Expressive Languages for Querying the Semantic Web

MARCELO ARENAS, Pontificia Universidad Católica de Chile & IMFD, Chile
GEORG GOTTLOB, University of Oxford, UK
ANDREAS PIERIS, University of Edinburgh, UK

The problem of querying RDF data is a central issue for the development of the Semantic Web. The query language SPARQL has become the standard language for querying RDF since its W3C standardization in 2008. However, the 2008 version of this language missed some important functionalities: reasoning capabilities to deal with RDFS and OWL vocabularies, navigational capabilities to exploit the graph structure of RDF data, and a general form of recursion much needed to express some natural queries. To overcome these limitations, a new version of SPARQL, called SPARQL 1.1, was released in 2013, which includes entailment regimes for RDFS and OWL vocabularies, and a mechanism to express navigation patterns through regular expressions. Unfortunately, there are a number of useful navigation patterns that cannot be expressed in SPARQL 1.1, and the language lacks a general mechanism to express recursive queries. To the best of our knowledge, no efficient RDF query language that combines the above functionalities is known. It is the aim of this work to fill this gap. To this end, we focus on a core fragment of the OWL 2 QL profile of OWL 2 and show that every SPARQL query enriched with the above features can be naturally translated into a query expressed in a language that is based on an extension of Datalog, which allows for value invention and stratified negation. However, the query evaluation problem for this language is highly intractable, which is not surprising since it is expressive enough to encode some inherently hard queries. We identify a natural fragment of it, and we show it to be tractable and powerful enough to define SPARQL queries enhanced with the desired functionalities.

CCS Concepts: • **Information systems → Structured Query Language**;

Additional Key Words and Phrases: Semantic web, RDF, SPARQL, query answering, Datalog-based languages

ACM Reference format:
Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2018. Expressive Languages for Querying the Semantic Web. *ACM Trans. Database Syst.* 43, 3, Article 13 (November 2018), 45 pages.
https://doi.org/10.1145/3238304

1 INTRODUCTION

The Resource Description Framework (RDF) is the W3C recommendation data model to represent information about World Wide Web resources. An atomic piece of data in RDF is a *Uniform Resource*

The work of M. Arenas was partially funded by the Fondecyt grant 1161473 and the Instituto Milenio Fundamentos de los Datos. The work of G. Gottlob and A. Pieris was funded by the EPSRC Programme Grant "VADA: Value Added Data Systems – Principles and Architecture" EP/M025268/.
Authors' addresses: M. Arenas, Pontificia Universidad Católica de Chile & IMFD, Avenue Vicuña Mackenna 4860, Santiago, Chile; email: marenas@ing.puc.cl; G. Gottlob, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; email: georg.gottlob@cs.ox.ac.uk; A. Pieris, University of Edinburgh, Informatics Forum, Crichton Street, Edinburgh, EH8 9AB, UK; email: apieris@inf.ed.ac.uk.

0362-5915/2018/11-ART13 $15.00
https://doi.org/10.1145/3238304

Identifier (URI). In the RDF data model, URIs are organized as RDF graphs, that is, labeled directed graphs where node labels and edge labels are URIs. As with any data model designed to model information, the natural problem of querying RDF data has been widely studied. Since its release in 1998, several designs and implementations of RDF query languages have been proposed [21]. In 2004, a first public working draft of a language, called SPARQL, was released by the W3C, which is in fact a graph-matching query language. Since then, SPARQL has been adopted as the standard language for querying the Semantic Web, and in 2008 it became a W3C recommendation.[1]

One of the distinctive features of Semantic Web data is the existence of vocabularies with predefined semantics: the *RDF Schema (RDFS)*[2] and the *Web Ontology Language (OWL)*[3], which can be used to derive logical conclusions from RDF graphs. Moreover, it has been recognized that navigational capabilities are of fundamental importance for data models with an explicit graph structure such as RDF [2, 5, 7, 20, 35], and, more generally, it is well-accepted that a general form of recursion is a central feature for a graph query language [7, 29, 39]. Therefore, it would be desirable to have an RDF query language equipped with reasoning capabilities to deal with the RDFS and OWL vocabularies, as well as a general mechanism to express recursive queries. Unfortunately, the 2008 version of SPARQL missed the above crucial functionalities. To overcome these limitations, a new version, called SPARQL 1.1 [25], was released in 2013, which includes entailment regimes for RDFS and OWL vocabularies, and a mechanism to express navigation patterns through regular expressions. However, it has already been observed that there exist some very natural queries that require a more general form of recursion and cannot be expressed in SPARQL 1.1 [29, 39].

1.1 Research Challenge

To the best of our knowledge, before the conference papers [4, 23], on which the present article is based, no RDF query language that combines all the above functionalities was known. This work aims at bridging the gap between RDF query languages and the desired functionalities, that is, reasoning capabilities and a general mechanism to express recursive queries. In particular, our ultimate goal is to propose an expressive query language that supports these features, and which can also be evaluated efficiently. Interestingly, Datalog with stratified negation [1, 17] has been shown to be expressive enough to represent every SPARQL query [2, 3, 5, 36, 40]. Thus, it has been used as a natural platform for SPARQL extensions with richer navigation capabilities and recursion mechanisms [29, 39]. Moreover, some extensions of Datalog with existential quantification in rule-heads are appropriate to encode some inferencing mechanisms in OWL [13].

From the above discussion, we can conclude that Datalog and some of its extensions (in particular, the members of the Datalog$^{\pm}$ family of knowledge representation and query languages [14]) appear to be suitable for our purposes. However, for the language obtained by extending Datalog with existential quantification, the query evaluation problem is undecidable (this is implicit in [8]). In fact, the undecidability holds even in the case of *data complexity*, i.e., when the input query is fixed, and only the extensional database (or the RDF graph) is considered as part of the input [13]. It is thus a very important and challenging task to single out an expressive RDF query language that

(1) is based on Datalog, which enables a modular rule-based style of writing queries;
(2) is expressive enough for being useful in real Semantic Web applications, and in particular to support reasoning and navigational capabilities, as well as a general form of recursion;
(3) ensures the decidability of the query evaluation problem; and

[1] http://www.w3.org/TR/rdf-sparql-query.
[2] http://www.w3.org/TR/rdf-schema.
[3] http://www.w3.org/TR/owl-features/.

(4) has good complexity properties in the case the input query is fixed—this is of fundamental importance, as a low data complexity is considered to be a key condition for a query language to be useful in practice.

1.2 Triple Query Language

A first attempt to design a Datalog-based RDF query language that fulfills the above desiderata, focusing on the profile OWL 2 QL of OWL 2, was made in [4]. The proposed language, called TriQ-Lite,[4] is based on Datalog$^{\exists,\neg s,\bot}$, that is, Datalog extended with existential quantification in rule-heads, stratified negation, and negative constraints expressed by using the symbol $\bot$ (false) in rule-heads. Unfortunately, TriQ-Lite suffers from a serious drawback, which may revoke its advantage as an expressive RDF query language, namely, it is not a *plain* language. We call a rule-based query language plain if it allows the user to express a query as a single program in a simple non-composite syntax. An example of a plain query language is Datalog itself, where the user simply needs to define a single Datalog program that captures the intended query. The property of plainness provides conceptual simplicity, which is considered to be a key condition for a query language to be useful in practice. Although TriQ-Lite is based on an extension of Datalog, the way its syntax and semantics are defined significantly deviates from the standard way of defining Datalog-like languages, and thus does not inherit the plainness of Datalog. TriQ-Lite is a composite language, where the user is forced to split the query program into several programs $\Pi_1, \ldots, \Pi_n$ so that each Π_i can be expressed by the fragment of Datalog$^{\exists,\neg s,\bot}$ underlying TriQ-Lite, while each pair (Π_i, Π_{i+1}) is bridged via a set Q_i of conjunctive queries. In view of the conceptual weakness of TriQ-Lite discussed above, the new version of it, dubbed TriQ-Lite 1.0, was introduced in [23]. TriQ-Lite 1.0, which is the main focus of this journal article, is a plain language based on Datalog$^{\exists,\neg s,\bot}$ that fulfills all the crucial desiderata discussed above.

1.3 Summary of Contributions

Our contributions can be summarized as follows:

(1) We introduce in Section 4 the language TriQ 1.0, which is a plain query language based on Datalog$^{\exists,\neg s,\bot}$. We show that this language is expressive enough for encoding some useful but costly queries; e.g., whether a graph contains a clique of size $k > 0$. We then proceed to show that the query evaluation problem for TriQ 1.0 is ExpTime-complete in data complexity.

(2) We show that TriQ 1.0 is expressive enough to deal with SPARQL queries over a relevant fragment of the OWL vocabulary. More precisely, we focus in Section 5 on a profile of OWL, called OWL 2 QL, that is designed to be used in applications where query answering is the most important reasoning task. In particular, we consider a fragment of OWL 2 QL that contains its core functionalities, called OWL 2 QL core. This fragment corresponds to the well-established description logic DL-Lite$_{\mathcal{R}}$ [16], which is essentially the logical underpinning of OWL 2 QL. Then we prove that every SPARQL query under the entailment regime for OWL 2 QL core, which is inherited from the entailment regime for OWL 2 QL [22, 28], can be translated into a TriQ 1.0 query. Moreover, we show in Section 5 that the use of TriQ 1.0 allows us to formulate SPARQL queries in a simpler way, as a more natural entailment regime described in that section can be easily defined by using this query language.

[4]This language is the lite version of a highly expressive language called TriQ, which stands for triple query language, also introduced in [4].

(3) Given the high data complexity of the query evaluation problem for TriQ 1.0, we investigate in Section 6 whether the results proved in Section 5 can also be obtained for a tractable sublanguage of this query language. More precisely, we identify a natural restriction on TriQ 1.0 queries that gives rise to a language, called TriQ-Lite 1.0, with the desired properties. In particular, we prove that the query evaluation problem for this language is PTime-complete in data complexity. We also show in Section 6 that TriQ-Lite 1.0 is a (nearly) maximal tractable sublanguage of TriQ 1.0 in the sense that the mildest relaxation of the condition posed on TriQ 1.0 (in order to obtain TriQ-Lite 1.0) that one can think of, leads to a language for which the query evaluation problem is ExpTime-hard in data complexity.

(4) A key advantage of TriQ-Lite 1.0 is the fact that, whenever the user wants to pose a new query over an RDF graph, (s)he does not need to modify the part of the query program that encodes the OWL 2 QL ontology. In Section 7, we show that this favorable behavior cannot be achieved if we consider Datalog$^{\neg s,\perp}$. In particular, we introduce a novel notion of expressiveness that allows us to collect the queries that can be answered via a fixed program, and we show that TriQ-Lite 1.0 is more expressive than Datalog$^{\neg s,\perp}$ under this notion.

The organization of the article is described in the summary of our contributions. Note that in Section 2 we give a series of examples that motivate our query languages, the notation used in the article is introduced in Section 3, and some concluding remarks are given in Section 8.

2 MOTIVATING SCENARIOS AND QUERIES

The goal of this section is to expose some of the difficulties and limitations encountered when querying RDF data with SPARQL, which motivated us to design an RDF query language based on Datalog. To this end, assume that G_1 is an RDF graph consisting of

(dbUllman, is_author_of, "The Complete Book"),
(dbUllman, name, "Jeffrey Ullman").

The first triple indicates that the object with URI dbUllman is one of the authors of the book "The Complete Book," while the second triple indicates that the name of dbUllman is "Jeffrey Ullman." To retrieve the list of authors occurring in G_1, we can use the following SPARQL query:

```
SELECT ?X
WHERE {
    ?Y is_author_of ?Z .
    ?Y name ?X }.
```
(1)

Note that variables start with the symbol ? in this query. Moreover, the expression ?Y is_author_of ?Z represents a triple pattern that is used to retrieve the pairs (a, b) of elements from G_1, which are stored in the variables ?Y and ?Z, such that a is an author of b. In the same way, the expression ?Y name ?X also represents a triple pattern that is used to retrieve the pairs (a, c) of elements from G_1, which are stored in the variables ?Y and ?X, such that c is the name of a. Finally, the symbol . (dot) is used as a separator of the triple patterns, whose results have to be joined when computing the answer to the query, and SELECT ?X indicates that we are interested in the values stored in the variable ?X.

In the query language proposed in this article, we assume that a predicate triple$(\cdot, \cdot, \cdot)$ is used to store the triples of an RDF graph. Thus, query (1) can be formulated in our language as follows:

$$\text{triple}(?Y, \text{is\_author\_of}, ?Z), \text{triple}(?Y, \text{name}, ?X) \rightarrow \text{query}(?X). \tag{2}$$

The possibility of returning an RDF graph as the answer to a SPARQL query is considered as a fundamental feature [25, 37]. For this reason, one can use the CONSTRUCT operator in order to produce an RDF graph as the output of a query. For example, the following query constructs an RDF graph consisting of triples $(a, \text{name\_author}, b)$, where a is the name of an author of b:

CONSTRUCT { ?X name_author ?Z }
WHERE {
?Y is_author_of ?Z .
?Y name ?X }.

The expression ?X name_author ?Z represents a triple pattern specifying which RDF triples are to be included in the output. Hence, the result of evaluating this query over G_1 is the RDF graph

(“Jeffrey Ullman”, name_author, “The Complete Book”).

In our language, the user is not forced to learn about a new operator in order to produce an RDF graph as output. (S)he can simply replace in (2) the predicate query$(\cdot)$ by the predicate triple$(\cdot, \cdot, \cdot)$ in order to produce an RDF graph:

$$\text{triple}(?Y, \text{is\_author\_of}, ?Z), \text{triple}(?Y, \text{name}, ?X) \rightarrow \text{triple}(?X, \text{name\_author}, ?Z). \tag{3}$$

Note that the CONSTRUCT operator in SPARQL is not recursive; to evaluate a query containing this operator, first the body of the query has to be evaluated to produce assignments for the variables, and then these assignments are used in the template of the CONSTRUCT operator to produce an RDF graph. In the same way, the rule (3) may appear recursive but a resulting tuple triple$(a, \text{name_author}, b)$ of this rule cannot be used in the body of (3) to produce new tuples, given that triple$(a, \text{name_author}, b)$ cannot be matched against any of the tuples in the body of (3).

The use of the operator CONSTRUCT in SPARQL allows one to have compositionality; the output of a query can be used as the input of another query. This is a fundamental property, which plays a crucial role when adding a recursion mechanism to SPARQL [38]. Notice that our language inherits the compositionality of Datalog, so that a recursion mechanism can be easily introduced without needing additional syntactic constructs.

Assume now that G_2 is an RDF graph extending G_1 with the following triples:

(dbAho, is_coauthor_of, dbUllman),
(dbAho, name, “Alfred Aho”).

The query language SPARQL allows the use of blank nodes in the CONSTRUCT operator to include some anonymous resources in an RDF graph. For example, a blank node is used in the following query to indicate that if a is a co-author of b, then there must be some publication c such that a and b are both authors of c.

CONSTRUCT { ?X is_author_of _:B . ?Y is_author_of _:B }
WHERE { ?X is_coauthor_of ?Y }. (4)

In the above query, _:B is a blank node, while ?X is_author_of _:B and ?Y is_author_of _:B specify the triples to be constructed for every possible match of the variables ?X and ?Y. The semantics

of SPARQL imposes the restriction that a fresh blank node has to be used for each match of the variables $?X$ and $?Y$. Although this constraint is natural in this case, this is yet another feature of SPARQL that the user needs to remember when formulating a query. In our case, we do not need to add extra notation for the creation of anonymous resources, as our query language allows existential quantification in the head of the rules:

$$\text{triple}(?X, \text{is\_coauthor\_of}, ?Y) \rightarrow \exists ?Z\, \text{triple}(?X, \text{is\_author\_of}, ?Z), \text{triple}(?Y, \text{is\_author\_of}, ?Z).$$

Moreover, our query language can be used to lift the restriction that blank nodes are used only locally. For example, our query language can be used to anonymize the subjects of the triples in an RDF graph, by replacing every URI in the subject position of a triple by a blank node:

$$\begin{aligned} \text{triple}(?X, ?Y, ?Z) &\rightarrow \text{subj}(?X), \\ \text{subj}(?X) &\rightarrow \exists ?Y\, \text{bn}(?X, ?Y), \\ \text{triple}(?X, ?Y, ?Z), \text{bn}(?X, ?U) &\rightarrow \text{output}(?U, ?Y, ?Z). \end{aligned}$$

The first rule is used to store in the predicate $\text{subj}(\cdot)$ the URIs mentioned in the subject of the triples of an RDF graph. The second rule creates a blank node for every URI in the predicate $\text{subj}(\cdot)$, which is stored in the predicate $\text{bn}(\cdot, \cdot)$. Finally, the third rule replaces in the predicate $\text{triple}(\cdot, \cdot, \cdot)$ every URI in the subject position by its associated blank node, producing an RDF graph in the predicate $\text{output}(\cdot, \cdot, \cdot)$. The ability to anonymize the subjects of an RDF graph is a useful feature as it can allow publishing data without leaking sensitive information. It is important to note that such a query cannot be expressed by using the local semantics of blank nodes in the CONSTRUCT operator of SPARQL, as the same blank node identifying a specific resource in an RDF graph has to be used every time this resource is considered in the result of the query.

Query (4) encodes some prior knowledge about the co-authorship relation. This type of knowledge can be explicitly encoded in an RDF graph by using the RDFS and OWL vocabularies. As an example of this, assume that G_3 is an RDF graph extending G_2 with the following triples:

$$\begin{aligned} &(\mathrm{r}_1, \text{rdf:type}, \text{owl:Restriction}), && (\mathrm{r}_2, \text{rdf:type}, \text{owl:Restriction}), \\ &(\mathrm{r}_1, \text{owl:onProperty}, \text{is\_coauthor\_of}), && (\mathrm{r}_2, \text{owl:onProperty}, \text{is\_author\_of}), \\ &(\mathrm{r}_1, \text{owl:someValuesFrom}, \text{owl:Thing}), && (\mathrm{r}_2, \text{owl:someValuesFrom}, \text{owl:Thing}), \\ & && (\mathrm{r}_1, \text{rdfs:subClassOf}, \mathrm{r}_2). \end{aligned} \tag{5}$$

In G_3, the URIs with prefix rdfs: are part of the RDFS vocabulary, while the URIs with prefix owl: are part of the OWL vocabulary. The first three triples of G_3 define r_1 as the class of URIs a for which there exists a URI b such that $(a, \text{is\_coauthor\_of}, b)$ holds, while the following three triples of this graph define r_2 as the class of URIs a for which there exists a URI b such that the triple $(a, \text{is\_author\_of}, b)$ holds. Finally, the last triple of G_3 indicates that r_1 is a subclass of r_2.

The above set of triples states that for every two elements a and b such that $(a, \text{is\_coauthor\_of}, b)$ holds, it must be the case that a is an author of some publication. Thus, if we want to retrieve the list of authors mentioned in G_3, then we expect to find dbAho in this list. However, the answer to the SPARQL query (1) over G_3 does not include this URI, and we are forced to encode the semantics of the RDFS and OWL vocabularies in the query. In fact, even if we try to obtain the right answer by using SPARQL 1.1 under the entailment regimes for these vocabularies, we are forced by the

restrictions of the language [22] to use a query of the form

SELECT ?X
WHERE {
?Y name ?X .
?Y rdf : type ?Z .
?Z rdf:type owl:Restriction .
?Z owl:onProperty is_author_of .
?Z owl:someValuesFrom owl:Thing }.

This query is obtained from (1) by replacing the expression ?Y is_author_of ?Z by the last four triples above, which explicitly state that we are looking for the objects that are authors of some publication (i.e., the objects of type r_2). It is apparent that the resulting query is quite complex. In our query language such complications can be avoided by using rules encoding the semantics of the RDFS and OWL vocabularies. For example, the following rule specifies the semantics of the owl:onProperty primitive of OWL:

$$\begin{aligned}&\text{triple}(?X, \text{rdf} : \text{type}, ?Y),\\ &\text{triple}(?Y, \text{rdf:type}, \text{owl:Restriction}),\\ &\text{triple}(?Y, \text{owl:onProperty}, ?Z),\\ &\text{triple}(?Y, \text{owl:someValuesFrom}, ?U) \rightarrow \exists ?W\ \text{triple}(?X, ?Z, ?W).\end{aligned}$$

Notice that a fixed set of rules is used to encode the semantics of the RDFS and OWL vocabularies. If such rules are available as a library, then the user just has to include them in order to answer queries, without needing to have prior knowledge about the semantics and inference rules for the respective vocabulary. For example, if these rules have been included, then to retrieve the list of authors mentioned in G_3 we can use query (1) again, as initially expected.

