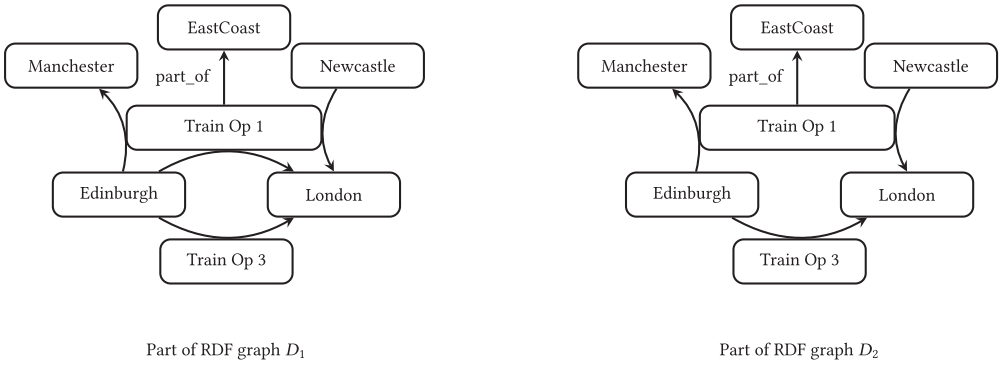


Fig. 5. Parts of D_1 and D_2 different from Figure 2.

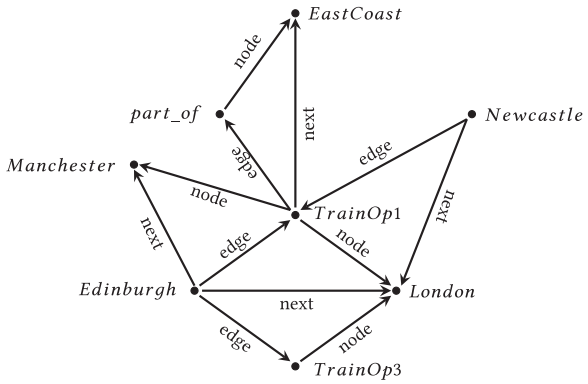


Fig. 6. Transforming part of the RDF databases D_1 and D_2.

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 [46], 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 Reference [46]) 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 $\sigma(D) = \sigma(D')$.[3] Now, since nSPARQL is the extension of a positive fragment of SPARQL 1.0 [45]

[3]The NREs defined in Reference [46] 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 Reference [46] 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 Reference [46].

with NREs, and SPARQL 1.0 was shown to be equivalent to first order logic [47], we can easily see that Q cannot be expressed in SPARQL 1.0, and therefore in nSPARQL if no NREs are used. We can now use Proposition 2.2 to show that it cannot be expressed in nSPARQL either. Assuming the query Q is indeed expressible by an nSPARQL expression S with m patterns, we just need to follow the proof of Proposition 2.2 to construct two graphs having more than m copies of D_1 and, respectively, D_2, so that at least one of them will have to be witnessed by an NRE that will not be able to distinguish both graphs, resulting in a contradiction. □

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.

2.3 Triplestore Databases

To introduce proper triple-based navigational languages, we first define a simple model of triplestores, which corresponds essentially to ground RDF documents. Let $\mathcal{O}$ be a countably infinite set of objects that can appear in our database.[4]

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 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 such as named graphs in the RDF setting. It 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.

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 that 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.

[4]This is our analogue of URIs in RDF.

Cartesian product, however, would create a relation of arity six, but instead we use *joins* that only keep three positions in the result.

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

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

Here, we refer to the positions in S as 1 and 2, and to the positions in S' as $1'$ and $2'$, so the join condition is $2 = 1'$ (written below the join symbol), and the output has positions 1 and $2'$. This suggests that our join operations on triples should be of the form $R \bowtie^{i,j,k}_{\text{cond}} R'$, where R and R' are 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 are (not necessarily distinct) elements of the set $\{1, 1', 2, 2', 3, 3'\}$,
- θ is a set of equalities and inequalities of the form:
  - $l = m$, where $l, m \in \{1, 1', 2, 2', 3, 3'\}$ comparing two values in our tirples; or
  - $l = o$, where $l \in \{1, 1', 2, 2', 3, 3'\}$ and $o \in O$ comparing an element of one of the triples to a constant.

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 $l = m$ ($l \neq m$, respectively), with $l, m \in \{1, 1', 2, 2', 3, 3'\}$, is in θ, then $o_l = o_m$ ($o_l \neq o_m$ resp.);
- if $l = o$ ($l \neq o$, respectively), with $o \in Obj$ and $l \in \{1, 1', 2, 2', 3, 3'\}$, is in θ, then $o_l = o$ ($o_l \neq o$ resp.).

Note that in the case that $\theta = \emptyset$ (that is, when no condition is specified), the join operation is equal to a projection to precisely three indices of the Cartesian product of R and R'.

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 $\{1, 2, 3\} \cup O$, 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.[5]

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 [32].

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 in the relation on the left, and indexes $(1', 2', 3')$ to positions in the relation on the right of 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 Figure 2, the triple (`London`, `Train Op 2`, `Brussels`) is joined with the triple (`Train Op 2`, `part_of`, `Eurostar`), producing (`London`, `Eurostar`, `Brussels`); the full result of evaluating e over the triplestore from Figure 2 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 into the result of our query (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 $(R \bowtie^{1,2',3} R') \bowtie^{1,2,3'} R'$, where the join conditions are empty. 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

[5]Note that one can in fact simulate the filter $\sigma_\theta(e)$ using the join $e \bowtie^{1,2,3}_{\theta} e$. We, however, opt for keeping the filter operator as a primitive in our language as it is easier to write and more intuitive.

need reachability, which relational algebra is well known to be incapable of expressing. Thus, we need to add recursion to our language.

To do so, we expand TriAL with *right* and *left Kleene closure* of any triple join $\bowtie_\theta^{i,j,k}$ over an expression e, denoted as $(e \bowtie_\theta^{i,j,k})^*$ for right, and $(\bowtie_\theta^{i,j,k} e)^*$ for left.[6] 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 ternary relations 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 expression

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

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

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

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

However, even though the two closure operations are not associative, one can in fact show that both closures are equivalent in terms of expressive power, and thus strictly speaking only one of these closures is needed. As usual, we say that two TriAL* expressions e and e' are equivalent if $e(T) = e'(T)$ for every triplestore T. We then have:

Proposition 3.3. *For every* TriAL* *expression e there are expressions e_l and e_r, where e_l does not use the right Kleene closure operator $(e \bowtie)^*$ and e_r does not use the left Kleene closure operator $(\bowtie e)^*$, and such that all of e, e_l and e_r are equivalent.*

Proof. Let us define the function $s : \{1, 2, 3, 1', 2', 3'\} \longrightarrow \{1, 2, 3, 1', 2', 3'\}$ such that $s(i) \in \{1', 2', 3'\}$ when $i \in \{1, 2, 3\}$ and $s(i) \in \{1, 2, 3\}$ when $i \in \{1', 2', 3'\}$. We extend the function for equalities and inequalities, so that $s(a \sim b)$ corresponds to $s(a) \sim s(b)$, for $\sim \in \{=, \neq\}$. Likewise, we extend the function to any set θ of equalities and inequalities, so that $s(\theta)$ is the set containing the (in)equality $s(\alpha)$ for each (in)equality α in θ. The following now follows from simple inspection:

Lemma 3.4. *Each* TriAL* *expression $A \overset{i,j,k}{\underset{\theta}{\bowtie}} B$ is equivalent to $B \overset{s(i),s(j),s(k)}{\underset{s(\theta)}{\bowtie}} A$.*

For the proof of this proposition, we show that we can apply the same transformation to Kleene closure of joins. Namely, we show that for every TriAL* expression e, we have that

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

[6] The intuition behind the used notation is as follows. In the right Kleene closure, $(e \bowtie)^*$, the position of the join symbol signifies that the base relation is joined to the right of the recursive part computed thus far, while in the $(\bowtie e)^*$ it is joined from the left.