Consider now the fact that it is very common in the Web to have several URIs for the same object. For example, the following are URIs of Jeffrey Ullman in DBpedia (the RDF version of Wikipedia) and the semantic knowledge base YAGO:

http://dbpedia.org/resource/Jeffrey_Ullman,
http://yago-knowledge.org/resource/Jeffrey_Ullman,

respectively. To alleviate the issue of having pieces of information about the same object that use distinct URIs for it, the OWL vocabulary includes the primitive owl:sameAs to indicate that two URIs represent the same element. For example, this primitive is used in the following RDF graph G_4 to indicate that dbUllman and yagoUllman are URIs for the same object:

(dbUllman, is_author_of, "The Complete Book"),
(dbUllman, owl:sameAs, yagoUllman),
(yagoUllman, name, "Jeffrey Ullman").

Assume now that we want to retrieve the list of authors mentioned in G_4. If we try to use again the SPARQL query (1), then we obtain the empty answer as the semantics of owl:sameAs is not

taken into consideration. To solve this problem, one has to use the following query:

$$
\begin{array}{l}
\text{SELECT } ?X \\
\text{WHERE } \{ \\
\quad \{ ?Y \text{ is\_author\_of } ?Z \ . \\
\quad\ \ ?Y \text{ name } ?X \ \} \\
\quad \text{UNION} \\
\quad \{ ?Y \text{ is\_author\_of } ?Z \ . \\
\quad\ \ ?Y \text{ owl:sameAs } ?W \ . \\
\quad\ \ ?W \text{ name } ?X \ \} \\
\quad \}.
\end{array}
\tag{6}
$$

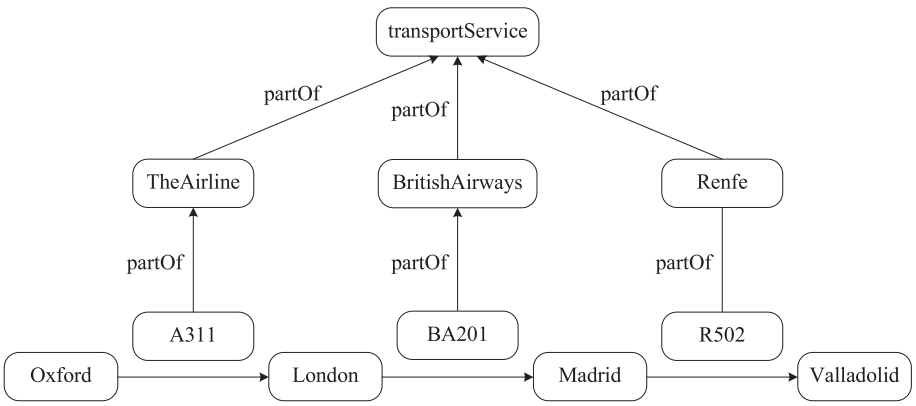
In this query, the operator UNION is used to obtain the union of the results of two queries, and the query occurring after this operator is used to encode the semantics of the owl:sameAs primitive. Therefore, as in the previous example, the user is forced to encode the semantics of the OWL vocabulary in the SPARQL query. Moreover, as the reader may has already noticed, the situation gets worse if we combine the triples in the graphs G_3 and G_4. Fortunately, all these problems can be easily solved in our framework by just incorporating a fixed set of rules encoding the semantics of the primitive owl:sameAs, which includes rules like the following:

$$\text{triple}(?X, \text{owl:sameAs}, ?Y), \text{triple}(?Y, \text{owl:sameAs}, ?Z) \rightarrow \text{triple}(?X, \text{owl:sameAs}, ?Z),$$

$$\text{triple}(?X_1, \text{owl:sameAs}, ?X_2), \text{triple}(?Y_1, \text{owl:sameAs}, ?Y_2), \text{triple}(?X_1, ?U, ?Y_1) \rightarrow \text{triple}(?X_2, ?U, ?Y_2).$$

If this fixed set of rules has been included, then to retrieve the list of authors mentioned in G_4 we can just use query (1) again.

As a final example, consider the following scenario from [29]:



In the above RDF graph, we have some transport services between cities. For example, the triples (TheAirline, partOf, transportService), (A311, partOf, TheAirline), (Oxford, A311, London) indicate that TheAirline is a transport service, A311 is a specific service provided by TheAirline, and A311 goes from Oxford to London, respectively. In this case, we would like to pose a query retrieving the pairs a, b of cities such that there is a way to travel from a to b. As shown in [29, 39], such a query cannot be expressed with the navigation mechanism of SPARQL 1.1, as it requires

navigating simultaneously in two different directions: the path of transport services from a to b can be of arbitrary length, and the paths necessary to check that we are connecting cities by transport services could also be of arbitrary length. For instance, in the RDF graph depicted in the figure, to check whether we can go from Oxford to Valladolid we need to follow a path of length three, and to check that A311 is a transport service we need to follow a path of length two to reach the node transportService. Notice that such paths could be of arbitrary length, as it could be necessary to use more than three transport services to go from Oxford to Valladolid, and the path from A311 to transportService could include some additional triples such as (TheAirline, partOf, busService) to indicate that TheAirline is a bus service, and likewise for BA201 and R502. On the other hand, the general recursion mechanism of the query language proposed in this article can be easily used to express this query. More specifically, we first use the following rules to collect all the transport services in an RDF graph:

$$\begin{aligned} \text{triple}(?X, \text{partOf}, \text{transportService}) &\rightarrow \text{ts}(?X), \\ \text{triple}(?X, \text{partOf}, ?Y), \text{ts}(?Y) &\rightarrow \text{ts}(?X). \end{aligned}$$

Then, the following rules collect all the pairs of connected cities:

$$\begin{aligned} \text{ts}(?T), \text{triple}(?X, ?T, ?Y) &\rightarrow \text{query}(?X, ?Y), \\ \text{ts}(?T), \text{triple}(?X, ?T, ?Z), \text{query}(?Z, ?Y) &\rightarrow \text{query}(?X, ?Y). \end{aligned}$$

3 DEFINITIONS AND BACKGROUND

Assume there are pairwise disjoint infinite countable sets $\mathbf{U}$, $\mathbf{B}$, $\mathbf{V}$. The elements of $\mathbf{U}$ are called URIs, the elements of $\mathbf{B}$ are called blank nodes, and the elements of $\mathbf{V}$ are called variables and are assumed to start with the symbol ?. The sets $\mathbf{U}$ and $\mathbf{B}$ are used when defining both RDF graphs and relational databases, and we also refer to them as constants and (labeled) nulls, respectively. Henceforth, for brevity, given two integers n, m such that $n \geq m$, we write $[m, n]$ for the set $\{m, m+1, \ldots, n\}$.

3.1 RDF and the Query Language SPARQL

A triple $(s, p, o) \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$ is called an *RDF triple*. In this tuple, s is the *subject*, p is the *predicate*, and o is the *object*. An *RDF graph* is a finite set of RDF triples.[5] SPARQL is essentially a graph-matching query language. Roughly speaking, a SPARQL query is a complex RDF graph pattern that may include RDF triples with variables, conjunctions, disjunctions, optional parts, and constraints over the values of the variables. The evaluation of a SPARQL query P against an RDF graph G is done by matching P against G in order to obtain a set of bindings for the variables in P. The formal syntax and semantics of SPARQL follow.

Syntax of SPARQL Graph Patterns. We adopt the algebraic formalization of SPARQL proposed in [34], using binary operators AND, UNION, OPT, and FILTER. We start by defining the notion of SPARQL *built-in condition*, which is used in filter expressions. Formally,

(1) If $?X, ?Y \in \mathbf{V}$ and $c \in \mathbf{U}$, then $?X = c$, $?X = ?Y$ and bound$(?X)$ are (atomic) built-in conditions.
(2) If R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions.

[5]RDF triples can also include literals and blank nodes. The former represent actual values, such as integers, real number, and dates, while the latter represent anonymous objects. Given the way these elements are treated in SPARQL [27], we do not include them in RDF graphs as our results can be established even if these elements are explicitly considered.

Then the set of (SPARQL) *graph patterns* is defined recursively as follows:

(1) A set $\{t_1, \dots, t_n\}$, where every $t_i \in (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V})$ $(1 \leq i \leq n)$, is a graph pattern (called a basic graph pattern).
(2) If P_1 and P_2 are graph patterns, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ UNION } P_2)$, $(P_1 \text{ OPT } P_2)$ are graph patterns.
(3) If P is a graph pattern and R is a SPARQL built-in condition, then $(P \text{ FILTER } R)$ is a graph pattern.
(4) If P is a graph pattern and W is a finite set of variables, then $(\text{SELECT } W\ P)$ is a graph pattern.

From now on, given a graph pattern P, we define $\text{var}(P)$ as the set of variables occurring in P, and likewise for $\text{var}(R)$ for a built-in condition R. Moreover, we assume that for every graph pattern $(P \text{ FILTER } R)$, it holds that $\text{var}(R) \subseteq \text{var}(P)$. Finally, we usually omit curly brackets in singleton basic graph patterns, i.e., we replace $\{t\}$ by t, where $t \in (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V})$.

Semantics of SPARQL Graph Patterns. To define the semantics of SPARQL, we need to introduce some extra terminology. A *mapping* μ is a partial function $\mu : \mathbf{V} \to \mathbf{U}$. Abusing notation, for a basic graph pattern $P = \{t_1, \dots, t_n\}$, we denote by $\mu(P)$ the basic graph pattern obtained by replacing the variables occurring in P according to μ. The *domain* of μ, denoted by $\text{dom}(\mu)$, is the subset of $\mathbf{V}$ where μ is defined. Two mappings μ_1 and μ_2 are *compatible*, denoted by $\mu_1 \sim \mu_2$, when for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it is the case that $\mu_1(?X) = \mu_2(?X)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping. Intuitively, μ_1 and μ_2 are compatible if μ_1 can be extended with μ_2 to obtain a new mapping, and vice versa. We use the symbol $\mu_\varnothing$ to represent the mapping with empty domain (which is compatible with any other mapping). Moreover, given a mapping μ and a set of variables W, the *restriction of* μ *to* W, denoted by $\mu_{|W}$, is a mapping such that $\text{dom}(\mu_{|W}) = (\text{dom}(\mu) \cap W)$ and $\mu_{|W}(?X) = \mu(?X)$ for every $?X \in (\text{dom}(\mu) \cap W)$. Finally, given a function $h : \mathbf{B} \to \mathbf{U}$, we denote by $h(P)$ the basic graph pattern obtained from P by replacing the blank nodes occurring in P according to h.

To define the semantics of graph patterns, we first need to introduce the notion of satisfaction of a built-in condition by a mapping, and then we need to introduce some operators for mappings. More precisely, given a mapping μ and a built-in condition R, we say that μ *satisfies* R, denoted by $\mu \models R$, if one of the following holds:

(1) R is $\text{bound}(?X)$ and $?X \in \text{dom}(\mu)$.
(2) R is $?X = c$, $?X \in \text{dom}(\mu)$ and $\mu(?X) = c$.
(3) R is $?X = ?Y$, $?X, ?Y \in \text{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$.
(4) R is $(\neg R_1)$, R_1 is a built-in condition, and it is not the case that $\mu \models R_1$.
(5) R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in conditions, and $\mu \models R_1$ or $\mu \models R_2$.
(6) R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in conditions, and $\mu \models R_1$ and $\mu \models R_2$.

Moreover, given sets Ω_1 and Ω_2 of mappings, the *join* of, the *union* of, the *difference* between, and the *left outer join* between Ω_1 and Ω_2 are defined as follows:

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\},\\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\
\Omega_1 \setminus \Omega_2 &= \{\mu \in \Omega_1 \mid \text{for every } \mu' \in \Omega_2 : \ \mu \nsim \mu'\},\\
\Omega_1 ⟕ \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2).
\end{aligned}$$

We are now ready to define the semantics of graph patterns as a function $[\![\cdot]\!]_G$, which takes a pattern expression and returns a set of mappings. The *evaluation* of a graph pattern P over an RDF graph G, denoted by $[\![P]\!]_G$, is recursively defined as follows:

(1) If P is a basic graph pattern, then $[\![P]\!]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(P)$ and there exists $h : \mathbf{B} \to \mathbf{U}$ such that $\mu(h(P)) \subseteq G\}$.
(2) If P is (P_1 AND P_2), then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.
(3) If P is (P_1 UNION P_2), then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$.
(4) If P is (P_1 OPT P_2), then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.
(5) If P is (P_1 FILTER R), then $[\![P]\!]_G = \{\mu \mid \mu \in [\![P_1]\!]_G$ and $\mu \models R\}$.
(6) If P if (SELECT W P_1), then $[\![P]\!]_G = \{\mu_{|_W} \mid \mu \in [\![P_1]\!]_G\}$.

3.2 Relational Databases and Datalog$^{\exists,\neg s,\bot}$ Queries

A *term* t is a constant ($t \in \mathbf{U}$), labeled null ($t \in \mathbf{B}$), or variable ($t \in \mathbf{V}$). An *atom* has the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate, and $t_1, \ldots, t_n$ are terms. A *position* $p[i]$ identifies the i-th attribute of a predicate p. We denote the *arity* of p by arity(p). For an atom $\underline{a}$, we denote by dom($\underline{a}$) and var($\underline{a}$) the sets of its terms and its variables, respectively; these notations extend to sets of atoms. We refer to the predicate of an atom $\underline{a}$ by pred($\underline{a}$). An *instance* I is a (possibly infinite) set of atoms $p(\mathbf{t})$, where $\mathbf{t}$ is a tuple of constants and labeled nulls. A *database* D is a finite instance where only constants occur; we refer to the constants in D as dom(D).

One of the most prominent languages for querying relational data is Datalog, which actually adds recursion to the relational algebra. The query languages that we are going to propose in this work are based on an extension of Datalog, and in particular on Datalog$^{\exists,\neg s,\bot}$, that is, the extension of Datalog with existentially quantified variables ($\exists$), stratified negation ($\neg s$), and the truth constant false ($\bot$). The formal syntax and semantics of Datalog$^{\exists,\neg s,\bot}$ follow.

Syntax of Datalog$^{\exists,\neg s,\bot}$. We start by introducing the syntax of Datalog$^{\exists,\neg}$, that is, the extension of Datalog with existential quantification in the head, and negation in the body. A Datalog$^{\exists,\neg}$ rule ρ is an expression of the form[6]

$$\underline{a}_1, \ldots, \underline{a}_n, \neg \underline{b}_1, \ldots, \neg \underline{b}_m \;\rightarrow\; \exists ?Y_1 \ldots \exists ?Y_k \; \underline{c},$$

where

(1) $n \geq 1$ and $m, k \geq 0$;
(2) every $\underline{a}_i$ ($1 \leq i \leq n$) and $\underline{b}_i$ ($1 \leq i \leq m$) is an atom with terms from ($\mathbf{U} \cup \mathbf{V}$);
(3) $\text{var}(\{\underline{b}_1, \ldots, \underline{b}_m\}) \subseteq \text{var}(\{\underline{a}_1, \ldots, \underline{a}_n\})$;
(4) $\{?Y_1, \ldots, ?Y_k\} \cap \text{var}(\{\underline{a}_1, \ldots, \underline{a}_n, \underline{b}_1, \ldots, \underline{b}_m\}) = \varnothing$; and
(5) $\underline{c}$ is an atom with terms from $(\mathbf{U} \cup \{?Y_1, \ldots, ?Y_k\} \cup \text{var}(\{\underline{a}_1, \ldots, \underline{a}_n\}))$.

The set $\{\underline{a}_1, \ldots, \underline{a}_n\}$ is denoted by body$^+(\rho)$, while $\{\underline{b}_1, \ldots, \underline{b}_m\}$ is denoted by body$^-(\rho)$. The *body* of ρ, denoted by body(ρ), is defined as (body$^+(\rho) \cup$ body$^-(\rho)$). The atom $\underline{c}$ is the *head* of ρ, denoted by head(ρ). A *Datalog$^{\exists,\neg}$ program* Π is a finite set of Datalog$^{\exists,\neg}$ rules. Let sch(X), where X is either a program or a set of atoms, be the set of predicates occurring in X. A *stratification* of Π is a function $\mu : \text{sch}(\Pi) \to [0, \ell]$ such that, for each $\rho \in \Pi$ with $p = \text{pred}(\text{head}(\rho))$: (1) $\mu(p) \geq \mu(p')$, for each $p' \in \text{sch}(\text{body}^+(\rho))$; and (2) $\mu(p) > \mu(p')$, for each $p' \in \text{sch}(\text{body}^-(\rho))$. For each $i \in [0, \ell]$, let $\Pi_i = \{\rho \mid \rho \in \Pi$ and $\mu(p) = i\}$. We say that Π is *stratified* if there exists a stratification of Π. A *constraint* ν is an assertion of the form

$$\underline{a}_1, \ldots, \underline{a}_n \;\rightarrow\; \bot,$$

where $n \geq 1$ and every $\underline{a}_i$ ($1 \leq i \leq n$) is an atom with terms from $\mathbf{U} \cup \mathbf{V}$. The *body* of ν, denoted body(ν), is the set $\{\underline{a}_1, \ldots, \underline{a}_n\}$. A *Datalog$^{\exists,\neg,\bot}$ program* Π is a finite set of Datalog$^{\exists,\neg}$ rules and

[6]For the sake of brevity, in the rest of the article we may write rules with more than one atom in the head. This is not a problem as such rules can be transformed into an equivalent set of rules with just one head-atom; see, e.g., [15].

constraints. We denote by $\text{ex}(\Pi)$ the set of Datalog$^{\exists,\neg}$ rules in Π; in other words, $\text{ex}(\Pi)$ is obtained from Π by dropping the constraints. We say that Π is *stratified* if $\text{ex}(\Pi)$ is stratified. A *stratified Datalog*$^{\exists,\neg,\bot}$ *query* Q is a pair (Π, p), where Π is a stratified Datalog$^{\exists,\neg,\bot}$ program, and $p \in \text{sch}(\Pi)$ does not occur in the body of a rule of Π. For brevity, we write Datalog$^{\exists,\neg s,\bot}$ for stratified Datalog$^{\exists,\neg,\bot}$ programs and queries. Moreover, a supra-index can be removed from Datalog$^{\exists,\neg s,\bot}$ to indicate that the corresponding feature is disallowed. For example, in a Datalog$^{\neg s}$ program neither existentially quantified variables in the heads of rules nor constraints are allowed.

Semantics of Datalog$^{\exists,\neg s,\bot}$. The semantics of Datalog$^{\exists,\neg s,\bot}$ are defined via the well-known chase procedure. Before defining the chase procedure, we need to recall some auxiliary definitions. A *homomorphism* from a set of atoms X to a set of atoms X' is a partial function $h : \mathbf{U} \cup \mathbf{B} \cup \mathbf{V} \to \mathbf{U} \cup \mathbf{B} \cup \mathbf{V}$ such that (1) $t \in \mathbf{U}$ implies $h(t) \in \mathbf{U}$, and (2) $p(t_1, \ldots, t_n) \in X$ implies $p(h(t_1), \ldots, h(t_n)) \in X'$. A Datalog$^{\exists}$ rule ρ (i.e., a Datalog$^{\exists,\neg}$ rule without negated atoms) is *applicable* to an instance I if there exists a homomorphism h such that $h(\text{body}(\rho)) \subseteq I$. The result of applying ρ to I in this case is an instance $I' = I \cup h'(\text{head}(\rho))$, where h' is a homomorphism such that $h'(?X) = h(?X)$ if $?X \in \text{var}(\text{body}(\rho)) \cap \text{var}(\text{head}(\rho))$, and $h'(?Y)$ is a fresh labeled null not occurring in I if $?Y \in \text{var}(\text{head}(\rho)) \setminus \text{var}(\text{body}(\rho))$. For such an application of ρ to I we write $I \langle \rho, h \rangle I'$; in fact, $I \langle \rho, h \rangle I'$ defines a single chase step.

The chase algorithm takes as input a database D and a Datalog$^{\exists}$ program Π, and performs an exhaustive application of the rules of Π starting from D, which leads to a (possibly infinite) instance denoted $\text{chase}(D, \Pi)$. A *chase sequence* of a database D and a Datalog$^{\exists}$ program Π is a sequence of chase steps $I_i \langle \rho_i, h_i \rangle I_{i+1}$, where $i \geq 0$, $I_0 = D$ and $\rho_i \in \Pi$. The chase of D and Π, denoted $\text{chase}(D, \Pi)$, is defined as follows.

- A *finite chase* of D and Π is a finite chase sequence $I_i \langle \rho_i, h_i \rangle I_{i+1}$, where $i \in [0, m-1]$, and there is no $\rho \in \Pi$ that is applicable to I_m; let $\text{chase}(D, \Pi) = I_m$.
- An infinite chase sequence $I_i \langle \rho_i, h_i \rangle I_{i+1}$, where $i \geq 0$, is *fair* if whenever a rule $\rho \in \Pi$ is applicable to I_i with homomorphism h, then there exists $h' \supseteq h$ and $k > i$ such that $h'(\text{head}(\rho)) \subseteq I_k$. An *infinite chase* of D and Π is a fair infinite chase sequence $I_i \langle \rho_i, h_i \rangle I_{i+1}$, where $i \geq 0$; let $\text{chase}(D, \Pi) = \bigcup_{i=0}^{\infty} I_i$.

We are now ready to define the semantics of Datalog$^{\exists,\neg s,\bot}$. A crucial notion is the indefinite grounding of a Datalog$^{\exists,\neg}$ program Π. A subset of $\mathbf{B}$ is partitioned into infinite sets of nulls $\mathbf{B}_{\rho,?Z}$, one for every $\rho \in \Pi$ and every existentially quantified variable $?Z$ occurring in ρ. An *indefinite instance* of a rule ρ is obtained from ρ by replacing every variable of $\text{var}(\text{body}(\rho))$ by an element of $\mathbf{U} \cup \mathbf{B}$, and every existentially quantified variable $?Z$ by an element of $\mathbf{B}_{\rho,?Z}$. The *indefinite grounding* of Π, denoted $\text{ground}(\Pi)$, is the set of all its indefinite instances. Given an instance I, let Π^I be the program $\{\text{body}^+(\rho) \to \text{head}(\rho) \mid \rho \in \text{ground}(\Pi) \text{ and } (\text{body}^-(\rho) \cap I) = \varnothing\}$. Notice that the rules of Π^I may contain nulls from $\mathbf{B}$. Therefore, we cannot directly use the chase algorithm as defined above with such rules. The reason is because the chase is defined for Datalog$^{\exists}$ rules that can mention only constants of $\mathbf{U}$ and variables of $\mathbf{V}$. Nevertheless, the chase algorithm can be naturally generalized to such rules by simply treating the null values from $\mathbf{B}$ in the same way as the constants from $\mathbf{U}$. Consider now a database D and a Datalog$^{\exists,\neg s,\bot}$ program Π that admits a stratification $\mu : \text{sch}(\Pi) \to [0, \ell]$. Recall that $\text{ex}(\Pi)$ is the program consisting of the Datalog$^{\exists,\neg}$ rules in Π. Therefore, $\text{ex}(\Pi)_i = \{\rho \mid \rho \in \text{ex}(\Pi) \text{ and } \mu(\text{pred}(\text{head}(\rho))) = i\}$. We inductively define the sets $S_0, \ldots, S_\ell$ as follows:

$$S_0 = \text{chase}(D, \text{ex}(\Pi)_0) \qquad \text{and} \qquad S_i = \text{chase}(S_{i-1}, (\text{ex}(\Pi)_i)^{S_{i-1}}).$$

If there is a constraint $\nu \in \Pi$ for which there exists a homomorphism h such that $h(\text{body}(\nu)) \subseteq S_\ell$, then D is *inconsistent* w.r.t. Π; otherwise, D is *consistent* w.r.t. Π. The semantics $\Pi(D)$ of Π over D is defined as $\top$ if D is inconsistent w.r.t. Π; otherwise, $\Pi(D)$ is defined as the (possibly infinite) instance S_ℓ. Note that $\top$ is a special symbol used to indicate that there is an inconsistency.

Consider a Datalog$^{\exists,\neg s,\bot}$ query $Q = (\Pi, p)$, where p is an n-ary predicate, and a database D. The *evaluation* of Q over D is defined as

$$Q(D) = \begin{cases} \top & \text{if } \Pi(D) = \top, \\ \{(t_1, \ldots, t_n) \in \mathbf{U}^n \mid p(t_1, \ldots, t_n) \in \Pi(D)\} & \text{if } \Pi(D) \neq \top. \end{cases}$$

As is customary when studying the complexity of the evaluation problem for a query language, we consider its associated decision problem:

Problem:	Eval
Input:	A database D, a Datalog$^{\exists,\neg s,\bot}$ query Q, and a tuple of constants $\mathbf{t}$.
Question:	Does $Q(D) \neq \top$ imply $\mathbf{t} \in Q(D)$?

Let us clarify that this general formulation refers to the combined complexity of the problem. In this work, we focus our attention on the *data complexity* of this problem, i.e., the complexity of the problem Eval(Q), when the query Q is fixed, and only the database D and the tuple $\mathbf{t}$ form the input. We adopt the convention that when we talk about the data complexity of a problem like Eval (i.e., the class of problems Eval(Q)), we say that it is complete for a complexity class C if each of the problems Eval(Q) is in C, and there exists one problem Eval(Q) that is C-hard.

4 TRIPLE QUERY LANGUAGE

Recall that the main goal of this work is to define a query language with reasoning capabilities to deal with RDFS and OWL vocabularies, navigational capabilities to exploit the graph structure of RDF data, and a general form of recursion much needed to express some natural and useful queries. To this end, we introduce a query language that is based on Datalog$^{\exists,\neg s,\bot}$ and incorporates all the above functionalities. It is well known that Eval for Datalog$^{\exists,\neg s,\bot}$ queries is undecidable. This already holds for Datalog$^{\exists}$ [8, 13], and thus several decidability paradigms have been proposed in the literature. Two of the most expressive decidable languages, which are of special interest for our work, are *weakly-guarded* Datalog$^{\exists}$ [13] and *weakly-frontier-guarded* Datalog$^{\exists}$ [6]. Our query language, dubbed TriQ 1.0, extends weakly-frontier-guarded Datalog$^{\exists}$ with stratified negation and constraints. Before introducing TriQ 1.0, let us recall the key idea of weak(-frontier)-guardedness.

4.1 Weakly(-Frontier)-Guarded Datalog$^{\exists}$

The main principle underlying weakly-guarded Datalog$^{\exists}$ can be informally described as follows: all the harmful body variables, i.e., variables that may be bound by the program to labeled nulls, jointly appear in a body atom. The notion of weak-guardedness is a relaxation of guardedness, which requires all the body variables (harmless or harmful) to jointly appear in a body atom; hence the name weakly-guarded. Weakly-frontier-guarded Datalog$^{\exists}$ extends weakly-guarded Datalog$^{\exists}$ by requiring only the dangerous body variables, i.e., harmful variables that are also propagated to the rule-head, to jointly appear in a body atom. The body variables that are propagated to the rule-head are also known as the *frontier* of the rule, and hence the name weakly-frontier-guarded. Before giving the formal definitions, we first need to recall some auxiliary terminology.

Given a set of predicates X, the set of positions of X, denoted $\text{pos}(X)$, is the set $\{p[i] \mid p \in \text{sch}(X) \text{ and } i \in [1, \text{arity}(p)]\}$. Given a Datalog$^\exists$ program Π, the set of *affected positions* of $\text{sch}(\Pi)$, denoted by $\text{affected}(\Pi)$, is inductively defined as follows:

(1) if there exists $\rho \in \Pi$ such that an existentially quantified variable occurs at position π, then $\pi \in \text{affected}(\Pi)$; and
(2) if there exists $\rho \in \Pi$ and a variable $?V$ that occurs in $\text{body}(\rho)$ only at positions of $\text{affected}(\Pi)$, and $?V$ appears in $\text{head}(\rho)$ at position π, then $\pi \in \text{affected}(\Pi)$.

Let $\text{nonaffected}(\Pi)$ be the set $(\text{pos}(\Pi) \setminus \text{affected}(\Pi))$ of *non-affected positions* of $\text{sch}(\Pi)$.

Example 4.1. Consider the Datalog$^\exists$ program Π:

$$\rho_1 = p(?X, ?Y), s(?Y, ?Z) \rightarrow \exists ?W\, t(?Y, ?X, ?W),$$
$$\rho_2 = t(?X, ?Y, ?Z) \rightarrow \exists ?W\, p(?W, ?Z),$$
$$\rho_3 = t(?X, ?Y, ?Z) \rightarrow s(?X, ?Y).$$

Because of the existentially quantified variables, $t[3]$ and $p[1]$ belong to $\text{affected}(\Pi)$. Since the variable $?X$ occurs in $\text{body}(\rho_1)$ at the affected position $p[1]$, and also at position $t[2]$ in $\text{head}(\rho_1)$, we conclude that $t[2] \in \text{affected}(\Pi)$. Similarly, $p[2]$ and $s[2]$ are affected positions of $\text{sch}(\Pi)$. Notice that, although $?Y$ occurs in the body of the first rule at the affected position $p[2]$, and also at position $t[1]$ in the head of the rule, $t[1]$ is not affected since $?Y$ occurs also at position $s[1] \notin \text{affected}(\Pi)$.

Having the notion of the (non-)affected position of a schema in place, we can classify the body variables of a rule into harmless, harmful and dangerous variables as follows. Let Π be a Datalog$^\exists$ program. Fix a rule $\rho \in \Pi$ and a variable $?V \in \text{var}(\text{body}(\rho))$. Then

— $?V$ is Π-*harmless* if at least one occurrence of it appears in $\text{body}(\rho)$ at a position of $\text{nonaffected}(\Pi)$;
— $?V$ is Π-*harmful* if it is not Π-harmless;
— $?V$ is Π-*dangerous* if it is Π-harmful and appears in $\text{head}(\rho)$.

Let $\text{harmless}(\rho, \Pi)$, $\text{harmful}(\rho, \Pi)$, and $\text{dangerous}(\rho, \Pi)$ be the set of body variables of ρ that are Π-harmless, Π-harmful and Π-dangerous, respectively.

A Datalog$^\exists$ program Π is weakly-frontier-guarded (weakly-guarded, respectively) if, for each $\rho \in \Pi$, there exists an atom $\underline{a} \in \text{body}(\rho)$, called a *guard*, such that $\text{dangerous}(\rho, \Pi) \subseteq \text{var}(\underline{a})$ ($\text{harmful}(\rho, \Pi) \subseteq \text{var}(\underline{a})$, respectively). In other words, the body atom $\underline{a}$ contains (or guards) all the Π-dangerous (Π-harmful, respectively) body variables of ρ. It is not difficult to verify that the program Π in Example 4.1 is weakly-frontier-guarded but not weakly-guarded. A *weakly(-frontier)-guarded Datalog$^\exists$ query* is a Datalog$^\exists$ query (Π, p) such that Π is weakly(-frontier)-guarded.

4.2 The Query Language TriQ 1.0

We proceed to introduce our main language called TriQ 1.0, which extends weakly-frontier-guarded Datalog$^\exists$ with stratified negation and constraints. To introduce negation though, we need to revisit the notion of weak-frontier-guardedness. Given a Datalog$^{\exists,\neg s}$ program Π, we write Π^+ for the program obtained from Π by dropping all the negative atoms. A Datalog$^{\exists,\neg s,\bot}$ program Π is called weakly-frontier-guarded if $\text{ex}(\Pi)^+$ is weakly-frontier-guarded, i.e., we simply need to check whether the program obtained from Π after eliminating the negative atoms and the constraints is weakly-frontier-guarded; weakly-guarded Datalog$^{\exists,\neg s,\bot}$ is defined analogously.

Definition 4.2. A TriQ 1.0 query is a Datalog$^{\exists,\neg s,\bot}$ query that is weakly-frontier-guarded.

A natural question at this point is how expressive TriQ 1.0 is. Interestingly, as we show in the following example, this language can encode some very useful but costly queries; e.g., whether a graph contains a clique of size k.

Example 4.3. Consider an undirected graph $G = (V, E)$, and an integer $k > 0$. Assume that $|V| = n$, where $n > 0$. The graph G and the integer k can be naturally encoded in a database D. More precisely, the database D is defined as

$$\{\text{node}_0(v) \mid v \in V\} \cup \{\text{edge}_0(v, w) \mid (v, w) \in E\} \cup \{\text{succ}_0(0, 1), \ldots, \text{succ}_0(k-1, k)\}.$$

Our goal is to construct a TriQ 1.0 query $Q = (\Pi, \text{yes})$, where yes() is a 0-ary predicate, such that G contains a k-clique iff $Q(D) \neq \varnothing$. The program Π is defined as the union of the two subprograms Π_{aux} and Π_{clique}. Π_{aux} is used to compute some auxiliary relations that are needed when checking whether G contains a k-clique, while Π_{clique} checks for the existence of a k-clique.

The Program Π_{aux}

Π_{aux} contains two rules to define the usual linear order on the domain of succ_0:

$$\text{succ}_0(?X, ?Y) \rightarrow \text{less}_0(?X, ?Y),$$
$$\text{succ}_0(?X, ?Y), \text{less}_0(?Y, ?Z) \rightarrow \text{less}_0(?X, ?Z).$$

It also contains rules that define the minimum and maximum elements of this linear order:

$$\text{less}_0(?X, ?Y) \rightarrow \text{not\_max}(?X),$$
$$\text{less}_0(?X, ?Y) \rightarrow \text{not\_min}(?Y),$$
$$\text{less}_0(?X, ?Y), \neg\text{not\_min}(?X) \rightarrow \text{zero}_0(?X),$$
$$\text{less}_0(?Y, ?X), \neg\text{not\_max}(?X) \rightarrow \text{max}_0(?X).$$

Finally, Π_{aux} contains the following rules that they simply copy the atoms of D, and the atoms generated by Π_{aux}, into a new schema that will be used by Π_{clique}:

$$\text{node}_0(?X) \rightarrow \text{node}(?X),$$
$$\text{edge}_0(?X, ?Y) \rightarrow \text{edge}(?X, ?Y),$$
$$\text{succ}_0(?X, ?Y) \rightarrow \text{succ}(?X, ?Y),$$
$$\text{less}_0(?X, ?Y) \rightarrow \text{less}(?X, ?Y),$$
$$\text{zero}_0(?X) \rightarrow \text{zero}(?X),$$
$$\text{max}_0(?X) \rightarrow \text{max}(?X).$$

The Program Π_{clique}

Let us first give the key idea underlying Π_{clique}. Intuitively, Π_{clique} constructs a tree of mappings (rooted at some dummy mapping), where a mapping at level $i \in [1, k]$ actually maps the set of integers $[1, i]$ to the vertices of G. Each mapping μ at level $i < k$ has n child-mappings, one for each node of G. The child-mapping μ' of μ (for a node v) simply extends μ by mapping $(i + 1)$ to v. The k-th level of the tree contains all the possible n^k mappings $\mu : [1, k] \rightarrow V$. It is then easy to check whether there exists a mapping that maps $[1, k]$ to a clique of G.

Now we define Π_{clique}. In this program, apart from the predicates $\text{node}(\cdot)$, $\text{edge}(\cdot, \cdot)$, $\text{succ}(\cdot, \cdot)$, $\text{less}(\cdot, \cdot)$, $\text{zero}(\cdot)$, and $\text{max}(\cdot, \cdot)$, generated by Π_{aux}, we also have the following:

(1) $\text{ism}(\mu, i)$: μ is a mapping at level i of the tree;
(2) $\text{map}(\mu, i, v)$: $\mu(i) = v$;

(3) next(μ, v, μ'): μ' is obtained from μ by mapping $(i+1)$ to v (assuming that μ is a mapping at level i); and
(4) noclique(μ): μ does not map to a clique.

The program Π_{clique} consists of the following rules:

$$\begin{aligned}
&\text{zero}(?X) \rightarrow \exists ?Y\, \text{ism}(?Y, ?X),\\
&\text{ism}(?X, ?Y), \text{succ}(?Y, ?Z), \text{node}(?W) \rightarrow\\
&\qquad\qquad \exists ?U\, \text{next}(?X, ?W, ?U), \text{ism}(?U, ?Z), \text{map}(?U, ?Z, ?W),\\
&\text{next}(?X, ?Y, ?Z), \text{map}(?X, ?U, ?V) \rightarrow \text{map}(?Z, ?U, ?V),\\
&\text{less}(?X, ?Y), \text{map}(?Z, ?X, ?W), \text{map}(?Z, ?Y, ?U), \neg\text{edge}(?W, ?U) \rightarrow \text{noclique}(?Z),\\
&\text{less}(?X, ?Y), \text{map}(?Z, ?X, ?W), \text{map}(?Z, ?Y, ?W) \rightarrow \text{noclique}(?Z),\\
&\text{ism}(?X, ?Y), \text{max}(?Y), \neg\text{noclique}(?X) \rightarrow \text{yes}().
\end{aligned}$$

Notice that the purpose of the fifth rule is to avoid the use of the same node more than once in a clique (which can happen if G contains self-loops).

4.3 The Complexity of TriQ 1.0

The above example shows that the query evaluation problem for TriQ 1.0 is intractable in data complexity. In fact, we show that:

Theorem 4.4. Eval *for* TriQ 1.0 *is* ExpTime*-complete in data complexity.*

Proof. Eval for weakly-guarded Datalog$^{\exists}$ is ExpTime-hard in data complexity [13], which immediately implies the desired lower bound. Let us now proceed with the upper bound. Consider a database D and a (fixed) TriQ 1.0 query $Q = (\Pi, p)$. We construct in constant time the query $Q' = (\text{ex}(\Pi) \cup \Pi_\perp, p)$, where

$$\Pi_\perp = \{\underline{a}_1, \ldots, \underline{a}_n \rightarrow p(\star, \ldots, \star) \mid \underline{a}_1, \ldots, \underline{a}_n \rightarrow \perp \in \Pi\},$$

with $\star$ being a special constant not in D or Π. It is clear that $Q(D) \neq \top$ iff $(\star, \ldots, \star) \notin Q'(D)$. Moreover, if $Q(D) \neq \top$, then $\mathbf{t} \in Q(D)$ iff $\mathbf{t} \in Q'(D)$, for every $\mathbf{t} \in \mathbf{U}^{\text{arity}(p)}$. Therefore, for an arbitrary tuple $\mathbf{t} \in \mathbf{U}^{\text{arity}(p)}$,

$$Q(D) \neq \top \text{ implies } \mathbf{t} \in Q(D) \quad \text{iff} \quad (\star, \ldots, \star) \notin Q'(D) \text{ implies } \mathbf{t} \in Q'(D).$$

By construction, Q' is a weakly-frontier-guarded Datalog$^{\exists,\neg s}$ query. Thus, to establish the desired upper bound, it suffices to show that query evaluation for weakly-frontier-guarded Datalog$^{\exists,\neg s}$ is in ExpTime in data complexity. The latter can be reduced to Eval for weakly-guarded Datalog$^{\exists,\neg s}$ via a database-independent reduction; implicit in [24]. Therefore, it suffices to show that query evaluation for weakly-guarded Datalog$^{\exists,\neg s}$ is in ExpTime in data complexity. This can be shown by exploiting a recent complexity result for guarded Datalog$^{\exists,\neg s}$ [26].

A *guarded Datalog*$^{\exists,\neg s}$ *query* is a Datalog$^{\exists,\neg s}$ query (Π, p) such that Π is guarded, i.e., for each rule $\rho \in \Pi$, there exists an atom $\underline{a} \in \text{body}^+(\rho)$ such that $\text{var}(\text{body}(\rho)) \subseteq \text{var}(\underline{a})$. It is implicit in [26] that Eval for guarded Datalog$^{\exists,\neg s}$ is feasible in double-exponential time in the arity of the underlying schema, in exponential time in the size of the given query program, and in polynomial time in the size of the given database.[7] Having this result in place, to establish the desired upper bound it

[7] In fact, the work [26] considers guarded Datalog$^{\exists,\neg}$, where the (non-stratified) negation is interpreted according to the well-founded semantics, which generalizes guarded Datalog$^{\exists,\neg s}$.

suffices to reduce Eval for weakly-guarded Datalog$^{\exists,\neg s}$ to Eval for guarded Datalog$^{\exists,\neg s}$ in polynomial time, without increasing the arity of the underlying schema. This can be done by instantiating the harmless variables occurring in a rule of the given query with constants occurring in the given database. More precisely, given a database D and a weakly-guarded Datalog$^{\exists,\neg s}$ query $Q = (\Pi, p)$, we construct the guarded Datalog$^{\exists,\neg s}$ query $Q' = (\Pi', p)$, where $\Pi' = \bigcup_{\rho \in \Pi} \text{inst}(\rho)$ and $\text{inst}(\rho)$ is the set of rules obtained after replacing the $\text{ex}(\Pi)^+$-harmless variables occurring in ρ with constants of $\text{dom}(D)$ in all the possible ways. It is clear that $Q(D) = Q'(D)$, while $|\Pi'|$ is polynomial in the size of $\text{dom}(D)$. We conclude that Eval for weakly-guarded Datalog$^{\exists,\neg s}$ is in ExpTime in data complexity, and the claim follows. □

4.4 The Expressive Power of TriQ 1.0

An important issue for a query language is to understand its expressive power, a topic common to database theory. Roughly, by the expressive power of a query language we refer to the set of all queries expressible in that language. In formal terms, a query Q defines a function f_Q that maps each input database D (over a certain schema) to a set of answers $f_Q(D) \subseteq \text{dom}(D)^n$, where $n \geq 0$ is the arity of Q. The *expressive power* of a query language $\mathbb{L}$ is the set of functions f_Q for all queries Q expressible in $\mathbb{L}$ by some query expression (or program); this syntactic expression is usually identified with the semantic query that it defines, and, by abuse of terminology, simply called query.