Note first that this suffices for the proof, as any TriAL* expression e can be transformed to any of e_l and e_r by transforming all right Kleene closures to left Kleene closures (for the case of e_l) or all left Kleene closures to right Kleene closures (for e_r).

We prove this statement by induction. To do this, we define operators

$$t_l(e \overset{i,j,k}{\underset{\theta}{\bowtie}}) = \begin{cases} \emptyset & l = 0 \\ e & l = 1 \\ t_{l-1} \overset{i,j,k}{\underset{\theta}{\bowtie}} e & l > 1 \end{cases} \quad \text{and} \quad t_l(\overset{i,j,k}{\underset{\theta}{\bowtie}} e) = \begin{cases} \emptyset & l = 0 \\ e & l = 1 \\ e \overset{i,j,k}{\underset{\theta}{\bowtie}} t_{l-1} & l > 1, \end{cases}$$

and show that $t_l(e \bowtie_{\theta}^{i,j,k}) = t_l(\bowtie_{s(\theta)}^{s(i),s(j),s(k)} e)$. It is clear that the equality holds for $l = 0$ and $l = 1$. Now suppose the above equality holds for $l = n$. So, we have that

$$t_{n+1}(e \overset{i,j,k}{\underset{\theta}{\bowtie}}) = t_n(e \overset{i,j,k}{\underset{\theta}{\bowtie}}) \overset{i,j,k}{\underset{\theta}{\bowtie}} e = t_n(\overset{s(i),s(j),s(k)}{\underset{s(\theta)}{\bowtie}} e) \overset{i,j,k}{\underset{\theta}{\bowtie}} e.$$

From Lemma 3.4, we obtain that

$$t_n(\overset{s(i),s(j),s(k)}{\underset{s(\theta)}{\bowtie}} e) \overset{i,j,k}{\underset{\theta}{\bowtie}} e = e \overset{s(i),s(j),s(k)}{\underset{s(\theta)}{\bowtie}} t_n(\overset{s(i),s(j),s(k)}{\underset{s(\theta)}{\bowtie}} e),$$

which corresponds to $t_{n+1}(\bowtie_{s(\theta)}^{s(i),s(j),s(k)} e)$ by definition. □

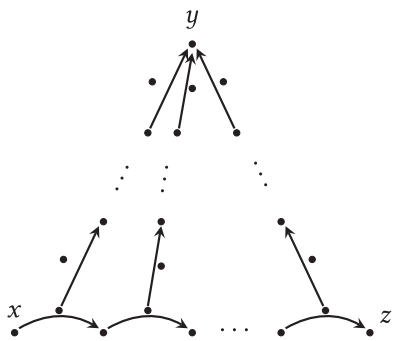
Now, we present several examples of queries one can ask using the Triple Algebra.

Example 3.5. We refer now to reachability queries Reach$_{\rightarrow}$ and 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 Q from Proposition 2.2, which we denote by 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 $(E \underset{2=1'}{\overset{1,3',3}{\bowtie}})^*$ computes all triples (x, y, z), such that $E(x, w, z)$ holds for some w, and y is reachable from w using some E-path. The outer join now simply computes the transitive closure of this relation, taking into account that the service that witnesses the connection between the cities is the same.

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 1. Here, we give some remarks about notation and implicit assumptions in the remainder of the article.

- 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 view of Proposition 3.3, in the proofs we will usually handle only the case of the right Kleene closure $(R \bowtie)^*$. The proofs for the left closure are usually completely symmetric.
- As usual in database theory, we only consider queries that are domain-independent, and therefore we lose 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, such as its well documented success in Web information extraction [28]. So it seems natural to look for Datalog fragments to capture TriAL and its recursive version, and we show in Section 4.1 that this can indeed be done, for both languages.

Of course, one could also compare TriAL and TriAL* against other standard logical formalisms like First Order Logic (FO) for the non-recursive variant or various extensions of FO, for the recursive TriAL*. Unfortunately, this time the comparison is not as clean as in the case of Datalog, as we need to focus on the number of *variables* used in the first order formalisms. This is done in Section 4.2.

4.1 TripleDatalog$^\neg$ and ReachTripleDatalog$^\neg$ Programs

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 relation 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 [1], 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 vectors of precisely three variables;
(3) u_is and v_is are either variables or objects in 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 Equation (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. All predicates appearing in the head of a rule of Π are called the *intensional* predicates. We assume that each program has a distinguished intensional predicate called *Ans*.

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 predicate 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 Reference [1] for a precise definition). The answer of the evaluation of a program Π over an instance I, denoted as $\Pi(I)$, is the evaluation of the answer predicate *Ans*, that is, the set of facts that can be deduced for *Ans* by applying all rules of Π, in the order explained above, as is customary for non-recursive programs.

Example 4.1. To illustrate how TripleDatalog$^\neg$ works consider again the query e from example 3.1 given by the expression

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

Then, e can be expressed as the following TripleDatalog$^\neg$ program:

$$Ans(x_1, x_3', x_3) \leftarrow E(x_1, x_2, x_3), E(x_1', x_2', x_3'), x_2 = x_1'.$$

Note that if the join additionally asked for inequalities such as, e.g., $2 \neq 3'$, we could simply add the inequality $x_2 \neq x_3'$ to our rule.

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 $\mathcal{L}_2$. The languages are equivalent if each is contained in the other, and they are incomparable if none is contained in the other. We are now ready to present the first capturing result.

Proposition 4.2. 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 $e_1 \bowtie_\theta^{i,j,k} e_2$, then 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

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

  where for each pth 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.

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 a TriAL expression e_S for each intentional predicate S of Π, in an inductive fashion, following its dependency graph.

- 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.
- Let S be a predicate appearing in the head of a rule in Π, and assume that Π_S is the set of all rules of Π mentioning S in the head. For each rule σ in Π_S, we construct a TriAL expression e_σ as follows. Assume σ is of the form

$$S(x_a, x_b, x_c) \leftarrow S_1(x_1, x_2, x_3), S_2(x_4, x_5, x_6), u_1(\neq) = v_1, \ldots, u_m(\neq) = v_m, \tag{3}$$

  where S_1 and S_2 are (not necessarily distinct) predicate symbols of arity 3 and all variables x_a, x_b, x_c, each u_j and each v_j are one of $x_1, \ldots, x_6$. Then, e_σ is

$$e_{S_1} \bowtie_{\theta^j}^{a,b,c} e_{S_2},$$

  where θ contains an (in)equality $\ell = k$ for each (in)equality $x_\ell = x_k$ in the rule. If either of S_1 or S_2 appear negated in the rule, then just replace e_{S_1} for $(e_{S_1})^c$ or $(e_{S_2})^c$. Note that the existence of e_{S_1} and e_{S_2} is given by the acyclicity of the program.

  The TriAL expression e_S, corresponding to the predicate S, is now

$$\bigcup_{\sigma \in \Pi_S} e_\sigma$$

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 Reference [18]. A ReachTripleDatalog$^\neg$ *program* is a TripleDatalog$^\neg$ program in which each recursive predicate S is the head of exactly two rules of the form

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

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 ReachTripleDatalog$^\neg$ programs is *stratified*. The semantics of these programs is the standard least-fixpoint semantics [1]. A similarly defined syntactic class, but over graph databases, rather than triplestores, was shown to capture the expressive power of FO with the transitive closure operator [18].

To illustrate how ReachTripleDatalog$^\neg$ works, we give a simple program that computes the query Q from Proposition 2.2.

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.

Finally, we show that ReachTripleDatalog$^\neg$ indeed captures TriAL*.