In this context, a crucial task is to determine the *absolute expressive power* of a query language $\mathbb{L}$. This is done by showing that $\mathbb{L}$ is able to express exactly the queries whose evaluation is in a complexity class C, and we write $\mathbb{L} = C$. The evaluation of an n-ary query Q is the problem of deciding, given a database D and a tuple $\mathbf{t} \in \text{dom}(D)^n$, whether $\mathbf{t} \in f_Q(D)$. It holds that

Theorem 4.5. TriQ 1.0 = *ExpTime*.

Proof. We need to show that (i) the evaluation complexity of a TriQ 1.0 query is in ExpTime, and (ii) for every query Q whose evaluation is in C, there exists a TriQ 1.0 query Q' such that $f_Q(D) = f_{Q'}(D)$, for every database D. The former follows from the fact that Eval for TriQ 1.0 is in ExpTime in data complexity (Theorem 4.4), while the latter follows from [24], where the same result is shown for weakly-guarded Datalog$^{\exists,\neg s}$. □

At this point, let us clarify that there is a crucial difference between the fact that Eval for a query language $\mathbb{L}$ is C-hard in data complexity, and the fact that $\mathbb{L} = C$. The former simply says that *there exists* a query Q expressible in $\mathbb{L}$ for which the evaluation problem is C-hard. The latter says that Q expresses *all* queries whose evaluation is in C (including all the C-hard queries). Clearly, the above result implies that TriQ 1.0 and weakly-guarded Datalog$^{\exists,\neg s}$ are equally expressive query languages. However, the fact that TriQ 1.0 is based on the more refined notion of weak-frontier-guardedness, allows us to write more intuitive and succinct queries than weakly-guarded Datalog$^{\exists,\neg s}$.

5 FROM SPARQL OVER OWL 2 QL TO TriQ 1.0

The first version of the Web ontology language OWL was released in 2004 [30]. The second version of this language, which is called OWL 2, was released in 2012 [41]. OWL 2 includes three profiles that can be implemented more efficiently [31]. One of these profiles, called OWL 2 QL, is based on the description logic DL-Lite$_\mathcal{R}$ [16] and designed to be used in applications where query answering is the most important reasoning task. As the main goal of our article is to design a query language that naturally embeds the fundamental features for querying RDF, we focus on OWL 2 QL, identify a core fragment of it, called OWL 2 QL core, which corresponds to DL-Lite$_\mathcal{R}$, and show that every

SPARQL query under the OWL 2 QL core direct semantics entailment regime, which is inherited from the OWL 2 direct semantics entailment regime [22, 28], can be naturally translated into a TriQ 1.0 query.[8] Furthermore, a second goal of this section is to show that the use of TriQ 1.0 allows us to formulate SPARQL queries in a simpler way, as a more natural notion of entailment can be easily encoded by using this query language.

For the sake of presentation, we first omit the direct semantics entailment regime, and explain in Section 5.1 how a SPARQL query can be translated into a Datalog$^{\neg s}$ query. It is important to clarify that it is known that SPARQL can be translated into Datalog$^{\neg s}$ [2, 3, 5, 18, 36, 40], if one focuses on RDF graphs with RDFS vocabulary extended with a special symbol to represent the null value (and with a built-in predicate to check for this symbol). Thus, the goal of Section 5.1 is not to prove that SPARQL can be embedded into Datalog$^{\neg s}$, but instead to propose a translation that uses such a special symbol for the null value in a fairly limited way (in fact, we only use this symbol to compute that final answer to the query), and which can be easily extended to deal not only with the RDFS vocabulary but also with the vocabulary used in OWL 2 QL core ontologies. In fact, we extend this translation in Section 5.2 and show that every SPARQL query under the OWL 2 QL core direct semantics entailment regime can be transformed into a TriQ 1.0 query. Moreover, we show in Section 5.3 that a more natural notion of entailment, which is obtained by removing a restriction from the regime proposed in [22], can also be encoded in TriQ 1.0.

5.1 Translating SPARQL into Datalog$^{\neg s}$

In this section, we explain via some illustrative examples how a SPARQL query can be translated into a Datalog$^{\neg s}$ query. As it is already known that SPARQL can be translated into Datalog$^{\neg s}$, we do not provide the details of the translation, but rather mention what is needed to fix the notation used in the rest of the article. The complete translation can be found in Appendix A.

From now on, given an RDF graph G, we define

$$\tau_{\mathrm{db}}(G) = \{\mathrm{triple}(a, b, c) \mid (a, b, c) \in G\},$$

i.e., the instance of the relational schema $\{\mathrm{triple}(\cdot, \cdot, \cdot)\}$ naturally associated with G.

Example 5.1. We give a series of graph patterns, where their structural complexity is progressively increased, and explain how they are encoded in Datalog$^{\neg s}$.

—We first consider the graph pattern

$$P_1 = (?X, \mathrm{name}, ?Y),$$

where name is a constant, that asks for the list of pairs (a, b) of elements from an RDF graph G such that b is the name of a in G. This graph pattern can be easily represented as a Datalog program over $\tau_{\mathrm{db}}(G)$:

$$\mathrm{triple}(?X, \mathrm{name}, ?Y) \rightarrow \mathrm{query}_{P_1}(?X, ?Y).$$

The predicate $\mathrm{query}_{P_1}(\cdot, \cdot)$ is used to store the answer to the graph pattern P_1.

—Now consider the graph pattern

$$P_2 = (?X, \mathrm{name}, \_ : B),$$

where $\_ : B$ is a blank node. This time we are asking for the list of elements in an RDF graph G that have a name (the blank node $\_ : B$ is used in P_2 to indicate that $?X$ has a name, but

[8] Let us clarify that we focus on OWL 2 QL core, instead of the full formalism of OWL 2 QL, for technical clarity. However, our approach is generic enough to deal with all the constructs of OWL 2 QL.

that we are not interested in retrieving it). As in the previous case, this graph pattern can be easily represented as a Datalog program over $\tau_{\text{db}}(G)$:

$$\text{triple}(?X, \text{name}, ?Y) \rightarrow \text{query}_{P_2}(?X). \tag{7}$$

Given that blank nodes are used as existential variables in basic graph patterns, $?Y$ is used in the previous rule to represent blank node $\_:B$. However, this time we do not include the variable $?Y$ in the head of the rule as we are not interested in retrieving names.

—As a third example, consider the graph pattern

$$P_3 = \underbrace{(?X, \text{name}, ?Y)}_{P_3^1} \text{ OPT } \underbrace{(?X, \text{phone}, ?Z)}_{P_3^2},$$

where phone is a constant. For every constant a in an RDF graph G, this graph pattern is asking for the name and phone number of a, if the information about the phone number of a is available in G, and otherwise it is only asking for the name of a. The basic graph patterns P_3^1 and P_3^2 are represented via the rules

$$\text{triple}(?X, \text{name}, ?Y) \rightarrow \text{query}_{P_3^1}(?X, ?Y), \tag{8}$$

$$\text{triple}(?X, \text{phone}, ?Z) \rightarrow \text{query}_{P_3^2}(?X, ?Z). \tag{9}$$

Predicates $\text{query}_{P_3^1}(\cdot,\cdot)$ and $\text{query}_{P_3^2}(\cdot,\cdot)$ are used in the representation of graph pattern P_3 in Datalog$^{\neg s}$. More precisely, we first construct a set of rules for the cases where the information about phone numbers is available:

$$\text{query}_{P_3^1}(?X, ?Y), \text{query}_{P_3^2}(?X, ?Z) \rightarrow \text{query}_{P_3}(?X, ?Y, ?Z), \tag{10}$$

$$\text{query}_{P_3^1}(?X, ?Y), \text{query}_{P_3^2}(?X, ?Z) \rightarrow \text{compatible}_{P_3}(?X). \tag{11}$$

As for the previous graph patterns, we use a predicate $\text{query}_{P_3}(\cdot,\cdot,\cdot)$ to store the answers to the query. But in this case, we also include a predicate $\text{compatible}_{P_3}(\cdot)$, which stores the individuals with phone numbers. This predicate is used in the definition of the third rule utilized to represent P_3, which takes care of the individuals without phone numbers:

$$\text{query}_{P_3^1}(?X, ?Y), \neg\text{compatible}_{P_3}(?X) \rightarrow \text{query}^{\{3\}}_{P_3}(?X, ?Y). \tag{12}$$

The predicate $\text{query}^{\{3\}}_{P_3}(\cdot,\cdot)$ is used to store the answer, which has a supra-index $\{3\}$ to indicate that the third argument in the answer to P_3 is missing (which is the phone number).

—As a final example, consider the graph pattern

$$P_4 = \underbrace{((?X, \text{name}, ?Y) \text{ OPT } (?X, \text{phone}, ?Z))}_{P_4^1} \text{ AND } \underbrace{(?Z, \text{phone\_company}, ?W)}_{P_4^2},$$

where phone_company is a constant used to indicate that a phone number is associated with a phone company. In this case, we first consider a set of Datalog$^{\neg s}$ rules that define the answer to the sub-pattern P_4^1, which is stored in predicates $\text{query}_{P_4^1}(\cdot,\cdot,\cdot)$ and $\text{query}^{\{3\}}_{P_4^1}(\cdot,\cdot)$, and to the sub-pattern P_4^2, which is stored in predicate $\text{query}_{P_4^2}(\cdot,\cdot)$. We have already seen what these rules look like, and thus we skip their definition. Having the above predicates in place, we now use two rules to define the answer to P_4. The first rule considers the case of the individuals with phone numbers:

$$\text{query}_{P_4^1}(?X, ?Y, ?Z), \text{query}_{P_4^2}(?Z, ?W) \rightarrow \text{query}_{P_4}(?X, ?Y, ?Z, ?W).$$

Moreover, the second rule used to define the answers to P_4 considers the case of the individuals without phone numbers, where a join is not needed:

$$\text{query}^{\{3\}}_{P_4^1}(?X, ?Y), \text{query}_{P_4^2}(?Z, ?W) \rightarrow \text{query}_{P_4}(?X, ?Y, ?Z, ?W). \tag{13}$$

Although query P_4 is a valid SPARQL query, it can be difficult to interpret because if a person has no phone number, then she gets all the phone companies associated to her. The rules used to translate P_4 make this phenomenon very clear: the two predicates in the body of rule (13) do not have any variables in common, so every pair of values assigned to variables $?X, ?Y$ is combined with every pair of values assigned to variables $?Z, ?W$.

This completes our example.

The approach shown in Example 5.1 can be generalized to represent any graph pattern P. Our goal is to construct a Datalog$^{\neg s}$ query $P_{\text{dat}} = (\Pi, \text{answer}_P)$, where Π is the union of three subprograms:

(1) $\tau_{\text{bgp}}(P)$ encodes the basic graph patterns occurring in P.
(2) $\tau_{\text{opr}}(P)$ represents the non-basic graph patterns occurring in P; in fact, these rules are used to encode the semantics of the SPARQL operators appearing in P.
(3) $\tau_{\text{out}}(P)$ computes the output predicate answer_P.

Example 5.1 gives a good idea of how the programs $\tau_{\text{bgp}}(P)$ and $\tau_{\text{opr}}(P)$ are defined (their precise definitions can be found in Appendix A). For the definition of $\tau_{\text{out}}(P)$, there is one issue that needs to be resolved. Assume that P_3 is the graph pattern in Example 5.1. In this case, we expect $\text{query}_{P_3}(\cdot, \cdot, \cdot)$ to be the output predicate. However, the predicate $\text{query}^{\{3\}}_{P_3}(\cdot, \cdot)$ is also used to collect some answers; more specifically, $\text{query}^{\{3\}}_{P_3}(?X, ?Y)$ is used to collect the answers to the query where $?Z$ is not assigned a value. To deal with this issue, the following rules are included in $\tau_{\text{opr}}(P_3)$:

$$\text{query}_{P_3}(?X, ?Y, ?Z) \rightarrow \text{answer}_{P_3}(?X, ?Y, ?Z),$$
$$\text{query}^{\{3\}}_{P_3}(?X, ?Y) \rightarrow \text{answer}_{P_3}(?X, ?Y, \star),$$

where $\star$ is a special constant used to represent the fact that some positions in a tuple have not been assigned values. Thus, $\text{answer}_{P_3}(\cdot, \cdot, \cdot)$ is the only output predicate in this example (the precise definition of $\tau_{\text{out}}(P)$ can be found in Appendix A).

Having the above three programs in place, we are now ready to define the Datalog$^{\neg s}$ query that represents the graph pattern P. In particular, we define

$$P_{\text{dat}} = (\tau_{\text{bgp}}(P) \cup \tau_{\text{opr}}(P) \cup \tau_{\text{out}}(P), \text{answer}_P).$$

Notice that P_{dat} is a non-recursive Datalog$^{\neg s}$ query of exponential size. Is it possible to represent a graph pattern P as a non-recursive Datalog$^{\neg s}$ query of polynomial size? This is an interesting question that goes beyond the scope of this work.

In order to state the correctness of our translation, we need to define one last notion. Let P be a graph pattern, G an RDF graph, and $\mathbf{t} = (t_1, \ldots, t_n)$ a tuple constants that belongs to $P_{\text{dat}}(\tau_{\text{db}}(G))$. By construction, in the set of rules $\tau_{\text{out}}(P)$ there is an atom $\text{answer}_P(?X_1, \ldots, ?X_n)$ that contains only variables (and not the constant $\star$). We define a mapping $\mu_{\mathbf{t},P}$ corresponding to $\mathbf{t}$ given P by taking $\text{dom}(\mu_{\mathbf{t},P}) = \{?X_i \mid i \in [1, n] \text{ and } t_i \neq \star\}$ and, for every $i \in [1, n]$, $t_i \neq \star$ implies $\mu_{\mathbf{t},P}(?X_i) = t_i$. We then define the set of mappings corresponding to the answers of P_{dat} given $\tau_{\text{db}}(G)$:

$$[\![P_{\text{dat}}, \tau_{\text{db}}(G)]\!] = \{\mu_{\mathbf{t},P} \mid \mathbf{t} \in P_{\text{dat}}(\tau_{\text{db}}(G))\}.$$

With this notation in place, we are ready to state that our translation is correct, which can be easily shown by induction on the structure of P.

Theorem 5.2. *For every graph pattern P and RDF graph G, it holds that* $[\![P]\!]_G = [\![(P_{\text{dat}}, \tau_{\text{db}}(G))]\!]$.

5.2 SPARQL Entailment Regime and TriQ 1.0

As pointed out in Section 1, several functionalities were added to SPARQL 1.1 [25] to overcome some of the limitations of the first version of this language. In particular, SPARQL 1.1 includes an entailment regime to deal with RDFS and OWL vocabularies [22, 28]. In this section, we show how this functionality can be encoded by using TriQ 1.0 if we focus on a specific ontology language.

Storing Ontologies in RDF. We start by defining a fragment of OWL 2 QL that includes the main features of the description logic DL-Lite$_{\mathcal{R}}$ [16], on which the profile OWL 2 QL is based. The vocabulary Σ of an OWL 2 QL core ontology is a finite set of unary and binary predicates, called classes and properties, respectively. A basic property over Σ is either p or p^-, where p is a property in Σ, while a basic class over Σ is either a or $\exists r$, where a is a class in Σ and r is a basic property over Σ. To represent an OWL 2 QL core ontology over a vocabulary Σ, we first include the following triples to indicate what the classes and properties in Σ are:

— For every class a in Σ, we include the triple

$$(a, \text{rdf:type}, \text{owl:Class}).$$

Notice that this triple uses the URIs rdf:type and owl:Class, and indicates that a, which is also a URI, is of type class.

— For every property p in Σ, we include the following triples, where p, p^-, $\exists p$, and $\exists p^-$ are considered as URIs (constants), and they are assumed to be pairwise distinct:

$$(p, \text{rdf:type}, \text{owl:ObjectProperty}) \qquad (p^-, \text{rdf:type}, \text{owl:ObjectProperty})$$

indicating that p and p^- are properties,

$$(p, \text{owl:inverseOf}, p^-) \qquad (p^-, \text{owl:inverseOf}, p)$$

indicating that p^- is the inverse of p and vice versa,

$$\begin{aligned} (\exists p, \text{rdf:type}, \text{owl:Restriction}) \qquad & (\exists p^-, \text{rdf:type}, \text{owl:Restriction}), \\ (\exists p, \text{owl:onProperty}, p) \qquad & (\exists p^-, \text{owl:onProperty}, p^-), \\ (\exists p, \text{owl:someValueFrom}, \text{owl:Thing}) \qquad & (\exists p^-, \text{owl:someValueFrom}, \text{owl:Thing}) \end{aligned}$$

indicating that $\exists p$ and $\exists p^-$ are restrictions of p and p^-, respectively, and finally

$$(\exists p, \text{rdf:type}, \text{owl:Class}) \qquad (\exists p^-, \text{rdf:type}, \text{owl:Class})$$

indicating that $\exists p$ and $\exists p^-$ are classes.

We now indicate how OWL 2 QL core ontologies are stored as RDF graphs, following the standard syntax to represent OWL 2 ontologies as RDF triples [33]. By using the functional-style syntax of OWL [32], we can have the following axioms in an OWL 2 QL core ontology:

— SubClassOf(b_1, b_2): a basic class b_1 is a sub-class of a basic class b_2.
— SubObjectProperty(r_1, r_2): r_1 is a subproperty of r_2, where r_1, r_2 are basic properties.
— DisjointClasses(b_1, b_2): basic classes b_1 and b_2 are disjoint.
— DisjointObjectProperties(r_1, r_2): basic properties r_1 and r_2 are disjoint.
— ClassAssertion(b, a): a constant a belongs to a basic class b.
— ObjectPropertyAssertion(p, a_1, a_2): a constant a_1 is related to a constant a_2 via a property p.

Table 1. Representation of OWL 2 QL Core Axioms as RDF Triples

OWL 2 QL core Axiom	**RDF Triple**
SubClassOf(b_1, b_2)	(b_1, rdfs:subClassOf, b_2)
SubObjectPropertyOf(r_1, r_2)	(r_1, rdfs:subPropertyOf, r_2)
DisjointClasses(b_1, b_2)	(b_1, owl:disjointWith, b_2)
DisjointObjectProperties(r_1, r_2)	(r_1, owl:propertyDisjointWith, r_2)
ClassAssertion(b, a)	(a, rdf:type, b)
ObjectPropertyAssertion(p, a_1, a_2)	(a_1, p, a_2)

Moreover, by following the mapping defined in [33], we have that the above axioms are stored as RDF triples as shown in Table 1. We say that an RDF graph G represents an OWL 2 QL core ontology if there is an OWL 2 QL core ontology $\mathcal{O}$ such that its representation as RDF generates G.

OWL 2 QL Core Direct Semantics Entailment Regime. We proceed to show how a graph pattern is evaluated under the OWL 2 QL core direct semantics entailment regime, which is based on the definition of a direct semantics entailment regime for SPARQL 1.1 given in [22]. To compute the answer to a graph pattern, this regime is first applied at the level of basic graph patterns, and then the results of this step are combined using the standard semantics for the SPARQL operators [28]. Thus, we only need to define the OWL 2 QL core direct semantics entailment regime for basic graph patterns. Consider a basic graph pattern P. Under the OWL 2 QL core direct semantics entailment regime, the evaluation of P over an RDF graph G adopts an active domain semantics, that is, it uses the notion of entailment in OWL 2 QL core (which corresponds to the notion of entailment in DL-Lite$_{\mathcal{R}}$) but allowing the variables and blank nodes in P to take only values from G. For example, assume that we are given an RDF graph G consisting of

$$(\text{dog}, \text{rdf:type}, \text{animal}) \qquad (\text{animal}, \text{rdfs:subClassOf}, \exists \text{eats}), \tag{14}$$

which indicate that dog is an animal, and every animal eats something. Moreover, assume that we want to retrieve the list of elements of G that eat something. The natural way to formulate this query is by using a graph pattern of the form $(?X, \text{eats}, \_ : B)$, where $\_ : B$ is a blank node. However, the answer to this query is empty under the OWL 2 direct semantics entailment regime, as there are no elements a, b in G that can be assigned to $?X$ and $\_ : B$ in such a way that the triple (a, eats, b) is implied by the axioms in G. In other words, the answer to $(?X, \text{eats}, \_ : B)$ is empty under the active domain semantics adopted in SPARQL 1.1. To obtain a correct answer in this case, we can consider the graph pattern $(?X, \text{rdf:type}, \exists \text{eats})$, as the triples in G can be used to infer the triple $(\text{dog}, \text{rdf:type}, \exists \text{eats})$, from which the correct answer dog is obtained.

Let G be an RDF graph representing an OWL 2 QL core ontology. Given $\mathbf{t} \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$, we write $G \models \mathbf{t}$ to indicate that $\mathbf{t}$ is implied by G as defined in [22, 31], which in turn is based on the notion of entailment for DL-Lite$_{\mathcal{R}}$ [16]. Moreover, given a basic graph pattern P, the evaluation of P over G under the OWL 2 QL core direct semantics entailment regime, denoted by $[\![P]\!]^{\mathbf{U}}_G$, is defined as

$$\{\mu \mid \text{dom}(\mu) = \text{var}(P) \text{ and there exists } h : \mathbf{B} \to \mathbf{U} \text{ such that for every } \mathbf{t} \in \mu(h(P)) \colon G \models \mathbf{t}\}. \tag{15}$$

Notice that $\mathbf{U}$ in $[\![P]\!]^{\mathbf{U}}_G$ indicates that every variable and blank node in P has to be assigned a constant, as $\mathbf{U}$ is the range of functions h and μ in the previous definition. Moreover, the evaluation of a graph pattern P over an RDF graph G under the OWL 2 QL core direct semantics entailment regime, denoted by $[\![P]\!]^{\mathbf{U}}_G$, is recursively defined as the usual semantics for graph patterns (which is given in Section 3) but replacing the rule for evaluating basic graph patterns by rule (15).

In what follows, we define a *fixed* Datalog$^{\exists,\neg s,\bot}$ program $\tau_{\text{owl2ql\_core}}$ that is used to encode the semantics $[\![\cdot]\!]^{\mathsf{U}}_{G}$. In this program, we first include a Datalog rule to store in a unary predicate C all the URIs from the graph (recall that we assume that an RDF graph does not contain blank nodes):

$$\text{triple}(?X, ?Y, ?Z) \rightarrow \text{C}(?X), \text{C}(?Y), \text{C}(?Z). \tag{16}$$

Then we define some Datalog rules that store the different elements in the ontology:

$$\begin{aligned}
\text{triple}(?X, \text{rdf:type}, ?Y) &\rightarrow \text{type}(?X, ?Y),\\
\text{triple}(?X, \text{rdfs:subPropertyOf}, ?Y) &\rightarrow \text{sp}(?X, ?Y),\\
\text{triple}(?X, \text{owl:inverseOf}, ?Y) &\rightarrow \text{inv}(?X, ?Y),\\
\text{triple}(?X, \text{rdf:type}, \text{owl:Restriction}),&\\
\text{triple}(?X, \text{owl:onProperty}, ?Y),&\\
\text{triple}(?X, \text{owl:someValueFrom}, \text{owl:Thing}) &\rightarrow \text{restriction}(?X, ?Y),\\
\text{triple}(?X, \text{rdfs:subClassOf}, ?Y) &\rightarrow \text{sc}(?X, ?Y),\\
\text{triple}(?X, \text{owl:disjointWith}, ?Y) &\rightarrow \text{disj}(?X, ?Y),\\
\text{triple}(?X, \text{owl:propertyDisjointWith}, ?Y) &\rightarrow \text{disj_property}(?X, ?Y),\\
\text{triple}(?X, ?Y, ?Z) &\rightarrow \text{triple}_1(?X, ?Y, ?Z).
\end{aligned}$$

If we have the triples $(a, \text{rdf:type}, b)$ and $(b, \text{rdfs:subClassOf}, \exists r)$ in an OWL 2 QL core ontology, then the Datalog$^{\exists,\neg s,\bot}$ program $\tau_{\text{owl2ql\_core}}$ will create a triple of the form (a, r, z), where z is a null value. If (a, r, z) is stored in the relation triple, then by using rule (16) we will conclude that $\text{C}(z)$ holds, violating the intended interpretation of predicate C. To solve this problem, we include the Datalog rule $\text{triple}(?X, ?Y, ?Z) \rightarrow \text{triple}_1(?X, ?Y, ?Z)$ to produce a copy of the predicate $\text{triple}(\cdot, \cdot, \cdot)$ in the predicate $\text{triple}_1(\cdot, \cdot, \cdot)$. In this way, the new values are added to $\text{triple}_1(\cdot, \cdot, \cdot)$, that is, we do not modify the predicate $\text{triple}(\cdot, \cdot, \cdot)$ but instead both $\text{triple}_1(a, \text{rdf:type}, b)$ and $\text{triple}_1(b, \text{rdfs:subClassOf}, \exists r)$ hold, from which we conclude that $\text{triple}_1(a, r, z)$ also holds. Moreover, we include

$$\begin{aligned}
\text{sp}(?X_1, ?X_2), \text{inv}(?Y_1, ?X_1), \text{inv}(?Y_2, ?X_2) &\rightarrow \text{sp}(?Y_1, ?Y_2),\\
\text{type}(?X, \text{owl:ObjectProperty}) &\rightarrow \text{sp}(?X, ?X),\\
\text{sp}(?X, ?Y), \text{sp}(?Y, ?Z) &\rightarrow \text{sp}(?X, ?Z)
\end{aligned}$$

to reason about properties. The first rule states that if p is a sub-property of q, then p^- is a sub-property of q^-. The other rules state that sub-property is reflexive and transitive. We also include

$$\begin{aligned}
\text{sp}(?X_1, ?X_2), \text{restriction}(?Y_1, ?X_1), \text{restriction}(?Y_2, ?X_2) &\rightarrow \text{sc}(?Y_1, ?Y_2),\\
\text{type}(?X, \text{owl:Class}) &\rightarrow \text{sc}(?X, ?X),\\
\text{sc}(?X, ?Y), \text{sc}(?Y, ?Z) &\rightarrow \text{sc}(?X, ?Z).
\end{aligned}$$

The first rule states that if p is a sub-property of q, then $\exists p$ is a sub-class of $\exists q$. The other rules state that sub-class is reflexive and transitive. We include the following to reason about disjointness:

$$\begin{aligned}
\text{disj}(?X_1, ?X_2), \text{sc}(?Y_1, ?X_1), \text{sc}(?Y_2, ?X_2) &\rightarrow \text{disj}(?Y_1, ?Y_2),\\
\text{disj_property}(?X_1, ?X_2), \text{sp}(?Y_1, ?X_1), \text{sp}(?Y_2, ?X_2) &\rightarrow \text{disj_property}(?Y_1, ?Y_2).
\end{aligned}$$

Finally, we include the following rules to reason about membership assertions:

$$\begin{aligned}
\text{triple}_1(?X, ?U, ?Y), \text{sp}(?U, ?V) &\rightarrow \text{triple}_1(?X, ?V, ?Y),\\
\text{triple}_1(?X, ?U, ?Y), \text{inv}(?U, ?V) &\rightarrow \text{triple}_1(?Y, ?V, ?X),\\
\text{type}(?X, ?Y), \text{restriction}(?Y, ?U) &\rightarrow \exists ?Z\, \text{triple}_1(?X, ?U, ?Z),\\
\text{type}(?X, ?Y) &\rightarrow \text{triple}_1(?X, \text{rdf:type}, ?Y),\\
\text{type}(?X, ?Y), \text{sc}(?Y, ?Z) &\rightarrow \text{type}(?X, ?Z),\\
\text{triple}_1(?X, ?U, ?Y), \text{restriction}(?Z, ?U) &\rightarrow \text{type}(?X, ?Z),\\
\text{type}(?X, ?Y), \text{type}(?X, ?Z), \text{disj}(?Y, ?Z) &\rightarrow \bot,\\
\text{triple}_1(?X, ?U, ?Y), \text{triple}_1(?X, ?V, ?Y),&\\
\text{disj_property}(?U, ?V) &\rightarrow \bot.
\end{aligned}$$

Given a graph pattern P and an RDF graph G, to compute $[\![P]\!]^{\mathrm{U}}_G$ we need to include $\tau_{\text{owl2ql\_core}}$ in the Datalog$^{\neg s}$ query P_{dat} defined in Section 5.1. More precisely, we need to add to the program of P_{dat} the program $\tau_{\text{owl2ql\_core}}$, but taking into consideration the active domain semantics in the entailment regime just defined. For example, assume that P is the basic graph pattern $(?X, \text{eats}, \_ : B)$ and G is the RDF graph in (14) storing information about animals. Then $\tau_{\text{bgp}}(P)$ is the following rule:

$$\text{triple}(?X, \text{eats}, ?Y) \rightarrow \text{query}_P(?X). \tag{17}$$

In order to combine this rule with $\tau_{\text{owl2ql\_core}}$, we first need to consider the fact that all the triples inferred by using the axioms in G are stored in the predicate $\text{triple}_1(\cdot, \cdot, \cdot)$. Thus, we need to replace $\text{triple}(\cdot, \cdot, \cdot)$ by $\text{triple}_1(\cdot, \cdot, \cdot)$ in (17). We also need to enforce the constraint that every variable and blank node in P can only take a value from G, which is done by including the predicate C:

$$\text{triple}_1(?X, \text{eats}, ?Y), \text{C}(?X), \text{C}(?Y) \rightarrow \text{query}_P(?X). \tag{18}$$

Thus, given a graph pattern P, let $\tau^{\mathrm{U}}_{\text{bgp}}(P)$ be the set of rules obtained from $\tau_{\text{bgp}}(P)$ by first replacing triple by triple_1 in every rule of $\tau_{\text{bgp}}(P)$, and then adding $\text{C}(?X)$ in the body of every resulting rule ρ if $?X$ occurs in ρ. Finally, we define

$$P^{\mathrm{U}}_{\text{dat}} = (\tau_{\text{owl2ql\_core}} \cup \tau^{\mathrm{U}}_{\text{bgp}}(P) \cup \tau_{\text{opr}}(P) \cup \tau_{\text{out}}(P), \text{answer}_P).$$

Then it is possible to prove that

Theorem 5.3. *For every graph pattern P and RDF graph G that represents an OWL 2 QL core ontology, $[\![P]\!]^{\mathrm{U}}_G = [\![(P^{\mathrm{U}}_{\text{dat}}, \tau_{\text{db}}(G))]\!]$.*

Interestingly, after a careful analysis of the syntax of the query $P^{\mathrm{U}}_{\text{dat}}$, we observe that

Corollary 5.4. *For every graph pattern P, $P^{\mathrm{U}}_{\text{dat}}$ is a* TriQ 1.0 *query.*

Before we proceed further, we would like to stress the fact that the program $\tau_{\text{owl2ql\_core}}$, which is responsible for encoding the semantics $[\![\cdot]\!]^{\mathrm{U}}_G$ for basic graph patterns, is fixed and does not depend on the given graph pattern P. This implies that, for a new graph pattern P', we only need to compute the programs $\tau^{\mathrm{U}}_{\text{bgp}}(P')$, $\tau_{\text{opr}}(P')$, and $\tau_{\text{out}}(P')$ without altering $\tau_{\text{owl2ql\_core}}$. This is quite beneficial since, whenever the user wants to pose a new query, (s)he can use $\tau_{\text{owl2ql\_core}}$ as a black box.

5.3 Removing the Active Domain Restriction

Consider the basic graph pattern

$$Q = \{(?X, \text{eats}, \_:B), (\_:B, \text{rdf:type}, \text{plant\_material})\},$$

which asks for the lists of animals that eat some plant material, and assume that G is an RDF graph. Under the active domain semantics, a is an answer to Q over G if we can replace the blank node $\_:B$ by a specific plant material b such that G implies $(?X, \text{eats}, b)$. But what happens if such a concrete witness cannot be found in G, and we can only infer that a is an answer to Q by using the axioms in the ontology? For example, this could happen if G stores information only about herbivores, so it includes the axiom $(\exists\text{eats}^-, \text{rdfs:subClassOf}, \text{plant\_material})$. In this case, Q has to be replaced by a basic graph pattern of the form

$$\{(?X, \text{rdf:type}, \exists\text{eats}), (\exists\text{eats}^-, \text{rdfs:subClassOf}, \text{plant\_material})\}$$

in order to obtain the correct answers. And even worse, what happens if the query has to be distributed over several RDF graphs, which is a very common scenario in the Web. Then the user is forced to use a graph pattern of the form

$$\{(?X, \text{eats}, \_:B), (\_:B, \text{rdf:type}, \text{plant\_material})\} \text{ UNION } \{(?X, \text{rdf:type}, \exists\text{eats}), (\exists\text{eats}^-, \text{rdfs:subClassOf}, \text{plant\_material})\},$$

in which some inferences have to be encoded. All these issues can be solved if we do not force $\_:B$ to take values only in G, as this allows us to use the initial basic graph pattern Q. This gives rise to the semantics $[\![P]\!]^{\text{All}}_G$ that is defined exactly as $[\![P]\!]^{\mathbf{U}}_G$, but considering every basic graph pattern as a conjunctive query, and treating blank nodes as existential variables that are not forced to take only values in G (they can take values in the interpretations of G).

At this point, one may be tempted to think that the semantics $[\![\cdot]\!]^{\text{All}}$ can be directly defined by transforming every basic graph pattern into a conjunctive query, which has to be evaluated over a DL ontology. In fact, this approach works well with our initial query Q, which can be transformed into the conjunctive query $\exists Y(\text{eats}(X, Y) \wedge \text{plant\_material}(Y))$. However, there are simple queries for which this approach does not work. For instance, consider the basic graph pattern $(?X, \text{rdfs:subClassOf}, \exists\text{eats})$. Given that $?X$ is used to store class names, this pattern cannot be transformed into a conjunctive query in order to define its semantics; instead, we need to replace $?X$ by every class name C, and then verify whether the inclusion $C \sqsubseteq \exists\text{eats}$ is implied by the DL ontology in order to define its semantics. Thus, the goal of this section is to show that the more natural semantics $[\![\cdot]\!]^{\text{All}}$ can be easily defined by using Datalog$^{\exists,\neg s,\bot}$, without the need to differentiate between variables that are used to store individuals, classes, or properties.

Given a basic graph pattern P, let $\tau^{\text{All}}_{\text{bgp}}(P)$ be the rule obtained from $\tau^{\mathbf{U}}_{\text{bgp}}(P)$ by removing every atom of the form $\text{C}(?X)$ such that $?X \notin \text{var}(P)$ (that is, every atom $\text{C}(?X)$ such that $?X$ is a variable associated to a blank node occurring in P). For example, assume that P is the basic graph pattern $(?X, \text{eats}, \_:B)$. Then $\tau^{\mathbf{U}}_{\text{bgp}}(P)$ is the rule (18), and thus $\tau^{\text{All}}_{\text{bgp}}(P)$ is the rule

$$\text{triple}_1(?X, \text{eats}, ?Y), \text{C}(?X) \rightarrow \text{query}_P(?X).$$

Moreover, given a graph pattern P, define $\tau^{\text{All}}_{\text{bgp}}(P)$ as the Datalog program consisting of the rules $\tau^{\text{All}}_{\text{bgp}}(P_i)$ for every basic graph pattern P_i occurring in P. Finally, we define

$$P^{\text{All}}_{\text{dat}} = (\tau_{\text{owl2ql\_core}} \cup \tau^{\text{All}}_{\text{bgp}}(P) \cup \tau_{\text{opr}}(P) \cup \tau_{\text{out}}(P), \text{answer}_P).$$

With this simple modification of $P^{\mathbf{U}}_{\text{dat}}$, we can formally define the semantics $[\![\cdot]\!]^{\text{All}}$.

Definition 5.5. Given a graph pattern P and an RDF graph G, define $[\![P]\!]_G^{\text{ALL}}$ as $[\![(P_{\text{dat}}^{\text{ALL}}, \tau_{\text{db}}(G))]\!]$.

We conclude by pointing out that $P_{\text{dat}}^{\text{ALL}}$ is a TriQ 1.0 query, for every graph pattern P. Thus, this query language is expressive enough to represent the OWL 2 core direct semantics entailment regime, even if the active domain restriction is not imposed.

6 A TRACTABLE QUERY LANGUAGE

TriQ 1.0 forms a natural language that embeds the fundamental features for querying RDF, as shown in Section 5. Unfortunately, Theorem 4.4 shows that this language is highly intractable in data complexity. The goal of this section is to identify a core sub-language of TriQ 1.0, dubbed TriQ-Lite 1.0, that is powerful enough for expressing every SPARQL query under the entailment regime for OWL 2 QL core, and ensures the tractability of query evaluation in data complexity.

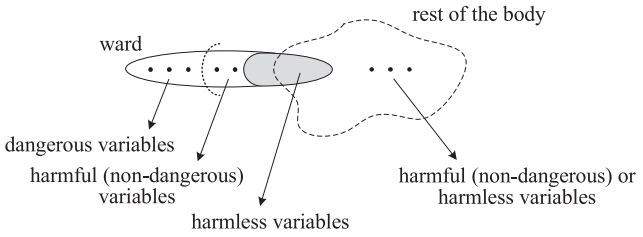
6.1 The Query Language TriQ-Lite 1.0

After a careful analysis of the program $\text{ex}(\Pi)$, where Π is the query program of $P_{\text{dat}}^{\text{U}}$ (or even $P_{\text{dat}}^{\text{ALL}}$) for an arbitrary graph pattern P, we observe that it enjoys an interesting property regarding the $\text{ex}(\Pi)^+$-dangerous variables: for each rule $\rho \in \text{ex}(\Pi)$, its dangerous variables are isolated in a single atom of $\text{body}^+(\rho)$, and they can interact with the rest of the rule-body only via $\text{ex}(\Pi)^+$-harmless variables. Another key observation is that the involved negation, apart from being stratified, is also grounded, i.e., it is used in front of predicates that can store only constants, but not nulls. Inspired by the above observations, we introduce a syntactic condition, called *wardedness*, that allows us to define TriQ-Lite 1.0, the sub-language of TriQ 1.0 that we are looking for.

A Datalog$^\exists$ program Π is *warded* if, for each rule $\rho \in \Pi$, either $\text{dangerous}(\rho, \Pi) = \varnothing$, or there exists an atom $\underline{a} \in \text{body}(\rho)$, called a *ward* and denoted by $\text{ward}(\rho)$, such that

(1) $\text{dangerous}(\rho, \Pi) \subseteq \text{var}(\underline{a})$, and
(2) $(\text{var}(\underline{a}) \cap \text{var}(\text{body}(\rho) \setminus \{\underline{a}\})) \subseteq \text{harmless}(\rho, \Pi)$.

Notice that the key difference between weakly-frontier-guarded Datalog$^\exists$ and warded Datalog$^\exists$ is the additional condition (2) in the definition of wardedness, which simply states that the guard can only share Π-harmless variables with the rest of the body. The body of a rule occurring in a warded Datalog$^\exists$ program Π can be graphically illustrated (via its hypergraph) as:



where the shaded part consists of Π-harmless variables, while the dashed area represents an arbitrary hypergraph. We can now define warded Datalog$^{\exists,\neg s,\bot}$ in the natural way. More precisely, a Datalog$^{\exists,\neg s,\bot}$ program Π is warded if the program $\text{ex}(\Pi)^+$ is warded, i.e., if the Datalog$^\exists$ program obtained from Π after eliminating the negative atoms and the constraints is warded.

Before introducing TriQ-Lite 1.0, which is based on warded Datalog$^{\exists,\neg s,\bot}$, we need the additional notion of grounded negation. A program Π is called Datalog$^{\exists,\neg sg,\bot}$ program ("*sg*" stands for

stratified and grounded) if, for each rule $\rho \in \Pi$, atom $p(t_1, \ldots, t_n) \in \text{body}^-(\rho)$, and $i \in [1, n]$, either $t_i \in \mathbf{U}$ or $t_i \in \text{harmless}(\rho, \text{ex}(\Pi)^+)$. We are now ready to introduce TriQ-Lite 1.0.

Definition 6.1. A TriQ-Lite 1.0 query is a Datalog$^{\exists,\neg sg,\bot}$ query that is warded.

TriQ-Lite 1.0 is powerful enough to express every SPARQL query under the entailment regime for OWL 2 QL core. In particular, it can be easily verified that, for every graph pattern P, both $P^{\mathbf{U}}_{\text{dat}}$ and $P^{\text{ALL}}_{\text{dat}}$ are TriQ-Lite 1.0 queries. This fact, together with Theorem 5.3, implies that:

Corollary 6.2. *Every graph pattern under the OWL 2 QL core direct semantics entailment regime (with or without the active domain restriction) can be expressed as a* TriQ-Lite 1.0 *query.*

At this point, one may be tempted to think that TriQ-Lite 1.0, and, in particular, the notion of wardedness, is ad-hoc and not well-justified. More precisely, in view of the fact that tractable sub-languages of weakly-frontier-guarded Datalog$^{\exists}$ already exist (details are given below), the next critical question comes up:

(1) Can we use a known tractable sub-language of weakly-frontier-guarded Datalog$^{\exists}$ to define TriQ-Lite 1.0? In other words, do we really need warded Datalog$^{\exists}$?