Theorem 4.4. *The expressive power of* TriAL* *and* ReachTripleDatalog$^\neg$ *programs is the same.*

Proof. Let us first show the containment of TriAL* in ReachTripleDatalog$^\neg$. The proof goes along the same lines as the proof of containment of TriAL in TripleDatalog$^\neg$. We have to show that for every TriAL* expression e there is a 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 TripleDatalog$^\neg$ is the treatment of the constructs $e = (e_1 \bowtie^{i,j,k}_\theta)^*$ and $e = (\bowtie^{i,j,k}_\theta e_1)^*$. For the former construct (the other one is symmetrical), assume that $\theta = (\bigwedge_{1 \le i \le m} p_i(\neq) = q_i)$. We let Π_e be the union of all rules of Π_{e_1}, plus rules

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

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

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

(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 $(e_R \bowtie^{a,b,c}_{\theta})^*$, 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. □

Remark 2. Note that TriAL expressions and Datalog programs exhibit the same type of relationship as regular expressions and finite state automata do; namely, the translation from TriAL (TriAL*, respectively) to TripleDatalog$^\neg$ (ReachTripleDatalog$^\neg$, respectively) can be carried out in linear time, while going the other direction can result in an exponential blow-up. This fact easily follows by examining the proofs of Proposition 4.2 and Theorem 4.4 and by observing that our Datalog variants allow folding by introducing new predicate names.

4.2 Triple Algebra and Other Relational Query Languages

As we have mentioned, even though the recursive part of TriAL* demands for a comparison with Datalog or other recursive languages, we could very well compare TriAL with FO. Since TriAL is a restriction of relational algebra, of course it is contained in FO; the more interesting comparison is when we focus on fragments of this logic with limited use of variables. In this context, we can show that the expressive power of TriAL lies between FO^3 and FO^6, the fragments of FO using 3 and 6 variables, respectively, and between FO^3 and FO^4 if inequalities are disallowed. As we see in Section 5, this results not only help us understand the limitations of TriAL (and therefore TriAL*) in more detail, but they also serve as a good machinery for proving various complexity bounds for problems related to query answering. We also note that several querying formalisms for graph or semistructured data have been shown to be equivalent to fragments FO with a restricted number of variables, such as XPath and FO^3 over trees [42] or navigational graph query languages and FO^3 over graphs [25].

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 $\langle E_1, \ldots, E_n \rangle$. 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, 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. As usual, the *answer* of a formula φ with free variables $x_1, \ldots, x_n$ over an instance I, denoted as $\varphi(I)$, is the set of all tuples $\tau(x_1), \ldots, \tau(x_n)$ such that τ is an assignment verifying $(I, \sigma) \models \varphi$.

To give an intuition why fragments of FO with limited use of variables 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: $\varphi(x_1, x_{3'}, x_3) = \exists x_2 \exists x_{1'} \exists x_{2'} (E(x_1, x_2, x_3) \wedge E(x_{1'}, x_{2'}, x_{3'}) \wedge x_2 = x_{2'})$. This suggests that we can simulate joins using only six variables, and this extends rather easily to the whole algebra. On the other hand, since TriAL contains all relational operators running over predicates of arity 3, one might think that TriAL can actually simulate all of FO^3. This is indeed the case.

Proposition 4.5.

- TriAL *is contained in* FO^6
- FO^3 *is contained in* TriAL.

Proof. Let us show first that TriAL is contained in FO^6. Let e be a TriAL expression. We construct an FO^6 formula φ_e such that $e(T) = \varphi_e(\mathcal{M}_T)$, 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 $e = e_1 \bowtie^{i,j,k}_{\theta} e_2$, then $\varphi_e(x_i, x_j, x_k) = \exists x_u \exists x_v \exists x_w \varphi_{e_1}(x_1, x_2, x_3) \wedge \varphi_{e_2}(x_{1'}, x_{2'}, x_{3'}) \wedge \alpha(\theta)$, 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 $o \in O$ and $p, q \in \{1, 1', 2, 2', 3, 3'\}$
- Similarly, if $e = \sigma_\theta e_1$ then $\varphi_e(x, y, z) = \varphi_{e_1}(x, y, z) \wedge \alpha(\theta)$, where $\alpha(\theta)$ is defined as in the previous bullet.

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

Next, we show that FO^3 is contained in TriAL. For every FO^3 formula φ, we construct an equivalent TriAL expression e_φ such that $e_\varphi(T) = \varphi(\mathcal{M}_T)$, 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 on x_i: $\bar{d}$ equals $1', 2, 3$ if $i = 1$, equals $1, 2', 3$ if $i = 2$ and it equals $1, 2, 3'$ if $i = 3$. Intuitively, when quantifying x_i, we do a projection on the ith 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).

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

Thus, the expressive power of TriAL should lie somewhere between FO^3 and FO^6: we can simulate all of FO^3 within our triplestore environment, but, as the following proposition shows, we need 6 variables to simulate the join operator.

Proposition 4.6. TriAL *is not contained in* FO^5.

Proof. We show that the following TriAL expression cannot be expressed in FO^5:

$$e_6 := \left(E \underset{\theta}{\overset{1,2,3}{\bowtie}} E\right), \text{ with } \theta = \bigwedge_{i,j \in \{1,2,3,1',2',3'\}, i \neq j} i \neq j.$$

This query states that our triplestore has two tuples in E where all objects are different.

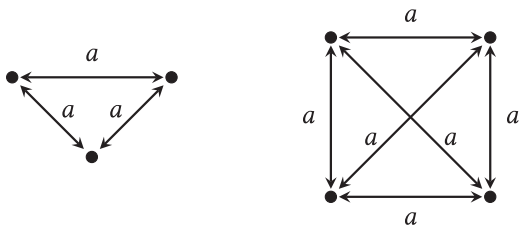
To see that this query is not expressible in FO^5, construct triplestores $T_5 = (O_5, E_5)$ with $O_5 = \{a, b, c, d, e\}$, and $E_5 = O_5 \times O_5 \times O_5$, and $T_6 = (O_6, E_6)$ with $O_6 = \{a, b, c, d, e, f\}$ and $E_6 = O_6 \times O_6 \times O_6$. It is immediate to see that that the duplicator has a winning strategy in a 5-pebble game on these two structures, so they can not be distinguished by an FO^5 formula—and in fact any formula in the infinitary logic $\mathcal{L}^5_{\infty,\omega}$ (see Reference [36]). On the other hand, our expression e_6 does distinguish them and is thus not expressible in FO^5 nor in $\mathcal{L}^5_{\infty,\omega}$. □

Interestingly, when no inequalities are allowed in TriAL one can instead show that the resulting formalism can be simulated with just four variables, and this time the containment in FO^3 is strict. Formally, let us denote by $\text{TriAL}^=$ the fragment of TriAL that only uses equalities in selections and joins.

Proposition 4.7.

- $\text{TriAL}^=$ *is not contained in* FO^3.
- $\text{TriAL}^=$ *is contained in* FO^4.

Proof. We first show that $\text{TriAL}^=$ is not contained in FO^3. Consider the following two triplestores T_1 and T_2



T_1 T_2

It is well known that no $\mathcal{L}^3_{\infty\omega}$ sentence F can distinguish the two models (see, e.g., Reference [36]). This is due to the fact that that duplicator has a winning strategy in an infinite 3-pebble game on these graphs, simply by preserving equality of pebbled elements. That is, for any F, we have $G_1 \models F$ iff $G_2 \models F$.

On the other hand, we exhibit a $\text{TriAL}^=$ expression that does distinguish these structures. Let us first define expression e as

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

Intuitively, e stores triples (n_1, n_2, n_3) such that there are triples (n_1, a, n_2), (n_2, a, n_3), and (n_1, a, n_3) (that is, a triangle) in the triplestore. Our expression is

$$(e \underset{1=1',3=3'}{\overset{1,2,2'}{\bowtie}} e) \underset{2=1',3=3'.2'=a}{\overset{1,2,3}{\bowtie}} .$$

The inner sub-expression computes all triples (n_1, n_2, n_3) such that there is an n_4 for which both (n_1, n_2, n_4) and (n_1, n_3, n_4) belong to e, and the outer sub-expression just checks that there is also a triple (n_2, a, n_3). In other words, if (n_1, n_2, n_4) and (n_1, n_3, n_4) are part of a triangle (such as in T_1), then there is a triple (n_2, a, n_3).