Furthermore, even if wardedness is essential for capturing SPARQL queries under the OWL 2 QL core direct semantics entailment regime, the next question comes up:

(2) Is warded Datalog$^{\exists}$ the best we can achieve? In other words, is there an obvious way to relax the wardedness condition without losing tractability?

The rest of this section is devoted to giving answers to the above questions. We show, via a model-theoretic argument, that a language based on one of the most expressive tractable sub-languages of weakly-frontier-guarded Datalog$^{\exists}$ would not be powerful enough for our purposes, and thus, warded Datalog$^{\exists}$ is essential (Section 6.2). We then proceed to establish that evaluation of TriQ-Lite 1.0 queries is tractable in data complexity (Section 6.3). Finally, we show that the mildest relaxation of warded Datalog$^{\exists}$ that one can think of, that is, at most one occurrence of exactly one harmful variable that occurs in the ward can appear also outside the ward, leads to an intractable language; more precisely, to an ExpTime-hard language (Section 6.4). This is a strong indication that there is no obvious way to extend warded Datalog$^{\exists}$ without losing tractability in data complexity.

6.2 Model-Theoretic Justification of Wardedness

A well-known tractable sub-language of weakly-frontier-guarded Datalog$^{\exists}$ is *frontier-guarded Datalog*$^{\exists}$ [6], where the guard must contain all the body variables that appear in the rule-head (and not only the dangerous body variables). A crucial limitation of this language is the fact that it is not able to compute the transitive closure of a binary relation. This has recently motivated the definition of a refined language, called *nearly frontier-guarded Datalog*$^{\exists}$, which allows for non-frontier-guarded rules as long as their body variables are harmless [24]. Formally, a Datalog$^{\exists}$ program Π is nearly frontier-guarded if, for each $\rho \in \Pi$, ρ is frontier-guarded or $\text{var}(\text{body}(\rho)) = \text{harmless}(\rho, \Pi)$. Although nearly frontier-guarded Datalog$^{\exists}$ is not widely known, it is considerably more expressive than frontier-guarded Datalog$^{\exists}$, while it remains tractable. Actually, it is currently the most expressive tractable sub-language of weakly-frontier-guarded Datalog$^{\exists}$.

We proceed to show that a query language based on nearly frontier-guarded Datalog$^{\exists}$ is not a good candidate for our purposes. But let us first clarify what we mean by saying a Datalog$^{\exists}$ language is a "good candidate." In the sequel, we call an OWL 2 QL core ontology *positive* if it does not contain axioms of the form $\text{DisjointClasses}(b_1, b_2)$ and $\text{DisjointObjectProperties}(r_1, r_2)$.

Definition 6.3. A Datalog$^{\exists}$ language $\mathbb{L}$ is a *good candidate* if there exists an $\mathbb{L}$ program Π such that, for every basic graph pattern P, and every RDF graph G that represents a positive OWL 2 QL core ontology, $[\![P]\!]_G^{\text{ALL}} = [\![(Q_\Pi, \tau_{\text{db}}(G))]\!]$, where $Q_\Pi = (\Pi \cup \tau_{\text{bgp}}^{\text{ALL}}(P) \cup \tau_{\text{out}}(P), \text{answer}_P)$.[9]

It is important to clarify that in the above definition we ask for a program Π in $\mathbb{L}$ that does the job for every P and every G since, as discussed in Section 5, it is vital to keep the program that encodes the semantics $[\![\cdot]\!]_G^{\text{ALL}}$ fixed. We would also like to stress that a Datalog$^{\exists}$ language $\mathbb{L}$ is a good candidate even if the query Q_Π does not fall in $\mathbb{L}$. The adoption of such a liberal definition allows us to keep independent the notion of the good candidate from the specific encodings of the programs $\tau_{\text{bgp}}^{\text{ALL}}(P)$ and $\tau_{\text{out}}(P)$. In other words, it would be conceptually misleading to classify a Datalog$^{\exists}$ language as a "bad candidate" only because the program $(\Pi \cup \tau_{\text{bgp}}^{\text{ALL}}(P) \cup \tau_{\text{out}}(P))$ does not syntactically fall in $\mathbb{L}$, as there might be different encodings of $\tau_{\text{bgp}}^{\text{ALL}}(P)$ and $\tau_{\text{out}}(P)$ such that $(\Pi \cup \tau_{\text{bgp}}^{\text{ALL}}(P) \cup \tau_{\text{out}}(P))$ is an $\mathbb{L}$ program. To sum up, Definition 6.3 states that a Datalog$^{\exists}$ language $\mathbb{L}$ is a good candidate if we are able to encode the semantics $[\![\cdot]\!]_G^{\text{ALL}}$ via a fixed $\mathbb{L}$ program. Then:

Proposition 6.4. *Nearly frontier-guarded Datalog$^{\exists}$ is not a good candidate.*

With the aim of showing that nearly frontier-guarded Datalog$^{\exists}$ is not a good candidate, we isolate a model-theoretic property, called unbounded ground-connection property, that is essential for a Datalog$^{\exists}$ language in order to be a good candidate. Roughly, a language $\mathbb{L}$ has this property if it allows us to connect, via a fixed program, an invented null value with an unbounded number of constants occurring in the underlying database. Given an instance I, the *ground connection* of a null $z \in (\text{dom}(I) \cap \mathbf{B})$, denoted $\text{gc}(z, I)$, is defined as the set of constants

$$\{c \in \mathbf{U} \mid \text{ there exists } \underline{a} \in I \text{ such that } \{c, z\} \subseteq \text{dom}(\underline{a})\},$$

i.e., all the constants that jointly appear with z in an atom of I. For a Datalog$^{\exists}$ program Π, and a family of databases $(D_n)_{n>0}$, we define the function

$$mgc(n) \;=\; \max_{z \in (\text{dom}(\Pi(D_n)) \cap \mathbf{B})} \{|\text{gc}(z, \Pi(D_n))|\};$$

if $(\text{dom}(\Pi(D_n)) \cap \mathbf{B}) = \varnothing$, then $mgc(n) = 0$. We say that a Datalog$^{\exists}$ language $\mathbb{L}$ has the *unbounded ground-connection property (UGCP)* if there exists a program Π in $\mathbb{L}$, and a family of databases $(D_n)_{n>0}$, such that $mgc(n) \notin O(1)$. The next lemma shows that the UGCP is essential for a Datalog$^{\exists}$ language in order to be a good candidate.

Lemma 6.5. *If a Datalog$^{\exists}$ language $\mathbb{L}$ is a good candidate, then $\mathbb{L}$ has the UGCP.*

Proof. Let $\mathcal{O}_n$, where $n > 0$, be the positive OWL 2 QL core ontology consisting of

$$\text{ClassAssertion}(a_0, c), \text{SubClassOf}(a_0, \exists p), \text{SubClassOf}(\exists p^-, a_1),$$
$$\text{SubClassOf}(a_1, a_2), \ldots, \text{SubClassOf}(a_{n-1}, a_n),$$

and let G_n be the RDF graph obtained after translating $\mathcal{O}_n$ into RDF. Let also P_n, where $n > 0$, be the basic graph pattern

$$\{(\_ : B, \text{rdf:type}, a_1), \ldots, (\_ : B, \text{rdf:type}, a_n)\},$$

[9] Notice that if we go beyond basic graph patterns and positive ontologies, then a Datalog$^{\exists}$ language is trivially not a good candidate since the features $\neg sg$ and $\bot$ are not available. Moreover, $\tau_{\text{opr}}(P)$ is empty, and this is the reason why it is not included in the definition of Q_Π.

where $\_ : B$ is a blank node, which simply asks whether there exists an object that belongs to the classes $a_1, \ldots, a_n$. Since, by hypothesis, $\mathbb{L}$ is a good candidate, there exists an $\mathbb{L}$ program Π such that $[\![P_n]\!]^{\text{ALL}}_{G_n} = [\![(Q_\Pi, \tau_{\text{db}}(G_n))]\!]$, where $n > 0$ and $Q_\Pi = (\Pi \cup \tau^{\text{ALL}}_{\text{bgp}}(P_n) \cup \tau_{\text{out}}(P_n), \text{answer}_{P_n})$. The latter implies that $\Pi(\tau_{\text{db}}(G_n))$ contains the atoms

$$\text{triple}(z, \text{rdf:type}, a_1), \ldots, \text{triple}(z, \text{rdf:type}, a_n),$$

where $z \in (\text{dom}(\Pi(\tau_{\text{db}}(G_n))) \cap \mathbf{B})$. Observe that $|\text{gc}(z, \Pi(\tau_{\text{db}}(G_n)))| = n$, which implies that, for the program Π, and the family of databases $(\tau_{\text{db}}(G_n))_{n>0}$, $mgc(n) \notin O(1)$. Thus, $\mathbb{L}$ has the UGCP. □

Having Lemma 6.5 in place, to establish Proposition 6.4 it remains to show that:

LEMMA 6.6. *Nearly frontier-guarded* Datalog$^\exists$ *does not have the UGCP.*

PROOF. Let Π be a nearly frontier-guarded Datalog$^\exists$ program, and $(D_n)_{n>0}$ a family of databases. Assume that $\Pi(D_n) = \bigcup_{i \geq 0} I_i$, where $I_i \langle \rho_i, h_i \rangle I_{i+1}$ is a chase sequence of D_n and Π; notice that, since Π is a Datalog$^\exists$ program, $\Pi(D_n) = \text{chase}(D_n, \Pi)$. By construction, for each null z in $\Pi(D_n)$, there exists $k_z > 0$ such that $z \notin \text{dom}(I_{k_z})$ and $z \in \text{dom}(I_{k_z+1})$. Let $I_{k_z+1} \setminus I_{k_z} = \{p(t_1, \ldots, t_m)\}$, i.e., $p(t_1, \ldots, t_m)$ is the atom in which z was invented. We claim that $|\text{gc}(z, \Pi(D_n))| \leq m + C_\Pi$, where C_Π is the number of constants in Π. Toward a contradiction, assume that $|\text{gc}(z, \Pi(D_n))| > m + C_\Pi$. This implies that there exists $i > k_z$, and a constant $c \in \text{dom}(D_n)$ that does not occur in $p(t_1, \ldots, t_m)$ or in Π, such that $\{c, z\} \subseteq \text{dom}(\underline{a})$, where $I_{i+1} \setminus I_i = \{\underline{a}\}$. In simple words, during the chase step $I_i \langle \rho_i, h_i \rangle I_{i+1}$ the rule ρ_i puts together in $\underline{a}$ the constant c and the null z. It is easy to verify that this can only be done via a non-frontier-guarded rule of Π since, after the application of a frontier-guarded rule ρ, z can jointly appear in the generated atom with constants in $p(t_1, \ldots, t_m)$ and $\text{head}(\rho)$. Therefore, ρ_i is a non-frontier-guarded rule. But this implies that $h_i(\text{body}(\rho_i))$ contains only constants since the body variables of ρ are Π-harmless, and thus, $z \notin \text{dom}(\underline{a})$. This contradicts the fact that $\{c, z\} \subseteq \text{dom}(\underline{a})$, and thus, $|\text{gc}(z, \Pi(D_n))| \leq m + C_\Pi$. Hence, $mgc(n) \in O(1)$, which in turn implies that nearly frontier-guarded Datalog$^\exists$ does not have the UGCP. □

6.3 The Complexity of TriQ-Lite 1.0

Interestingly, TriQ-Lite 1.0 queries can be evaluated in polynomial time in the size of the database.

THEOREM 6.7. EVAL *for* TriQ-Lite 1.0 *is* PTIME*-complete in data complexity.*

It is easy to verify that every Datalog program is a warded Datalog$^{\exists, \neg sg, \bot}$ program. More precisely, given a Datalog program Π, since $\text{affected}(\Pi) = \varnothing$, we conclude that for every rule $\rho \in \Pi$, $\text{dangerous}(\rho, \Pi) = \varnothing$, which in turn implies that Π is trivially warded. Therefore, every Datalog query is a TriQ-Lite 1.0 query. This allows us to deduce the lower bound in Theorem 6.7, as the query evaluation problem for Datalog is PTIME-hard in data complexity (see, e.g., [19]). The rest of this subsection is devoted to establishing the membership of our problem in PTIME.

Consider a database D and a (fixed) TriQ-Lite 1.0 query $Q = (\Pi, p)$. As discussed in the proof of Theorem 4.4, for an arbitrary tuple $\mathbf{t} \in \mathbf{U}^{\text{arity}(p)}$,

$$Q(D) \neq \top \text{ implies } \mathbf{t} \in Q(D) \quad \text{iff} \quad (\star, \ldots, \star) \notin Q'(D) \text{ implies } \mathbf{t} \in Q'(D),$$

where $Q' = (\text{ex}(\Pi) \cup \Pi_\bot, p)$, and $\Pi_\bot$ is defined as the Datalog program

$$\{\underline{a}_1, \ldots, \underline{a}_n \rightarrow p(\star, \ldots, \star) \mid \underline{a}_1, \ldots, \underline{a}_n \rightarrow \bot \in \Pi\},$$

with $\star$ being a constant not in D or Π. By construction, Q' is a warded Datalog$^{\exists, \neg sg}$ query. Therefore, to establish the desired upper bound, it suffices to show that:

Proposition 6.8. Eval *for warded* Datalog$^{\exists,\neg sg}$ *is in* PTime *in data complexity.*

Consider an instance of Eval for warded Datalog$^{\exists,\neg sg}$, i.e., a database D, a warded Datalog$^{\exists,\neg sg}$ query $Q = (\Pi, p)$, and a tuple of constants $\mathbf{t}$. Our goal is to show that the problem of deciding whether $\mathbf{t} \in Q(D)$ is feasible in polynomial time in D. Notice that we focus on the problem whether $\mathbf{t} \in Q(D)$, without checking if $Q(D) \neq \top$, since $Q(D) \neq \top$ holds trivially due to the absence of constraints. The algorithm for checking whether $\mathbf{t} \in Q(D)$ consists of the following two steps.

Step 1 - Eliminate Negation. We construct a database $D_+ \supseteq D$ and eliminate the negation from the given query $Q = (\Pi, p)$ to produce $Q_+ = (\Pi_+, p)$ such that $Q(D) = Q_+(D_+)$. Since the negation in Π is stratified and grounded, Π_+ can be computed from Π in the standard way by replacing each negative atom $\neg s(\mathbf{t})$ with a positive atom $\bar{s}(\mathbf{t})$, where the relation $\bar{s}$ in D_+ stores the complement of s with respect to the *ground semantics* of Π over D, that is, the instance

$$\Pi(D)_{\downarrow} = \{\underline{a} \in \Pi(D) \mid \mathrm{dom}(\underline{a}) \subset \mathbf{U}\},$$

which collects all the atoms of $\Pi(D)$ with constants only. We proceed to formalize the above informal construction. Let $\sigma : \mathrm{sch}(\Pi) \to [0, \ell]$ be a stratification of Π, and let $\Pi_0, \ldots, \Pi_\ell$ be the partition of Π induced by σ. We denote by $(\Pi_i)_+$, where $i \in [1, \ell]$, the program obtained from Π_i by replacing each negative atom $\neg s(\mathbf{t})$ with the positive atom $\bar{s}(\mathbf{t})$. Let $\mathrm{sch}^-(\Pi_i)$ be the set of predicates occurring in Π_i in at least one negative atom. We inductively define $D^\star_\ell$ and $\Pi^\star_\ell$ as follows: $D^\star_0 = D$ and $\Pi^\star_0 = \Pi_0$; and for $i \in [1, \ell]$, let $D^\star_i = (D^\star_{i-1} \cup C_{i-1})$, where

$$C_{i-1} = \left\{ \bar{s}(\mathbf{t}) \;\middle|\; \begin{array}{l} s \in \mathrm{sch}^-(\Pi_i), \\ \mathbf{t} \in (\mathrm{dom}(D))^{\mathrm{arity}(p)}, \\ s(\mathbf{t}) \notin \Pi^\star_{i-1}(D^\star_{i-1})_{\downarrow} \end{array} \right\},$$

and $\Pi^\star_i = \Pi^\star_{i-1} \cup (\Pi_i)_+$. Let $D_+ = D^\star_\ell$ and $\Pi_+ = \Pi^\star_\ell$.

Step 2 - Scan the Ground Semantics. We simply check whether the atom $p(\mathbf{t})$ belongs to the ground semantics of Π_+ over D_+. Formally, if $p(\mathbf{t}) \in \Pi_+(D_+)_{\downarrow}$, then accept; otherwise, reject.

It is not difficult to verify that the above algorithm is correct. In fact, by construction, $Q(D) = Q_+(D_+)$, which in turn implies that $\mathbf{t} \in Q(D)$ iff the algorithm accepts. However, at this point, it is not clear whether the above algorithm runs in polynomial time. This depends on the complexity of computing the ground semantics of a program over a database. Observe that during the computation of the algorithm, we are always interested in the ground semantics of a warded Datalog$^\exists$ program (without negation) over a database. Moreover, it is easy to verify that, if the ground semantics of a warded Datalog$^\exists$ over a database D can be computed in polynomial time in D, then the above algorithm runs in polynomial time in D. Consequently, to establish Proposition 6.8, it suffices to show the following crucial technical lemma:

Lemma 6.9. *Consider a database D, and a warded Datalog$^\exists$ program Π. The instance $\Pi(D)_{\downarrow}$ can be constructed in polynomial time in D.*

It is easy to see that the size of $\Pi(D)_{\downarrow}$ is polynomial in the size of D. More precisely, $|\Pi(D)_{\downarrow}| \leq |\mathrm{sch}(\Pi)| \cdot |\mathrm{dom}(D)|^{\mathrm{arity}(\Pi)}$, that is, the maximum number of ground atoms that can be formed using predicates of $\mathrm{sch}(\Pi)$ and constants of $\mathrm{dom}(D)$. Hence, to establish our claim, it suffices to show that the problem of deciding whether a ground atom $p(\mathbf{t})$, where $p \in \mathrm{sch}(\Pi)$ and $\mathbf{t} \in \mathrm{dom}(D)^{\mathrm{arity}(p)}$, belongs to $\Pi(D)$ is feasible in polynomial time in D. The rest of this subsection is devoted to establishing this rather involved result.

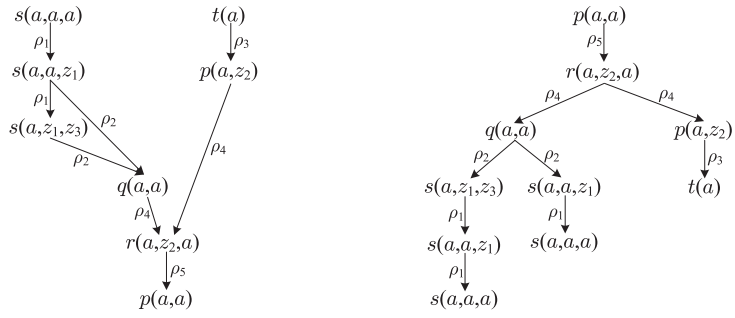


Fig. 1. Proof and proof-tree from Example 6.10.

A crucial notion in our analysis is the so-called proof-tree of $p(\mathbf{t})$ with respect to D and Π, introduced in [15].[10] Such a proof-tree (if it exists) can be conceived as a tree-like representation of the proof of $p(\mathbf{t})$ with respect to D and Π, that is, the part of $\Pi(D)$ which entails $p(\mathbf{t})$. Before we proceed further, let us illustrate the notion of the proof-tree via a simple example.

Example 6.10. Consider the warded Datalog$^{\exists}$ program Π:

$$\rho_1 = s(?X, ?Y, ?Z) \rightarrow \exists ?W\, s(?X, ?Z, ?W),$$
$$\rho_2 = s(?X, ?Y, ?Z), s(?Y, ?Z, ?W) \rightarrow q(?X, ?Y),$$
$$\rho_3 = t(?X) \rightarrow \exists ?Z\, p(?X, ?Z),$$
$$\rho_4 = p(?X, ?Y), q(?X, ?Z) \rightarrow r(?X, ?Y, ?Z),$$
$$\rho_5 = r(?X, ?Y, ?Z) \rightarrow p(?X, ?Z),$$

the database $D = \{s(a, a, a), t(a)\}$, and the ground atom $p(a, a)$. A proof and a proof-tree of $p(a, a)$ with respect to D and Π are given in Figure 1(a) and (b), respectively. Observe that a proof of $p(a, a)$ with respect to D and Π encodes which rules must be applied during the construction of chase(D, Π) in order to entail $p(a, a)$. A proof-tree is a tree-like representation of such a proof.

It is clear that $p(\mathbf{t}) \in \Pi(D)$ iff $p(\mathbf{t})$ has a proof with respect to D and Π. Now, having a proof of $p(\mathbf{t})$ with respect to D and Π, we can construct a proof-tree of $p(\mathbf{t})$ by, roughly speaking, reversing the edges and unfolding the obtained graph into a tree by repeating some of the nodes. On the other hand, having a proof-tree of $p(\mathbf{t})$, we can construct a proof of $p(\mathbf{t})$ by reversing the edges and collapsing some of the nodes. Therefore, $p(\mathbf{t}) \in \Pi(D)$ iff $p(\mathbf{t})$ has a proof-tree with respect to D and Π. Thus, our problem is equivalent to the problem of deciding whether a proof-tree of $p(\mathbf{t})$ with respect to D and Π exists. We solve the latter problem via a recursive alternating algorithm that constructs a proof-tree P of $p(\mathbf{t})$ with respect to D and Π (if it exists) by building the branches of P in parallel universal computations. We proceed to formalize the above informal discussion.

For technical clarity, in the rest of this section, we focus on rules with at most one occurrence of an existentially quantified variable. This does not affect the generality of our proof since every warded Datalog$^{\exists}$ program Π can be transformed into a warded Datalog$^{\exists}$ program Π', where each rule contains at most one occurrence of an existentially quantified variable, that preserves all the ground atoms that can be inferred from Π. More precisely, given a rule ρ

$$\underline{a}_1, \ldots, \underline{a}_n \rightarrow \exists ?Y_1 \ldots \exists ?Y_k\ \underline{c},$$

[10]Notice that in [15] the term resolution proof-scheme is adopted. However, for the sake of readability, we prefer to use the more compact term proof-tree.

with $\mathbf{X} = \text{var}(\text{body}(\rho)) \cap \text{var}(\text{head}(\rho))$, we define $\mathsf{N}(\rho)$ as the set of rules

$$\begin{aligned}
\underline{a}_1, \ldots, \underline{a}_n &\rightarrow \exists ?Y_1 \; p_1^\rho(\mathbf{X}, ?Y_1), \\
p_1^\rho(\mathbf{X}, ?Y_1) &\rightarrow \exists ?Y_2 \; p_2^\rho(\mathbf{X}, ?Y_1, ?Y_2), \\
&\vdots \\
p_{k-1}^\rho(\mathbf{X}, ?Y_1, \ldots, ?Y_{k-1}) &\rightarrow \exists ?Y_k \; p_k^\rho(\mathbf{X}, ?Y_1, \ldots, ?Y_k), \\
p_k^\rho(\mathbf{X}, ?Y_1, \ldots, ?Y_k) &\rightarrow \underline{c},
\end{aligned}$$

where $p_1^\rho, \ldots, p_k^\rho$ are auxiliary predicates not occurring in Π. The program Π' is defined as $\bigcup_{\rho \in \Pi} \mathsf{N}(\rho)$. It is easy to verify that, if Π is warded, then also Π' is warded. Moreover, $\Pi(D)_\downarrow = \Pi'(D)_\downarrow$, for every database D. Given a rule ρ, let $\pi_\exists(\rho)$ be the position at which the existentially quantified variable occurs in ρ; $\pi_\exists(\rho) = \varepsilon$ if there is no existentially quantified variable in ρ.

Let us now recall the key notion of the proof-tree. To this end, we need to introduce some auxiliary notation and terminology. Given a Datalog$^\exists$ rule ρ and an atom $\underline{a} = p(t_1, \ldots, t_n)$, we say that ρ is *compatible* with $\underline{a}$, written $\rho \rhd \underline{a}$, if the following two conditions hold: (i) there exists a homomorphism h such that $h(\text{head}(\rho)) = \underline{a}$, and (ii) for each $i \in [1, \text{arity}(p)]$, if $t_i \in \mathbf{U}$ or t_i occurs more than once in $\underline{a}$, then $\pi_\exists(\rho) \neq p[i]$. Observe that the homomorphism that maps $\text{head}(\rho)$ to $\underline{a}$ is unique, and we refer to it by $h_{\rho,\underline{a}}$. Given a set of terms T and a set of predicates X, let $\text{base}(T, X)$ be the set of atoms $\{p(\mathbf{t}) \mid p \in X \text{ and } \mathbf{t} \in T^{\text{arity}(p)}\}$, i.e., the atoms that can be formed using terms from T and predicates from X. We are now ready to recall the definition of the proof-tree of a ground atom with respect to a database and a program [15].

Definition 6.11. Consider a database D, a Datalog$^\exists$ program Π, and an atom $p(\mathbf{t})$ with $p \in \text{sch}(\Pi)$ and $\mathbf{t} \in \text{dom}(D)^{\text{arity}(p)}$. Let $P = (N, E, \lambda_N, \lambda_E)$ be a labeled rooted tree, where N is the node set, E is the edge set, $\lambda_N : N \rightarrow \text{base}(\text{dom}(D) \cup \mathbf{B}, \text{sch}(\Pi))$, and $\lambda_E : E \rightarrow \Pi$. P is a *proof-tree* of $p(\mathbf{t})$ with respect to D and Π if the following hold:

(1) If v is the root node of P, then $\lambda_N(v) = p(\mathbf{t})$.
(2) For each $v \in N$ with child nodes $u_1, \ldots, u_n$, there exists $\rho \in \Pi$ such that
  (a) for each $i \in [1, n]$, $\lambda_E((v, u_i)) = \rho$,
  (b) $\rho \rhd \lambda_N(v)$, and
  (c) there exists a bijective function $f : \text{body}(\rho) \rightarrow \{u_1, \ldots, u_n\}$ such that, for each $\underline{a} \in \text{body}(\rho)$, $\lambda_N(f(\underline{a})) = \gamma(\underline{a})$, where

$$\gamma \;=\; h_{\rho, \lambda_N(v)} \cup \{?V \rightarrow t \mid ?V \in \text{var}(\text{body}(\rho) \setminus \text{head}(\rho)) \text{ and } t \in (\text{dom}(D) \cup \mathbf{B})\}.$$

(3) Let $B_P = \bigcup_{v \in N} \{z \in \mathbf{B} \mid z \in \text{dom}(\lambda_N(v))\}$. For a null $z \in B_P$, we define the set of its *critical edges* as follows:

$$\text{critical}(z) = \left\{ e = (v, u) \in E \;\middle|\; \begin{array}{l} z \in (\text{dom}(\lambda_N(v)) \cap \mathbf{B}), \\ \pi_\exists(\lambda_E(e)) \neq \varepsilon, \\ z \text{ appears in } \lambda_N(v) \text{ at position } \pi_\exists(\lambda_E(e)) \end{array} \right\}.$$

  For each $z \in B_P$, and pairs $(v, u), (v', u') \in \text{critical}(z)$, it holds that $\lambda_N(v) = \lambda_N(v')$.
(4) For each leaf node $v \in N$, $\lambda_N(v) \in D$.

Let us clarify that the above definition is slightly different than the one in [15]. However, the two definitions are equivalent in the sense that an atom $\underline{a}$ has a proof-tree (adopting the definition in [15]) with respect to a database D and a program Π iff $\underline{a}$ has a proof-tree (adopting Definition 6.11) with respect to D and Π. The next lemma is implicit in [15].

Lemma 6.12. *Consider a database D, a Datalog$^\exists$ program Π, and an atom $p(\mathbf{t})$ with $p \in \text{sch}(\Pi)$ and $\mathbf{t} \in \text{dom}(D)^{arity(p)}$. Then $p(\mathbf{t}) \in \Pi(D)$ iff $p(\mathbf{t})$ has a proof-tree with respect to D and Π.*

The above lemma shows that our problem is equivalent to the problem of deciding whether $p(\mathbf{t})$ has a proof-tree with respect to D and Π. For technical clarity, we normalize even further the rules occurring in a warded Datalog$^\exists$ program Π so that every rule is *head-grounded*, i.e., each term in the head is either a constant or a harmless variable, or *semi-body-grounded*, i.e., there exists at most one body atom that contains a harmful variable. Consider a rule $\rho \in \Pi$ of the form

$$s_0(\mathbf{X}, \mathbf{Y}, \mathbf{Z}), s_1(\mathbf{Z}, \mathbf{W}), \ldots, s_n(\mathbf{Z}, \mathbf{W}) \rightarrow \exists \mathbf{U}\, t(\mathbf{X}, \mathbf{Y}', \mathbf{Z}', \mathbf{W}', \mathbf{U}),$$

where $n \geq 1$, $\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{W}$ are pairwise disjoint, $\text{dangerous}(\rho, \Pi) = \mathbf{X}$, $\mathbf{Y}' \subseteq \mathbf{Y}$, $\mathbf{Z}' \subseteq \mathbf{Z}$, and $\mathbf{W}' \subseteq \mathbf{W}$. Clearly, due to wardedness, $\mathbf{Z}$ and $\mathbf{W}'$ are Π-harmless variables. Let $\mathsf{N}(\rho)$ be the set of rules

$$s_1(\mathbf{Z}, \mathbf{W}), \ldots, s_n(\mathbf{Z}, \mathbf{W}) \rightarrow t_\rho(\mathbf{Z}, \mathbf{W}'), \tag{19}$$

$$s_0(\mathbf{X}, \mathbf{Y}, \mathbf{Z}), t_\rho(\mathbf{Z}, \mathbf{W}') \rightarrow \exists \mathbf{U}\, t(\mathbf{X}, \mathbf{Y}', \mathbf{Z}', \mathbf{W}', \mathbf{U}), \tag{20}$$

where t_ρ is an auxiliary predicate not occurring in $\text{sch}(\Pi)$. Let $\Pi' = \Pi_1 \cup \bigcup_{\rho \in \Pi \setminus \Pi_1} \mathsf{N}(\rho)$, where Π_1 is the set of rules of Π with one body atom only. It is clear that each variable in the head of (19) is Π'-harmless, while in the body of (20) only the atom $s_0(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ contains Π'-harmful variables. Moreover, $\Pi(D)_\downarrow = \Pi'(D)_\downarrow$, for every database D. We are now ready to present our alternating algorithm ProofTree. Let us first give a high-level description of it.

A High-Level Description of ProofTree

The algorithm ProofTree accepts as input a database D, a warded Datalog$^\exists$ program Π, and an atom $p(\mathbf{t})$, where $\mathbf{t}$ is a tuple of constants of $\text{dom}(D)$. As explained above, Π can be normalized in such a way that each rule is head-grounded or semi-body-grounded. Henceforth, we assume that Π is in normal form, and we write Π^h and Π^b for the head-grounded and the semi-body-grounded rules of Π, respectively. ProofTree starts from $p(\mathbf{t})$ and applies resolution steps until the database D is reached. It consists of the following steps:

- If $p(\mathbf{t}) \in D$, then accept; otherwise, a rule $\rho \in \Pi^h$ such that $\rho \rhd p(\mathbf{t})$ is guessed. After resolving $p(\mathbf{t})$ with ρ we get the set of atoms $\gamma(\text{body}(\rho))$, where γ extends $h_{\rho, p(\mathbf{t})}$ by mapping the variables in the body but not in the head of ρ to $(\text{dom}(D) \cup \mathbf{B})$.
- The set $\gamma(\text{body}(\rho))$ is partitioned into $\{S_1, \ldots, S_n\}$ in such a way that, for each null z occurring in $\gamma(\text{body}(\rho))$, there exists exactly one $i \in [1, n]$ such that S_i contains z, and there is no partition of $\gamma(\text{body}(\rho))$ with $n+1$ elements that satisfies the latter condition, i.e., each element of $\{S_1, \ldots, S_n\}$ is $\subseteq$-minimal. The intention underlying the above partitioning step is to keep together, in a parallel universal computation of the alternating algorithm, the nulls that appear in $\gamma(\text{body}(\rho))$, until the atom in which they are invented is known. This is vital for ensuring the compatibility of the various branches that are built in parallel computations.
- Universally select each set $S \in \{S_1, \ldots, S_n\}$ and prove it. In fact, if S consists of a single atom $p'(\mathbf{t}')$, where $\mathbf{t}'$ is a tuple of constants, then we recursively call $\text{ProofTree}(D, \Pi, p'(\mathbf{t}'))$; otherwise, we proceed as follows.
- For each atom $\underline{a} \in S$, a rule $\rho_{\underline{a}} \in \Pi^b$ is guessed such that $\rho_{\underline{a}} \rhd \underline{a}$, and the set of atoms $\gamma_{\underline{a}}(\text{body}(\rho_{\underline{a}}))$, where $\gamma_{\underline{a}}$ extends $h_{\rho_{\underline{a}}, \underline{a}}$ by mapping the variables that appear in the body but not in the head of $\rho_{\underline{a}}$ to $(\text{dom}(D) \cup \mathbf{B})$, is obtained.
- The set $\bigcup_{\underline{a} \in S} \gamma_{\underline{a}}(\text{body}(\rho_{\underline{a}}))$ is partitioned as above, and each component of the partition is proved in a parallel universal computation as done for $\{S_1, \ldots, S_n\}$.

During the execution of the above procedure, the first time that a null z is lost after resolving an atom $\underline{a}$ (that contains z) with a rule $\rho \in \Pi$, which means that z is associated with the existentially quantified variable in head(ρ), we store $h_{\rho,\underline{a}}(\text{head}(\rho))$ as the atom where z is invented. It is vital to ensure that the atoms where z is invented in parallel computations are precisely $h_{\rho,\underline{a}}(\text{head}(\rho))$. This is achieved by carrying the atom $h_{\rho,\underline{a}}(\text{head}(\rho))$ together with the component that contains z.

The Formal Definition of ProofTree

Before formalizing the above algorithm, we need to introduce an additional auxiliary notion. Consider a set of atoms S such that $\text{dom}(S) \subset (\mathbf{U} \cup \mathbf{B})$, and a set $N \subseteq (\text{dom}(S) \cap \mathbf{B})$. A partition $\{S_1, \ldots, S_n\}$ of S is called $[N]$-*linking* if, for each $z \in (\text{dom}(S) \cap \mathbf{B}) \setminus N$, there exists exactly one $i \in [1, n]$ such that $z \in \text{dom}(S_i)$. Moreover, $\{S_1, \ldots, S_n\}$ is called $[N]$-*optimal* if (i) it is $[N]$-linking, and (ii) for every $i \in [1, n]$ and $\underline{a} \in S_i$, the partition $\{S_1, \ldots, S_{i-1}, S_i \setminus \{\underline{a}\}, S_{i+1}, \ldots, S_n, \{\underline{a}\}\}$ of S is not $[N]$-linking. Consider, for example, the set $S = \{p(c, z_1), p(z_1, z_2), p(z_2, z_3), p(z_3, z_4)\}$, where $c \in \mathbf{U}$ and $z_1, z_2, z_3, z_4 \in \mathbf{B}$, and let $N = \{z_2, z_3, z_4\}$. The partition $\{\{p(c, z_1), p(z_1, z_2)\}, \{p(z_2, z_3), p(z_3, z_4)\}\}$ is $[N]$-linking since $z_1 \in (\text{dom}(S) \cap \mathbf{B}) \setminus N$ occurs in exactly one component. However, it is not $[N]$-optimal since the partition $\{\{p(c, z_1), p(z_1, z_2)\}, \{p(z_2, z_3)\}, \{p(z_3, z_4)\}\}$ is still $[N]$-linking. In fact, the latter partition is $[N]$-optimal since, once we split the component $\{p(c, z_1), p(z_1, z_2)\}$, the obtained partition is not $[N]$-linking. We are now ready to formalize our alternating algorithm.

ProofTree$(D, \Pi, p(\mathbf{t}))$ consists of the following steps:

(1) If $p(\mathbf{t}) \in D$, then accept.
(2) Guess a rule $\rho \in \Pi^h$ such that $\rho \rhd p(\mathbf{t})$; if there is no such rule, then reject.
(3) Guess a mapping $\mu : \text{var}(\text{body}(\rho)) \setminus \text{var}(\text{head}(\rho)) \to (\text{dom}(D) \cup \mathbf{B})$, and let $\gamma = h_{\rho,p(\mathbf{t})} \cup \mu$.
(4) Let $\{S_1, \ldots, S_n\}$ be the $[\varnothing]$-optimal partition of $\gamma(\text{body}(\rho))$.
(5) Universally select $S \in \{S_1, \ldots, S_n\}$ and do the following:
  (a) if $S = \{p'(\mathbf{t}')\}$ and $\text{dom}(p'(\mathbf{t}')) \subseteq \text{dom}(D)$, then call ProofTree$(D, \Pi, p'(\mathbf{t}'))$;
  (b) if $(\text{dom}(S) \cap \mathbf{B}) \neq \varnothing$, then go to (6).
(6) $R_S := \{(z, \varepsilon) \mid z \in (\text{dom}(S) \cap \mathbf{B})\}$.
(7) For each $\underline{a} \in S$ do the following:
  (a) Guess a rule $\rho_{\underline{a}} \in \Pi^b$ such that $\rho_{\underline{a}} \rhd \underline{a}$; if there is no such rule, then reject.
  (b) Assume that $z \in (\text{dom}(\underline{a}) \cap \mathbf{B})$ occurs in $\underline{a}$ at position $\pi_\exists(\rho_{\underline{a}})$, and $(z, x) \in R_S$. If $x = \varepsilon$, then $R_S := (R_S \setminus \{(z, \varepsilon)\}) \cup \{(z, \underline{a})\}$; otherwise, if $x \neq \underline{a}$, then reject.
  (c) Guess $\mu_{\underline{a}} : \text{var}(\text{body}(\rho_{\underline{a}})) \setminus \text{var}(\text{head}(\rho_{\underline{a}})) \to (\text{dom}(D) \cup \mathbf{B})$ such that $\text{dom}(\gamma_{\underline{a}}(\text{body}(\rho_{\underline{a}}) \setminus \{\text{ward}(\rho_{\underline{a}})\})) \subseteq \text{dom}(D)$, where $\gamma_{\underline{a}} = h_{\rho_{\underline{a}},\underline{a}} \cup \mu_{\underline{a}}$.
(8) $S^+ := \bigcup_{\underline{a} \in S} \gamma_{\underline{a}}(\text{body}(\rho_{\underline{a}}))$.
(9) $N := \{z \in (\text{dom}(S^+) \cap \mathbf{B}) \mid (z, x) \in R_S \text{ and } x \neq \varepsilon\}$.
(10) Let $\{S_1^+, \ldots, S_n^+\}$ be the $[N]$-optimal partition of S^+.
(11) $F := \{z \in \mathbf{B} \mid z \in \text{dom}(S^+) \setminus \text{dom}(S)\}$.
(12) For each $i \in [1, n]$, let $R_{S_i^+} = \{(z, x) \in R_S \mid z \in (\text{dom}(S_i^+) \cap \mathbf{B}) \setminus F\} \cup \{(z, \varepsilon) \mid z \in (\text{dom}(S_i^+) \cap F)\}$.
(13) Universally select $S \in \{S_1^+, \ldots, S_n^+\}$ and do the following:
  (a) If $S = \{p'(\mathbf{t}')\}$ and $\text{dom}(p'(\mathbf{t}')) \subseteq \text{dom}(D)$, then call ProofTree$(D, \Pi, p'(\mathbf{t}'))$.
  (b) If $(\text{dom}(S) \cap \mathbf{B}) \neq \varnothing$, then go to (7).

The correctness of the above algorithm follows by construction

Lemma 6.13. *Consider a database D, a warded Datalog$^\exists$ program Π, and an atom $p(\mathbf{t})$ with $p \in \text{sch}(\Pi)$ and $\mathbf{t} \in \text{dom}(D)^{\text{arity}(p)}$. The following are equivalent:*

(1) $\mathsf{ProofTree}(D, \Pi, p(\mathbf{t}))$ *accepts.*
(2) $p(\mathbf{t})$ *has a proof-tree with respect to* D *and* Π.