Clearly, the answer of this expression is nonempty on T_2, while empty on T_1. Since T_1 and T_2 are indistinguishable by FO^3, it follows that $\mathrm{TriAL}^=$ is not contained in FO^3.

Next, we show that $\mathrm{TriAL}^=$ is contained in FO^4. For every $\mathrm{TriAL}^=$ expression e, we construct an FO^4 formula φ_e such that a triple (a, b, c) belongs to $e(T)$ if and only if it belongs to $\varphi_e(\mathcal{M}_T)$. The proof is done by induction, with all cases following from the proof of the first part of Proposition 4.5 except when $e = e_1 \bowtie^{i,j,k}_{\theta} e_2$, which is the only interesting case.

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

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

- θ_ℓ and θ_r contain only equalities between indexes in $\{1, 2, 3\}$ and $\{1', 2', 3'\}$, respectively.
- θ^c_ℓ and θ^c_r contain only equalities where one element is in 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 O and an element in $\{1, 1', 2, 2', 3, 3'\}$, we let $\alpha(\theta^c) = \bigwedge_{o=i \in \theta^c} o = x_i$.

To construct formula φ_e, we distinguish two types of joins:

- Joins of form $e = e_1 \bowtie^{i,j,k}_{\theta} e_2$ 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

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

  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 $e = e_1 \bowtie^{1,2,3}_{1=2} e_2$. 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 $e = e_1 \bowtie^{i,j,k}_{\theta} e_2$ 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

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

  —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

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

Having established how to construct φ_e, it is now straightforward to show that it satisfies the desired property. □

TriAL* *and Infinitary Logic.* As expected, we can lift some of the results for TriAL and FO into TriAL* if we use instead the the infinitary logic $\mathcal{L}^{w}_{\infty,w}$. In other words, this suggests that the kleene star only gives us more power to express paths, but it does not help us in computing more complex first order or relational queries.

Proposition 4.8. TriAL* *is contained in the infinitary logic* $\mathcal{L}^{6}_{\infty,\omega}$ *and is not contained in the infinitary logic* $\mathcal{L}^{5}_{\infty,\omega}$.

Proof. That TriAL* is not contained in $\mathcal{L}^{5}_{\infty,\omega}$ follows from Proposition 4.6, since TriAL* is more expressive than TriAL. Next, we show containment in $\mathcal{L}^{6}_{\infty,\omega}$. Let e be a TriAL* expression. Just as before, we need to show how to construct an $\mathcal{L}^{6}_{\infty,\omega}$ formula φ_e such that $e(T) = \varphi_e(\mathcal{M}_T)$, for each triplestore T. The proof is again by induction. Since FO^6 is contained in $\mathcal{L}^{6}_{\infty,\omega}$ (see, e.g., Reference [36]), all the base cases except the star follow from the first part of Proposition 4.5, and so we only need to show how recursion is handled. For 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 $F_R(x_i, x_j, x_k)$ that is equivalent to R. 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:

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

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, which finishes our proof. □

Remark 3. To finish this section, we would like to note that there is nothing intrinsically special about ternary relations when considering the results above. In fact, we could easily define an algebra of relations of arity k analogously as we did in Section 3 for ternary relations and show that this algebra contains FO^k, that it is contained in FO^{2k} (as we did in Theorem 4.5), that it has a Datalog analogue, and so on.

The reason for restricting ourselves to ternary relations is because the RDF data format is based on triples. However, we believe that the ability to extend our results to relations of higher arity is also a sign that the notion of Kleene closure we introduce is a good generalisation of the notion of transitivity over binary relations.

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 $(x_1, x_2, x_3) \in e(T)$?

The majority of the navigational graph querying primitives in the literature (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 observation that, in the proof of Proposition 4.5, one can construct for every FO^3 query an equivalent TriAL query in polynomial size, and the fact that evaluating FO^k queries is Ptime-hard already when $k = 3$ [55].

For the NLogspace upper bound, the idea is to divide the expression e into all its subexpressions, corresponding to subtrees of the parse tree of e. 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, where $|e|$ is measured as the number of operators (joins, unions, etc.) in e, which is 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 prove 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. As with usual reachability algorithms, at each time we only need to remember the triple computed in the previous iteration and the one we are joining it with. Since this is doable in NLogspace space, the upper bound follows immediately. □

Tractable evaluation (even with respect to combined complexity) is practically a must when dealing with very large and dynamic semi-structured databases. However, 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 articles on graph database query evaluation (in particular, graph pattern matching algorithms), as well as bounded variable relational languages (cf. References [22, 23, 27]), 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.[7]

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:

[7]It is important to note that all bounds are in the size of the matrix representation of our triplestore, as described above.

(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, then 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, then 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), then we simply compute R_e as the union of these two matrices. Note that this takes time $O(|T|)$.

If $e = R_1 \cap R_2$, then 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 $e = R_1 - R_2$, then 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, then we can compute R_e in time $O(|\varphi||T|)$ 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 $|\varphi|$ 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 $|\varphi|$, because we have to check (in)equality only with the constants that actually appear in e.

Finally, in the case that $e = R_1 \bowtie^{i',j',k'}_{\theta} R_2$, we can compute R_e using Procedure 1 shown below:

PROCEDURE 1: Computing Joins

Input: Matrix representation of R_1, R_2
Output: Matrix R_e representing the result of evaluating 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** $i = 1 \to n$ **do**
5: **for** $j = 1 \to n$ **do**
6: **for** $k = 1 \to n$ **do**
7: **if** $R_1[i,j,k] = 1$ **then**
8: **for** $l = 1 \to n$ **do**
9: **for** $m = 1 \to n$ **do**
10: **for** $p = 1 \to n$ **do**
11: **if** $R_2[l,m,p] = 1$ **then**
12: **if** (o_i, o_j, o_k) and (o_l, o_m, o_p) 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 time $O(|\theta||T|)$ 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 $O(|\theta||T|^2)$. We recall again that the size of T is n^3, where n is the total number of objects in our triplestore.

Summing up all independent times, 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 the result of evaluating e
1: Initialize $R_e := R_1$ and $R_{tmp} := R_1$
2: **for** $i = 1 \to n^3$ **do**
3: Compute $R_{tmp} := R_{tmp} \bowtie_\theta^{i',j',k'} R_1$
4: Compute $R_e := R_e \cup R_{tmp}$

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. Note that this also makes a strong argument to prefer the use of the algebra in place of the Datalog programs of Section 4, since the latter have intrinsically high evaluation complexity (i.e., *EXPTIME* [21]).

5.1 Low-complexity Fragments

Even though we have acceptable combined complexity of query computation, if the size of T is very large, then 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 once again at TriAL$^=$, the fragment of TriAL in which the conditions θ used in joins can only use equalities.[8] We have seen that the expressive power of this fragment is apparently lower than the full TriAL. We'll also see that 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 5.3. *The* QUERYCOMPUTATION *problem for* 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 TriAL$^=$ with FO^4 established in Proposition 4.7. Indeed, by looking at the proof of Proposition 4.7, we can see that the transformation used in this proof is actually linear: for each TriAL$^=$ expression e one can compute, in time $O(|e|)$, an FO formula φ_e equivalent to e. To finish the proof, we show in Lemma 5.4 that

[8]Due to the fact that we use a matrix representation of triplestores (and intermediate results), we can still keep operations such as union and difference, since these can be computed using a single scan over the matrix domain.

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 5.3 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 $|\varphi_e| = O(|e|)$ (note as well that the transformation from a triplestore T to its relational structure $\mathcal{M}_T$ is trivially linear). The proof of Lemma 5.4 is given below. □

Lemma 5.4. *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 Proposition 4.7 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, then 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 5.4. □

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 TriAL$^=$ with recursion. However, we can improve this result for the fragment reachTA$^=$ that extends 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 reachTA$^=$. For this fragment, we have the same lower complexity bound.

Proposition 5.5. *The problem* QueryComputation *for* reachTA$^=$ *can be solved in time* $O(|e| \cdot |O| \cdot |T|)$.

Proof. To show this, we will use the algorithm presented in Proposition 5.3. 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 5.4 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 5.4. 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 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. □

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

Input: Matrix representation of R
Output: Matrix M_e representing the result of evaluating e

```
Precomputing the reachability matrix R_reach:
for i = 1 → n do
    for j = 1 → n do
        for k = 1 → n do
            if R[i,k,j] = 1 then
                R_reach[i,j] = 1
Compute the transitive closure R*_reach
Compute the output matrix M_e:
for i = 1 → n do
    for j = 1 → n do
        for k = 1 → n do
            if R[i,k,j] = 1 then
                for l = 1 → n do
                    if R*_reach[j,l] = 1 then
                        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 the result of evaluating e

```
for k = 1 → n do
    Precomputing the reachability matrix R^k_reach:
    for i = 1 → n do
        for j = 1 → n do
            if R[i,k,j] = 1 then
                R^k_reach[i,j] = 1
    Compute the transitive closure (R^k_reach)*
    Compute the output matrix M_e:
    for i = 1 → n do
        for j = 1 → n do
            if R[i,k,j] = 1 then
                for l = 1 → n do
                    if (R^k_reach)*[j,l] = 1 then
                        M_e[i,k,l] = 1
```

6 TRIPLE ALGEBRA AS A GRAPH LANGUAGE

So far, we have demonstrated some properties not expressible by graph query languages over RDF triplestores. The goal of this section is to go the other way around and compare TriAL* with navigational languages for 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, *conjunctive* regular path queries (CRPQs), and SPARQL property paths. As our yardstick language for comparison, we use a recently proposed

modification of the XML navigational language XPath, designed for graph querying [37]. This adaptation of XPath subsumes both NREs and RPQs. Its navigational fragment, presented next, is essentially Propositional Dynamic Logic (PDL) [31] 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 no_edge be a new label not appearing in Σ and define $\Sigma' = \Sigma \cup \{\text{no\_edge}\}$. Next, let $E' = E \cup \{(v, \text{no\_edge}, v') | (v, v') \notin \pi_{1,3}(E)\}$, 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 to allow each element of the triplestore domain to appear in the relation E, as per Definition 2.4. 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. 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 $E' \bowtie^{1,2,3}_{2 \neq \text{no\_edge}} E'$. 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 6.1. *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, then 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.

Basis:

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

Inductive step:

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

Note that in the case of path formulas, we only care about the first and the third element of the triple relation representing it, while for node formulas, we only care about the third element of the triple. For all our considerations the other elements play no role and are simply place holders.

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 $(a, x, b) \in E_\alpha(T_G)$ for some x 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 $(a, x, c) \in E_\alpha(T_G)$ and $(c, x', b) \in E_\beta(T_G)$, for some $x, x' \in O$. By the definition of the join, we conclude that $(a, x, b) \in E_{\alpha'}(T_G)$. 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 $\mathcal{L}^3_{\infty,\omega}$ [56]. Consider now the following TriAL expression:

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

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 References [25, 26] 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 6.2.

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

Noting that SPARQL property paths are just a syntactic variant of two-way RPQs[9] [35], Theorem 6.1 also gives us the following:

Corollary 6.3.

- *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) [11]. 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.

To do these comparisons, we will rely on the fact that TriAL* is subsumed by infinitary logic with six variables, as shown in Section 4.2, which basically shows that TriAL* cannot express all conjunctive queries (while CNREs can).

When comparing TriAL* with CNREs, we obtain the following.

Theorem 6.4.

- *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 $\mathcal{L}^6_{\infty,\omega}$, shown in Proposition 4.8. 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-known 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.

[9]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.

Next, consider the following TriAL expressions:

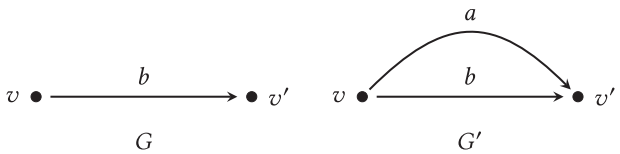
$$E_a := E_G \underset{2=a}{\overset{1,2,3}{\bowtie}} E_G \qquad E_{aux} := E_G \underset{1=1',3=3'}{\overset{1,2,3}{\bowtie}} E_a.$$

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 $b = \mathsf{no\_edge}$.) 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 6.2). 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 6.1. 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. Last, 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 [17, 18]. 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 [10] we obtain:

Corollary 6.5.

- *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*.

Let us note again that TriAL and TriAL* are designed to be used as a basic navigational feature that can be added to any graph query language (such as SPARQL). As such, we do not need nor want to have the power of expressing any conjunctive query or graph pattern, as it would immediately turn query evaluation into an intractable problem. This is why we do not compare against full SPARQL or other, more powerful, navigational graph query languages such as GraphLog [18], Monadically Defined Queries [14, 51], TriQ [7], Regular Queries [48], or other similar languages such as those in, e.g., References [12, 24]. All these languages can indeed express conjunctive queries, which means that they are automatically incomparable or more expressive than TriAL and TriAL*. But it also means that, unlike TriAL and TriAL*, they are meant to serve as a standalone query language for graph databases. On the other hand, one can probably find syntactic restrictions on these languages, on the lines of Theorem 6.4, where the comparison against TriAL* would make more sense. We leave these comparisons as future work.

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 still does not guarantee that having such expressive languages is feasible in practice. To make a case for this claim, we describe in this section 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 stress that our implementation is meant to serve as a proof of concept and not as a standalone system. Indeed, the main focus of this article is not the development of a working tool, but rather 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, the queries used for testing, and the scripts used to generate the synthetic data available to the journal editors, so that they can be sent to the reviewers upon request. We also added the executable version of each query used for the experiments available in the online appendix of this article.

Implementing TriAL*

Our implementation is designed to work on top of any relational database system that supports the `WITH RECURSIVE` feature. 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, which requires recursion. In our experiments, we used PostgreSQL v.9.5.3., so we will consider this engine as our reference point. We shall discuss the performance of recursive queries in detail below. The triplestore is viewed as one ternary table. Note that the user is free to choose a relational database as desired.

The implementation consists of two modules. The first module is a parser that takes a TriAL* expression[10] 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 closures of a ternary relation are concerned, they are rewritten into `WITH RECURSIVE` queries that take into account whether 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.[11]

Example 7.1. As an example of how the parser works, let us show how to rewrite the query

$$((\texttt{yagofacts} \underset{2=\texttt{acted\_in},2'=\texttt{acted\_in},3=3',1\neq1'}{\overset{1,2,1'}{\bowtie}} \texttt{yagofacts}) \underset{1=3',}{\overset{1',2,3}{\bowtie}})^*$$

into SQL queries. In this example, our triplestore is called `yagofacts` (the reason for this will soon become clear) and contains information about movies and actors starring in those movies. For instance, the database contains triples of the form `(Kevin Bacon, actedIn, Unforgiven)`, which tells us that a certain actor starred in a certain movie. The query above then finds all pairs of actors who co-starred in the same movie transitively; i.e., we are finding pairs of actors that have finite collaboration distance. Here the collaboration distance is defined to be equal one if the actors co-starred in the same movie, two if the shortest link between them goes through two movies, and so on. Namely, we are exploring a path starting in a node corresponding to some actor, traverse an `actedIn` labelled edge to reach a movie this actor acted in, and then traverse another `actedIn` edge backwards to find all pairs of actors starring in the same movie (our condition uses $1 = 1'$ so that the answer contains pairs of different actors). The procedure is then repeated transitively using $*$.

Our rewriting proceeds in two steps. First, the innermost expression is rewritten into a (non-recursive) SQL query, and then given the name `actors`:

```
WITH actors AS (
  SELECT yagofacts.s, yagofacts.p, yf2.s
  FROM yagofacts, yagofacts yf2
  WHERE yagofacts.p = 'actedIn' AND
```

[10] For the implementation, we provide a keyboard-friendly syntax resembling SQL's commands.

[11] 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.

```
        yf2.p = 'actedIn' AND
        yagofacts.o = yf2.o AND
        yagofacts.s <> yf2.s
    )
```

Here `yagofacts.s`, `yagofacts.p`, and `yagofacts.o` represent the first, second, and third component of a triple in the `yagofacts` triplestore. Using the `actors` subquery, we can then express the recursive part as follows:

```
WITH RECURSIVE rec(s, p, o) AS (
  SELECT * FROM actors
      UNION
  SELECT rec.s, actors.p, actors.o
  FROM actors, rec
  WHERE actors.s = rec.o
  )
SELECT * FROM rec
```

Clearly, this gives us a closure over the result of the query `actors` shown above.

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 three parts. First, we consider real world RDF data from YAGO [53], a huge knowledge base containing information from Wikipedia, WordNet, and GeoNames. Here, we design several property path queries and nested regular expressions that test various aspects of navigational querying over this dataset. In the second part, we use the GMark framework [9] to compare how TriAL* fares against Apache Jena, a popular SPARQL query engine at the time of processing SPARQL queries that use property paths expressions. We demonstrate that, in general, and specially when multiple regular expressions are used in the same query, the flexibility of the algebra in TriAL* does lead to faster processing times. Finally, we take the three TriAL* specific queries introduced in this article: $\text{Reach}_{\rightarrow}$, $\text{Reach}_{\nearrow}$ and the query reachTA from Example 3.5, and design several synthetic datasets intended to push their performance, and see how they scale.

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.

Real World RDF Data and a Comparison with SPARQL Engines

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

The dataset. We use a piece of YAGO known as YAGOFacts,[12] which contains the core information available in the YAGO database. The dataset contains information about famous people, movies, geographical entities, and so on, and shows how such entities are connected. For instance, some triples state that a particular actor acted in some movie (e.g., (`Kevin Bacon, actedIn, Unforgiven`)), others that a city is located in a particular region or country (e.g.,

[12] See http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/downloads/ for more details.

(Berlin, isLocatedIn, Germany)), and so on. When loaded into Postgres, the dataset was of size 2.4GB and it contains 5.6 million triples.

Queries. For testing our implementation, we selected five navigational queries that appeared in previous literature on RDF querying [29, 49]. 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 Postgres (and comparing it to SPARQL engines), which was previously done in, e.g., References [5, 33]. We therefore focus solely on queries that use non trivial reachability patterns under the Kleene star. The queries are as follows:

(1) **Q1:** This query finds all actors that have a finite Bacon number.[13] One can view this query as a special instantiation of the TriAL* query we used in Section 7 for the case of actors, and with the URI of Kevin Bacon as a starting point.
(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 that 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).
(5) **Q5:** In this query, we test a pattern similar to the one from Example 3.5. Namely, we look for all people with a finite Bacon number, but such that the connection is made through movies that 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 Reference [29], while Q1, Q4, and Q5 come from Reference [49].

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.). To take advantage of the starting point in queries, we extend the language in order to support specifying the starting point of the reachability pattern we want to compute. This 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).

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

[13] 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.

Table 1. Execution Times for Queries Q1 Through Q5 Over the YAGO Dataset

	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

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. Regarding the size of the answers, for Q1 and Q2 the size corresponds roughly to all actors in the database and most geographical entities, respectively, and both are huge numbers. For Q3, the unoptimised version has 34230 tuples in the base relation, but the optimised version has only 448 tuples in the base relation. The final answer is the same 448 tuples. Finally, Q4 and Q5 use the same base relation as Q1 (all people that acted in a movie with Kevin Bacon), but the answers are smaller: just 225 tuples for Q4 and 3 for Q5.

popular SPARQL engines: Open Link Virtuoso[14] version 7.2.1.3214-pthreads (open source edition) and Apache Jena[15] 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*.

Results and discussion. We present the execution times of our queries in Table 1. 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 subquery 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 specifying the starting point may not be best speed-up technique for TriAL* queries, and that proper query planning seems to hold a lot of promise for optimising navigational queries in practice (especially if we can estimate the size of intermediate results). As the main focus of this article 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,

[14] http://virtuoso.openlinksw.com/.
[15] https://jena.apache.org/.

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 recognises 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 running out of memory quite fast for more complex questions, while our implementation of TriAL* uses an RDBMS as a backend, 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. To illustrate the last point further, we now show how the implementation of TriAL* performs against SPARQL engines on benchmarks supporting property path queries.

TriAL* to Compute Property Path Queries

Property paths, as a subset of TriAL* queries, are a good feature for comparing our proof-of-concept implementation against other SPARQL processors. We would have preferred to test our implementation against nSPARQL or other similar recursive extensions for SPARQL, but as far as we are aware property paths are the largest subset of TriAL* that is currently implemented in all of the typical SPARQL systems. Unfortunately, even property paths are difficult to compare with. Most well known SPARQL benchmarks do not include queries with property paths (see, e.g., LUBM [30] or BSBM [13]). We use instead gMark [9], the only benchmark supporting property paths that we are aware of.

The framework in gMark allows users to generate both graph instances and queries. The graphs are generated as triples in a file and can be loaded into any database system, while the queries are generated in four different languages, including SPARQL. There is a wide range of parameters that allows one to define more complex instances of queries; we used the most complex datasets that the framework could produce and for queries we modified the parameters to ensure all queries featured at least one property path expression with the kleene-star operator.

Experiments. We used gMark to build 10 synthetic SPARQL queries, all of them using property paths with at least one kleene-star, so that we would have to use kleene star in TriAL as well. We also used the database generator to build a synthetic dataset with 200.000 nodes, and around 270.000 triples. The gMark framework was only including queries using the `DISTINCT` result modifier of SPARQL (which works just as in SQL). Since our goal is to focus on the path computation features and not a comparison on duplicate elimination in SPARQL versus that of SQL, we also tried a version of each of the 10 gMark queries without the `DISTINCT` operator. All 20 queries were ran over Jena, and we also ran them on our implementation built on top of postgreSQL, as explained in the previous sections.[16]

Results. Table 2 shows the running times of all queries. At first, we see that our concern about the `DISTINCT` operator was founded, as Jena shows a hard time at the time of removing duplicates of queries with huge internal results.

[16]Unfortunately, Virtuoso does not support queries that do not have a specified starting or ending point, so we could not use it in this comparison.

Table 2. Execution Times for Queries QG1 Through QG10 Over the Instance Generated by gMark

	Jena	Jena Distinct	TriAL*	TriAL* Distinct
QG1	8.7845	T	1.4363	2.0937
QG2	15.0968	15.7101	80.1402	81.3911
QG3	8.7546	T	1.6473	1.7960
QG4	31.4198	T	2.2090	2.8047
QG5	9.3979	T	0.7063	0.7473
QG6	9.0621	T	4.0343	4.6067
QG7	8.6981	8.9465	2.5639	2.9365
QG8	13.4507	T	41.7757	42.0853
QG9	45.5639	T	12.6236	12.8000
QG10	9.5007	T	4.0373	4.4833

Jena and TriAL* refer to the tmes on Jena and our implementation, respectively, on queries without the DISTINCT modifier, while Jena Distinct and TriAL* Distinct refer to the times for the original gMark queries featuring the DISTINCT modifier. All times are in seconds, and T stands for timeout.

However, when looking at queries without `DISTINCT`, we find two different patterns.

- Jena is generally faster on queries of the form
  `SELECT * WHERE {?x exp ?y . ?y :p ?z},`
  where `exp` is a property path and `:p` is a single predicate. From the ten queries we report, only QG2 and QG8 are of this form. These queries are translated into a TriAL* expression involving one recursive subquery and one nonrecursive one.
- Our implementation is generally faster on queries that have either only one pattern, or more than one property path, such as
  `SELECT * WHERE {?x exp1 ?y},` or
  `SELECT * WHERE {?x exp2 ?y . ?y :exp3 ?z},`
  where `exp1`, `exp2`, and `exp3` are property paths. All the remaining queries have one of these two forms, and specifically QG4 and QG9 are the only ones of the second form, that is, demanding one join between two property path patterns. These queries are translated into a TriAL* expression involving two recursive subqueries.

The reason why Jena is superior to our implementation on QG2 and QG8 is because we always materialise the results of inner recursive queries. We need to materialise intermediate computations to avoid repeated computations of the same recursive queries (which is what happens when two WITH RECURSIVE clauses are nested in postgreSQL).

On the other hand, our implementation beats Jena on all queries involving just a single property path pattern, and also in queries demanding operations on at least two recursive subqueries. We believe this is a consequence of using a closed language to compute them, as the combination of intermediate results is faster (even if we materialise them) simple, because we stay in SQL all the time.

We remark that these experiments are carried over with a prototype implementation running over SQL, a language that is not specifically tailored at dealing with triples. These results should improve even further if we construct an implementation reusing the common machinery of SPARQL processors such as indexing, optimisation, statistics, and so on. And one could definitely study how

Table 3. Datasets and Running Times for TriAL* Queries Over Synthetic Data

	dataset	size	triples	patterns	time-u (s)	time-sp (s)
Reach$_{\rightarrow}$	D1	572MB	7.14M	7,000	181	43
	D2	1.2GB	15.3M	15,000	1,061	96
	D3	1.7GB	21.4M	20,000	T	131
Reach$_{\nearrow}$	D4	565MB	7.14M	7,000	228	43
	D5	1.2GB	15.3M	15,000	795	93
	D6	1.7GB	21.4M	20,000	T	131
reachTA	D7	567MB	7.19M	2,400	7,097	564
	D8	1.2GB	15.4M	5,000	T	13,354
	D9	1.7GB	21.6M	7,500	T	T

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 that has the staring point specified. The symbol T marks an execution timeout.

to do a better job at combining recursion with other algebraic operators. One could, for example, exploit the connection between TriAL* and ReachTripleDatalog$^{\neg}$ to use some of the well-known datalog optimisation and planning techniques (see, e.g., Reference [34]).

Synthetic Data and TriAL* Queries

So far, we have mostly concentrated our efforts in computing TriAL* expressions that are equivalent to property paths, as this is the only recursive feature that is part of the current SPARQL standard. However, we have seen that the expressive power of TriAL* surpasses that of property paths by a great extent. Thus, in this section, we concentrate in testing our implementation with more expressive recursive queries. For these experiments, we choose to implement our own synthetic data generator, in order to have more control on the complexity of the data with respect to the recursive queries we test.

Queries. We take three queries illustrating the types of navigational patterns that 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 3.5).

Datasets. The RDF datasets used for testing these queries were generated in such a way that the number of patterns that 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 3, 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 3.5. 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. The hardware setup is the same as in the previous experiments.

Results and discussion. The evaluation results are presented in Table 3. 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 $\text{Reach}_{\rightarrow}$ and $\text{Reach}_{\nearrow}$ 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. 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 $\text{Reach}_{\rightarrow}$ and $\text{Reach}_{\nearrow}$ all the runs execute efficiently. There are still some issues with reachTA on larger datasets, which suggests that 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 in practice, 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. We have presented a reasonable case for the utility of algebraically closed languages in real world scenarios, as we were able to beat one of the most well known SPARQL systems when computing the answers of queries that required algebraic operators on top of recursive queries. When pushing our performance over synthetic data and more complicated queries, we see the importance of specifying a starting point: the "well-behaved" queries pose no significant evaluation problems when a starting point is known, although there are some queries that run slow (but given enough time do terminate). On the other hand, when no starting point is specified the problem becomes much more difficult in practice (even though the theoretical bounds are the same). The need for a starting point to filter unnecessary recursive computations has also been recognised in other different languages. For example, postgreSQL assumes the definition of an *inner query* inside WITH RECURSIVE computations, and, as we see by analysing its code, the engine assumes this inner query does specify a starting point for the recursion. Our results suggest that there is a lot of room for improvement when it comes to query planning and optimisation, and we hope to address this issue in future work.

8 RELATED WORK

There is a long history of languages that treat relations of fixed arity, starting with the early philosophical works by DeMorgan and Pierce [44], which were later formalised by Schröder [52], and used substantially in mathematics and computer science. Perhaps the most relevant treatise of those from a database point of view is given in Reference [54], where an algebra of binary relations, called *relation algebra*, is developed and shown equivalent to first order logic with three

variables. And while one could use binary relations to model RDF triplestore in some restricted scenarios (for instance, by adding a new relation for each predicate appearing in the triplestore, we could model edge-labelled graphs), in full generality relation algebra is too restrictive to reason about triples where one element can be a predicate and a subject at the same time (like in, e.g., the triplestore from Figure 2). Furthermore, as we show in Theorem 4.6, the language TriAL goes beyond the power of FO^3 and thus relation algebra. Finally, relation algebra lacks the ability to express recursive queries that form the core of the language TriAL* and provide us with a navigational framework for querying RDF. To overcome some of the limitations of relation algebra in the context of RDF/triples several proposals have appeared throughout the years, the most notable being the *Trirel* language of Reference [50] and the calculus of triads presented in Reference [40].

In Reference [40] Longyear presents an algebra of ternary relations whose design principles are similar to those of TriAL; namely, it is built on the premise that when working with triples one should manipulate only ternary relations, and that each operator of the algebra should always produce ternary relations. On top of basic operators that permute the elements of a ternary relation, Reference [40] also allows existential and universal quantification, and using those defines six different operations that take as input three ternary relations and produce as output a single ternary relation. Three of these operations can be seen as joins of three relations (called products in Reference [40]), and are defined using existential quantifiers over three variables, while the other three use universal quantification in their definition and have no clear database analogue. All of these operations are easily expressible in FO^6 without the equality predicate, and it is thus straightforward to show that they subsume the fragment of TriAL that uses no inequalities, nor complementation. On the other hand, since the triad calculus can not stipulate that two values are different, it is easy to see that in terms of expressive power it is incomparable with full TriAL. Furthermore, similarly to relation algebra, the calculus of triads has no navigational capabilities and can not express even the most basic reachability queries of TriAL*. Apart from these formal differences, the work of Reference [40] has a very different focus than our presentation, and mainly studies simplification rules for the expressions in the calculus of triads.

Reference [50] also presents an algebra of ternary relations, called Trirel, this time designed to operate over RDF triples, and overcoming several deficiencies of Reference [40]. The basic operation of Trireal is a join of three ternary relations called tri-join that again produces a ternary relation, and four different variants of this join are given. On top of this, several other operations and primitives for manipulating ternary relations are given, with the main result of Reference [50] showing that Trirel is equivalent to a fragment of Datalog called Trilog. Similarly as in the case of the calculus of triads, it is straightforward to show that Trirel and TriAL are incomparable in terms of expressive power. Namely, as some variants of tri-join are intrinsically FO^7 operations they can not be expressed in TriAL, and on the other hand, the lack of inequalities between variables in Trilog programs and Trirel makes them incapable of expressing "negative" joins of the form $E \bowtie^{1,2,3'}_{3\neq 1'} E$. Again, as in the case of Reference [40], if we restrict ourselves to purely positive TriAL queries, then one can easily show that Trirel strictly subsumes positive TriAL, as its Datalog analogue is more expressive than TripleDatalog (that uses no negation nor inequalities). Finally, in terms of navigational queries, Trirel can not simulate basic reachability supported in TriAL*, as it is purely relational language.

Overall, while both References [50] and [40] describe frameworks designed on the same premise as TriAL, the languages introduced there are in general incomparable with our proposal. Furthermore, there are stark philosophical differences between TriAL and two approaches, which both use a fixed set of joins between three ternary relations, without explaining why these particular joins are allowed (with 84 possible combinations available), while we opt to allow all possible

joins between two ternary relations. More importantly, the main focus of our work is to support navigational queries through Kleene closures—a feature not supported by neither Reference [40] nor [50].

9 CONCLUSIONS AND FUTURE WORK

As the current approaches for extracting navigational patterns from RDF data are primarily based on graph query languages, in this article, we explore if this tells the whole story. In particular, we identify several types of reachability patterns supported by RDF that 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 that 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 navigational 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 [38], or property graphs, which are the data model used in current graph database systems (see, e.g., the popular Neo4j graph engine [43]). We present some initial results on this in the online appendix 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 that includes data structures that support fast evaluation of navigational queries might be preferred.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. 1995. *Foundations of Databases.* Addison-Wesley.
[2] R. Angles. 2012. A comparison of current graph database models. In *Proceedings of the International Conference on Data Engineering (ICDE'12).* 171–177.
[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2016. Foundations of modern graph query languages. Retrieved from http://arxiv.org/abs/1610.06264.
[4] R. Angles and C. Gutierrez. 2008. Survey of graph database models. *Comput. Surveys* 40, 1 (2008).
[5] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. 2013. Benchmarking database systems for social network applications. In *Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems (GRADES'13).*
[6] K. Anyanwu and A. P. Sheth. 2003. ρ-queries: Enabling querying for semantic associations on the semantic web. In *Proceedings of the 12th International World Wide Web Conference (WWW'03).* 690–699.
[7] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2014. Expressive languages for querying the semantic web. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'14).* 14–26.

[8] M. Arenas and J. Pérez. 2011. Querying semantic web data with SPARQL. In *Proceedings of the Symposium on Principles of Database Systems (PODS'11)*. 305–316.
[9] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869.
[10] P. Barceló, J. Pérez, and J. L. Reutter. 2012. Relative expressiveness of nested regular expressions. In *Proceedings of the American mathematical Society (AMW'12)*. 180–195.
[11] P. Barceló, J. Pérez, and J. L. Reutter. 2013. Schema mappings and data exchange for graph databases. In *Proceedings of the International Conference on Database Theory (ICDT'13)*.
[12] Meghyn Bienvenu, Magdalena Ortiz, and Mantas Simkus. 2015. Navigational queries based on frontier-guarded datalog: Preliminary results. In *Proceedings of the Alberto Mendelzon International Workshop on Foundations of Data Management*. 162.
[13] Christian Bizer and Andreas Schultz. 2009. The Berlin sparql benchmark. (2009).
[14] Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. 2015. Reasonable highly expressive query languages. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI'15)*. 2826–2832.
[15] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. 2000. Containment of conjunctive regular path queries with inverse. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 176–185.
[16] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. 2002. Rewriting of regular expressions and regular path queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 443–465.
[17] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. 2003. Reasoning on regular path queries. *ACM SIGMOD Rec.* 32, 4 (2003), 83–92.
[18] M. Consens and A. O. Mendelzon. 1990. GraphLog: A visual formalism for real life recursion. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems (PODS'90)*. 404–416.
[19] I. Cruz, A. O. Mendelzon, and P. Wood. 1987. A graphical query language supporting recursion. In *Proceedings of the ACM Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD'87)*. 323–330.
[20] P. Cudré-Mauroux and S. Elnikety. 2011. Graph data management systems for new application domains. *Proceedings of the VLDB Endowment (PVLDB)* 4, 12 (2011), 1510–1511.
[21] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33, 3 (2001), 374–425.
[22] 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.
[23] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. 2011. Adding regular expressions to graph reachability and pattern queries. In *Proceedings of the 27th International Conference on Data Engineering (ICDE'11)*. 39–50.
[24] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Dimitri Surinx, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. 2015. Relative expressive power of navigational querying on graphs. *Info. Sci.* 298 (2015), 390–406.
[25] 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 *Proceedings of the International Conference on Database Theory (ICDT'11)*. 197–207.
[26] 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 *Proceedings of the FoIKS*. 124–143.
[27] 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.
[28] 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.
[29] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling kleene: Fast property paths in RDF-3X. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems (GRADES'13)*. 14.
[30] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.: Sci., Serv. Agents WWW* 3, 2 (2005), 158–182.
[31] D. Harel, D. Kozen, and J. Tiuryn. 2000. *Dynamic Logic*. MIT Press.
[32] S. Harris and A. Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. Retrieved from http://www.w3.org/TR/sparql11-query.
[33] Daniel Hernández, Aidan Hogan, Cristian Riveros, Carlos Rojas, and Enzo Zerega. 2016. Querying wikidata: Comparing SPARQL, relational and graph databases. In *Proceedings of the International Semantic Web Conference (ISWC'16)*. 88–103.

[34] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, 1213–1216.
[35] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with property paths. In *Proceedings of the International Semantic Web Conference (ISWC'15)*. 3–18.
[36] L. Libkin. 2004. *Elements of Finite Model Theory*. Springer.
[37] L. Libkin, W. Martens, and D. Vrgoč. 2013. Querying graph databases with XPath. In *Proceedings of the International Conference on Database Theory (ICDT'13)*.
[38] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying graphs with data. *J. ACM* 63, 2 (2016), 14.
[39] Leonid Libkin, Juan L. Reutter, and Domagoj Vrgoč. 2013. Trial for RDF: Adapting graph query languages for RDF data. In *Proceedings of the Symposium on Principles of Database Systems (PODS'13)*. 201–212.
[40] Christopher Longyear. 1972. Towards a triadic calculus, I-III. *J. Cybernet.* (1972), 50–65.
[41] K. Losemann and W. Martens. 2012. The complexity of evaluating path expressions in SPARQL. In *Proceedings of the Symposium on Principles of Database Systems (PODS'12)*. 101–112.
[42] M. Marx. 2005. Conditional XPath. *ACM Trans. Database Syst.* 30, 4 (2005), 929–959.
[43] Neo4j. 2013. Neo4j, The graph database. Retrieved from http://www.neo4j.org/.
[44] Charles Sanders Peirce. 1885. On the algebra of logic: A contribution to the philosophy of notation. *Amer. J. Math.* 7, 2 (1885), 180–196.
[45] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16.
[46] J. Pérez, M. Arenas, and C. Gutierrez. 2010. nSPARQL: A navigational language for RDF. *J. Web Semant.* 8, 4 (2010), 255–270.
[47] Axel Polleres and Johannes Peter Wallner. 2013. On the relation between SPARQL1.1 and answer set programming. *J. Appl. Non-Class. Logics* 23, 1–2 (2013), 159–212.
[48] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2015. Regular queries on graph databases. In *Proceedings of the 18th International Conference on Database Theory (ICDT'15)*, Vol. 31. 177–194.
[49] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. 2015. Recursion in SPARQL. In *Proceedings of the Semantic Web Conference (ISWC'15)*. 19–35.
[50] Edward L. Robertson. 2004. Triadic relations: An algebra for the semantic web. In *Proceedings of the Semantic Web and Databases 2nd International Workshop (SWDB'04)*. 91–108.
[51] S. Rudolph and M. Krötzsch. 2013. Flag & check: Data access with monadically defined queries. In *Proceedings of the Symposium on Principles of Database Systems (PODS'13)*. 151–162.
[52] E. Schröder. 1890. 1905: Vorlesungen über die Algebra der Logik. 3 Bde. (1890).
[53] 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.
[54] A. Tarski and S. Givant. 1987. *A Formalization of Set Theory Without Variables*. AMS.
[55] M. Y. Vardi. 1995. On the complexity of bounded-variable queries. In *Proceedings of the Symposium on Principles of Database Systems (PODS'95)*. 266–276.
[56] D. Vrgoč. 2014. *Querying Graphs with Data*. Ph.D. Dissertation. School of Informatics, University of Edinburgh.
[57] P. T. Wood. 2012. Query languages for graph databases. *Sigmod Rec.* 41, 1 (2012), 50–60.

Received November 2016; revised July 2017; accepted September 2017