Recall that our goal is to show that the problem of deciding whether $p(\mathbf{t})$ belongs to $\Pi(D)$ is feasible in polynomial time in D. By Lemma 6.12 and Lemma 6.13, $p(\mathbf{t}) \in \Pi(D)$ iff $\mathsf{ProofTree}(D, \Pi, p(\mathbf{t}))$ accepts. It is well-known that alternating logarithmic space coincides with polynomial time. Therefore, it suffices to show the following:

Lemma 6.14. *Consider a database* D*, a warded Datalog*$^{\exists}$ *program* Π*, and an atom* $p(\mathbf{t})$ *with* $p \in \mathsf{sch}(\Pi)$ *and* $\mathbf{t} \in \mathsf{dom}(D)^{\mathsf{arity}(p)}$*. If* Π *is fixed, then* $\mathsf{ProofTree}(D, \Pi, p(\mathbf{t}))$ *uses* $O(\log(|\mathsf{dom}(D)|))$ *space at each step of its computation.*

Proof. We first show that the size of a component of an $[N]$-optimal partition computed during the execution of $\mathsf{ProofTree}(D, \Pi, p(\mathbf{t}))$ is at most $\max_{\rho \in \Pi}\{|\mathsf{body}(\rho)|\}$. This is done by induction on the number of partitioning steps that are being applied during a universal computation of ProofTree. It is clear that the first partitioning step is actually step (4), where the $[\varnothing]$-optimal partition $\{S_1, \ldots, S_n\}$ of a set of atoms $\gamma(\mathsf{body}(\rho))$, where $\rho \in \Pi$ and γ is a mapping $\mathsf{var}(\mathsf{body}(\rho)) \to (\mathsf{dom}(D) \cup \mathbf{B})$, is computed. Observe that, for each $i \in [1, n]$, $|S_i| \leq |\mathsf{body}(\rho)|$, and the claim follows. Consider now a component S' obtained during the i-th partitioning step, for $i > 1$. Observe that in this case, S' is actually obtained during step (10) of the algorithm, where the $[N]$-optimal partition of a set of atoms $S^+ = \bigcup_{\underline{a} \in S} \gamma_{\underline{a}}(\mathsf{body}(\rho_{\underline{a}}))$, where S is a component obtained during the $(i-1)$-th partitioning step, $\rho_{\underline{a}} \in \Pi$, $\gamma_{\underline{a}}$ is a mapping $\mathsf{var}(\mathsf{body}(\rho_{\underline{a}})) \to (\mathsf{dom}(D) \cup \mathbf{B})$, and $N \subseteq (\mathsf{dom}(S^+) \cap \mathbf{B})$, is computed. We claim that $|S'| \leq |S|$, which in turn implies that $|S'| \leq \max_{\rho \in \Pi}\{|\mathsf{body}(\rho)|\}$ since, by induction hypothesis, $|S| \leq \max_{\rho \in \Pi}\{|\mathsf{body}(\rho)|\}$. By construction, $\rho_{\underline{a}} \in \Pi^b$, i.e., is a semi-body-grounded rule of Π. This implies that, for each $\underline{a} \in S$, only one atom $\underline{a}^\star$ of $\gamma_{\underline{a}}(\mathsf{body}(\rho_{\underline{a}}))$ may contain nulls, while all the other atoms contain only constants. Assuming that $S = \{\underline{a}_1, \ldots, \underline{a}_m\}$, it is easy to verify that the largest component that we can have in the $[N]$-optimal partition of S^+ is $\{\underline{a}_1^\star, \ldots, \underline{a}_m^\star\}$, while all the other components consist of a single atom. Thus, $|S'| \leq |S|$.

Having a bound on the size of a set of atoms that belongs to an $[N]$-optimal partition computed during the execution of $\mathsf{ProofTree}(D, \Pi, p(\mathbf{t}))$, it is not difficult to bound the space needed at each step of its computation. In the worst case, we need to remember $(\max_{\rho \in \Pi}\{|\mathsf{body}(\rho)|\})^2$ due to step (8), where the set S^+ is computed. It is not difficult to see that the space needed to represent an atom depends polynomially on Π, and is logarithmic in $|\mathsf{dom}(D)|$. The same holds for a pair of the form (z, x), where z is a null and x is either ε or an atom. Therefore, assuming that Π is fixed, $\mathsf{ProofTree}(D, \Pi, p(\mathbf{t}))$ uses $O(\log(|\mathsf{dom}(D)|))$ space at each step of its computation. □

6.4 Complexity-Theoretic Justification of Wardedness

We conclude this section by justifying the design choices made in the definition of wardedness. To this end, we show that the mildest relaxation of warded Datalog$^{\exists}$ that one can think of leads to an inherently intractable language; in fact, to an ExpTime-hard language. This is a strong indication that there is no obvious way to extend warded Datalog$^{\exists}$ without losing tractability in data complexity. Recall that the key idea underlying wardedness is to collect all the dangerous body variables in a single body atom, the so-called ward, while this atom can share only harmless variables with the rest of the rule-body. In other words, the ward can interact with the rest of the rule-body only via harmless variables. The mildest relaxation of wardedness that one can propose is as follows: allow at most one occurrence of exactly one harmful variable $?V$ that occurs in the ward to appear outside the ward in an atom of the form $p(t_1, \ldots, t_{i-1}, ?V, t_{i+1}, \ldots, t_n)$, where each t_i is either a constant or a harmless variable; in this case, we say that the warded Datalog$^{\exists}$ program

is with minimal interaction. Formally, a warded Datalog$^{\exists}$ program Π is *with minimal interaction* if, for each rule $\rho \in \Pi$, where $\underline{a} \in \text{body}(\rho)$ is the ward, the following hold:

(1) $|\underbrace{(\text{var}(\underline{a}) \cap \text{var}(\text{body}(\rho) \setminus \{\underline{a}\})) \setminus \text{harmless}(\rho, \Pi)}_{B}| \leq 1$;
(2) if $B = \{?V\}$, then there exists at most one occurrence of $?V$ in $(\text{body}(\rho) \setminus \{\underline{a}\})$; and
(3) if $?V$ occurs in $\underline{b} \in (\text{body}(\rho) \setminus \{\underline{a}\})$, then $\text{var}(\underline{b}) \setminus \{?V\} \subseteq \text{harmless}(\rho, \Pi)$.

It is possible to show that query evaluation for warded Datalog$^{\exists}$ with minimal interaction is ExpTime-hard. This is done by simulating the behavior of an alternating Turing machine that uses linear space. Before we proceed further, let us recall the basics on alternating Turing machines.

An *alternating Turing machine* is a tuple $M = (S, \Lambda, \delta, s_0)$, where $S = S_\forall \uplus S_\exists \uplus \{s_a\} \uplus \{s_r\}$ is a finite set of states partitioned into universal states, existential states, an accepting state, and a rejecting state, Λ is the tape alphabet, $\delta \subseteq (S \times \Lambda) \times (S \times \Lambda \times \{-1, +1\})$ is the transition relation, and $s_0 \in S$ is the initial state. We assume that Λ contains a special blank symbol $\sqcup$. The symbols -1 and $+1$ denote the cursor directions left and right, respectively. A *computation tree* for M is a tree labeled by configurations, i.e., tape content, cursor position, and internal state, of M such that

(1) if node v is labeled by an existential configuration, then v has one child, labeled by one of the possible successor configurations;
(2) if v is labeled by a universal configuration, then v has one child for each possible successor configuration;
(3) the root is labeled by the initial configuration; and
(4) all leaves are labeled by accepting or rejecting configurations.

A computation tree is *accepting* if it is finite and all leaves are labeled by accepting configurations. We are now ready to show that

Theorem 6.15. Eval *for warded Datalog$^{\exists}$ with minimal interaction is ExpTime-hard in data complexity.*

Proof. The proof is by simulating the behavior of an alternating Turing machine M on input I that uses linear space. We assume, w.l.o.g., that M is well-behaved and never tries to read beyond its tape boundaries, and uses $n = |I|$ tape cells. We also assume that each configuration has exactly two successor configurations. Our goal is to construct in polynomial time a database D_M that depends on M, and a warded Datalog$^{\exists}$ query $Q = (\Pi, \text{accept}(\cdot))$ with minimal interaction that does not depend on M, such that M accepts on input I iff $Q(D_M) = \{(\iota)\}$, where ι is a special constant the represents the initial configuration of M.

The Predicates. We first describe the predicates that we are going to use in the definition of Π. These predicates, together with their semantic meaning, are as follows:

— config($?V$): $?V$ is a configuration;
— succ($?V, ?V_1, ?V_2$): $?V_1$ and V_2 are successor configurations of $?V$;
— follows($?V, ?V'$): $?V'$ is a successor configuration of $?V$;
— state($?S, ?V$): in configuration $?V$ the state is $?S$;
— previous-state($?S, ?V$): the state of the predecessor configuration of $?V$ is $?S$;
— cursor($?C, ?V$): in configuration $?V$ the cursor points to the cell $?C$;
— symbol($?A, ?C, ?V$): in configuration $?V$ the cell $?C$ contains the symbol $?A$;
— state-cursor-symbol($?S, ?C, ?A, ?V$): in configuration $?V$ the state is $?S$, and the cursor points to the cell $?C$ that contains the symbol $?A$;

—next-cell($?C, ?C'$): cell $?C'$ follows cell $?C$ on the tape;
—neq($?C, ?C'$): $?C$ and $?C'$ are different cells;
—next-symbol($?C, ?A, ?V$): in a successor configuration of $?V$ the cell $?C$ contains $?A$;
—exists($?S$): state $?S$ is existential;
—forall($?S$): state $?S$ is universal;
—accept($?V$): $?V$ is an accepting configuration;
—previous-accept($?V$): the predecessor configuration of $?V$ is an accepting configuration;
—sibling-accept($?V$): the sibling configuration of $?V$, that is, the one that has the same predecessor as $?V$, is an accepting configuration;
—both-siblings-accept($?V$): both $?V$ and its sibling configuration are accepting configurations;
—transition($S, A, S_1, A_1, M_1, S_2, A_2, M_2$): encodes $\delta(S, A) = ((S_1, A_1, M_1), (S_2, A_2, M_2))$.

Notice that the above set of predicates does not depend on M.

The Database. We now define the database D_M, which actually describes the initial configuration of M, and also stores the transition function of M. We use constants to identify the cells and states of M. In particular, we use the constant c_i for the i-th cell of the tape, and the constant s for the state s of M; recall that s_0 represents the initial state of M. Moreover, we use the constant ι for identifying the initial configuration of M. D_M is defined as the database

$$\begin{aligned}
&\{\text{config}(\iota), \text{state}(s_0, \iota), \text{cursor}(c_1, \iota)\} \\
&\quad \cup \{\text{symbol}(\alpha_i, c_i, \iota) \mid i \in [1, n] \text{ and } \alpha_i \text{ is the } i\text{-th symbol of the input string}\} \\
&\quad \cup \{\text{next-cell}(c_i, c_{i+1}) \mid i \in [1, n-1]\} \\
&\quad \cup \{\text{neq}(c_i, c_j) \mid i, j \in [1, n] \text{ and } i \neq j\} \\
&\quad \cup \{\text{exists}(s) \mid s \in S_\exists\} \cup \{\text{forall}(s) \mid s \in S_\forall\} \\
&\quad \cup \{\text{transition}(s, \alpha, s_1, \alpha_1, m_1, s_2, \alpha_2, m_2) \mid (s, \alpha) \to ((s_1, \alpha_1, m_1), (s_2, \alpha_2, m_2)) \in \delta\}.
\end{aligned}$$

Notice that D_M depends on M, and can be constructed in polynomial time.

The Program. We are now ready to define the fixed warded Datalog$^\exists$ program Π with minimal interaction. We start with the rule that generates the configurations of M:

$$\begin{aligned}
\text{config}(?V) \;\to\; \exists ?V_1 \exists ?V_2\; &\text{succ}(?V, ?V_1, ?V_2), \\
&\text{config}(?V_1), \text{config}(?V_2), \\
&\text{follows}(?V, ?V_1), \text{follows}(?V, ?V_2).
\end{aligned}$$

We also add rules that encode the transition function of M. For example, the transitions that move the cursor to the left in the first successor configuration, and to the right in the second successor configuration are encoded as follows:

$$\begin{aligned}
\text{transition}(?S, ?A, ?S_1, ?A_1, -1, ?S_2, ?A_2, +1), \\
\text{succ}(?V, ?V_1, ?V_2), \text{state-cursor-symbol}(?S, ?C, ?A, ?V), \\
\text{next-cell}(?C_1, ?C), \text{next-cell}(?C, ?C_2) \;\to \\
\text{state}(?S_1, ?V_1), \text{state}(?S_2, ?V_2), \\
\text{symbol}(?A_1, ?C, ?V_1), \text{symbol}(?A_2, ?C, ?V_2), \\
\text{cursor}(?C_1, ?V_1), \text{cursor}(?C_2, ?V_2).
\end{aligned}$$

Similar rules are used to encode all the possible moves of the cursor in the successor configurations. The auxiliary predicate state-cursor-symbol$(\cdot,\cdot,\cdot,\cdot)$, which allows us to write the above rule as a warded rule with minimal interaction, is defined via the rules

$$\text{state}(?S,?V),\text{cursor}(?C,?V) \rightarrow \text{state-cursor}(?S,?C,?V),$$
$$\text{state-cursor}(?S,?C,?V),\text{symbol}(?A,?C,?V) \rightarrow \text{state-cursor-symbol}(?S,?C,?A,?V).$$

It should not be forgotten that the cells that are not involved in the transition must keep their old values, which is encoded by the following rules:

$$\text{transition}(?S,?A,?S_1,?A_1,-1,?S_2,?A_2,+1),$$
$$\text{state-cursor-symbol}(?S,?C,?A,?V),\text{neq}(?C,?C'),\text{symbol}(?C',?A',?V) \rightarrow \text{next-symbol}(?C',?A',?V)$$

and

$$\text{follows}(?V,?V'),\text{next-symbol}(?C,?A,?V) \rightarrow \text{symbol}(?C,?A,?V').$$

Finally, we define when a configuration is accepting, which in turn will be used to conclude whether ι is accepting. This can be achieved by the following rules:

$$\begin{aligned}
\text{state}(s_a,?V) &\rightarrow \text{accept}(?V),\\
\text{follows}(?V,?V'),\text{state}(?S,?V) &\rightarrow \text{previous-state}(?S,?V'),\\
\text{succ}(?V,?V_1,?V_2),\text{accept}(?V_2) &\rightarrow \text{sibling-accept}(?V_1),\\
\text{succ}(?V,?V_1,?V_2),\text{accept}(?V_1) &\rightarrow \text{sibling-accept}(?V_2),\\
\text{accept}(?V),\text{sibling-accept}(?V) &\rightarrow \text{both-siblings-accept}(?V),\\
\text{previous-state}(?S,?V),\text{exists}(?S),\text{accept}(?V) &\rightarrow \text{previous-accept}(?V),\\
\text{previous-state}(?S,?V),\text{forall}(?S),\text{both-siblings-accept}(?V) &\rightarrow \text{previous-accept}(?V),\\
\text{follows}(?V,?V'),\text{previous-accept}(?V') &\rightarrow \text{accept}(?V).
\end{aligned}$$

This concludes the construction of the program Π.

Clearly, Π does not depend on M. Observe that, for each rule ρ introduced above, the Π-harmful variables that occur in ρ are the variables $?V, ?V_1, ?V_2$. It is then easy to verify that Π is indeed a warded Datalog$^{\exists}$ program with minimal interaction. Moreover, by construction, M accepts on input I iff $Q(D_M) = \{(\iota)\}$, and the claim follows. □

7 PROGRAM EXPRESSIVE POWER

As already discussed in Section 4.4, an important issue for a query language is to understand its expressive power, and, in particular, its expressiveness relative to other central and well-studied query languages; such a key language is Datalog. It is a common practice in database theory to study the expressiveness of a newly introduced query language $\mathbb{L}$ relative to Datalog, which in turn gives some insights about the kind of queries that can be expressed in $\mathbb{L}$. The goal of this section is to perform such a relative expressive power analysis for warded Datalog$^{\exists}$ and TriQ-Lite 1.0.

By using the results of Section 6.2, it is easy to show that Datalog is not a good candidate for our purposes. Indeed, given a Datalog program Π, the instance $\Pi(D)$ does not contain a null value, for every database D, which immediately implies that Datalog does not have the UGCP. Thus, by Lemma 6.5, Datalog is not a good candidate. On the other hand, the fact that $P^{\text{ALL}}_{\text{dat}}$ is a TriQ-Lite 1.0 query, for every graph pattern P, implies that warded Datalog$^{\exists}$ is a good candidate. This suggests that warded Datalog$^{\exists}$ is more expressive than plain Datalog. However, according to the classical

notion of expressive power, the languages in question are equally expressive. It can be shown that, for every warded Datalog$^\exists$ query Q_1, we can construct a Datalog query Q_2 such that Q_1 and Q_2 are equivalent, i.e., $Q_1(D) = Q_2(D)$, for every database D; the converse is trivial since a Datalog query is, by definition, a warded Datalog$^\exists$ query. Therefore, to formally show that warded Datalog$^\exists$ is more expressive than Datalog, we need to adopt a refined notion of expressive power, which allows us to classify query languages according to their expressive power on a finer scale.

By Definition 6.3, a Datalog$^\exists$ language $\mathbb{L}$ is a good candidate if we can encode the semantics $[\![\cdot]\!]^{\text{ALL}}_G$ via a *fixed* $\mathbb{L}$ program. Thus, intuitively speaking, the key advantage of warded Datalog$^\exists$ against Datalog is the fact that we can express more via a single program. This led us to introduce the refined notion of *program expressive power*. Consider a Datalog$^\exists$ language $\mathbb{L}$, and a Datalog$^\exists$ program Π. The program expressive power of Π relative to $\mathbb{L}$, denoted $\mathsf{Pep}_{\mathbb{L}}[\Pi]$, is defined as the set of triples $(D, \Lambda, \mathbf{t})$, where D is a database, Λ is a set of Datalog rules of the form $\underline{a}_1, \ldots, \underline{a}_n \rightarrow p(?X_1, \ldots, ?X_n)$ with p being an n-ary predicate that does not appear in Π or in the body of a rule of Λ, and $\mathbf{t} \in \mathbf{U}^n$, such that the query $Q = (\Pi \cup \Lambda, p)$ falls in $\mathbb{L}$, and $\mathbf{t} \in Q(D)$; the rules of Λ act as the output rules of the query Q. In simple words, $\mathsf{Pep}_{\mathbb{L}}[\Pi]$ collects the tuples $\mathbf{t}$ that can be inferred from a database D via an $\mathbb{L}$ query Q, where Π is the query program of Q excluding the output rules. Now, for a Datalog$^\exists$ language $\mathbb{L}$, it is natural to define its program expressive power as the set

$$\mathsf{Pep}[\mathbb{L}] = \{\mathsf{Pep}_{\mathbb{L}}[\Pi] \mid \Pi \text{ is an } \mathbb{L} \text{ program}\}.$$

Roughly, $\mathsf{Pep}[\mathbb{L}]$ is a family of sets of triples, where each of its members encodes the program expressive power of an $\mathbb{L}$ program relative to $\mathbb{L}$. Given two languages $\mathbb{L}_1$ and $\mathbb{L}_2$, we write $\mathbb{L}_1 \preceq_{\mathsf{Pep}} \mathbb{L}_2$ if $\mathsf{Pep}[\mathbb{L}_1] \subseteq \mathsf{Pep}[\mathbb{L}_2]$. Finally, we say that $\mathbb{L}_2$ is *more expressive (w.r.t. the program expressive power)* than $\mathbb{L}_1$, written $\mathbb{L}_1 \prec_{\mathsf{Pep}} \mathbb{L}_2$, if $\mathbb{L}_1 \preceq_{\mathsf{Pep}} \mathbb{L}_2 \not\preceq_{\mathsf{Pep}} \mathbb{L}_1$. We proceed to show that

THEOREM 7.1. *Datalog* $\prec_{\mathsf{Pep}}$ *warded Datalog*$^\exists$.

PROOF. For notational convenience, we write DAT for Datalog and WAR for warded Datalog$^\exists$. It is clear that $\mathsf{Pep}[\text{DAT}] \subseteq \mathsf{Pep}[\text{WAR}]$ since, by definition, a Datalog program is a warded Datalog$^\exists$ program, and, therefore, DAT $\preceq_{\mathsf{Pep}}$ WAR. It remains to show that WAR $\not\preceq_{\mathsf{Pep}}$ DAT, or, equivalently, $\mathsf{Pep}[\text{WAR}] \not\subseteq \mathsf{Pep}[\text{DAT}]$. Consider the database $D = \{p(c)\}$, and the warded Datalog$^\exists$ queries $Q_1 = (\Pi \cup \Lambda_1, q)$ and $Q_2 = (\Pi \cup \Lambda_2, q)$, where

$$\Pi = \{p(X) \rightarrow \exists Y\, s(X, Y)\}, \quad \Lambda_1 = \{s(X, Y) \rightarrow q\}, \quad \Lambda_2 = \{s(X, Y), p(Y) \rightarrow q\}.$$

Clearly, $() \in Q_1(D)$ and $() \notin Q_2(D)$. Hence, $(D, \Lambda_1, ()) \in \mathsf{Pep}_{\text{WAR}}[\Pi]$ and $(D, \Lambda_2, ()) \notin \mathsf{Pep}_{\text{WAR}}[\Pi]$, which in turn implies that Pep[WAR] contains a set of triples T such that $(D, \Lambda_1, ()) \in T$ and $(D, \Lambda_2, ()) \notin T$. We claim that $T \notin \mathsf{Pep}[\text{DAT}]$, which in turn implies that $\mathsf{Pep}[\text{WAR}] \not\subseteq \mathsf{Pep}[\text{DAT}]$, as needed. It is not difficult to see that, for every Datalog program Π', $() \in Q'_1(D)$ implies $() \in Q'_2(D)$, where $Q'_1 = (\Pi' \cup \Lambda_1, q)$ and $Q'_2 = (\Pi' \cup \Lambda_2, q)$. Thus, the triples $(D, \Lambda_1, ())$ and $(D, \Lambda_2, ())$ necessarily coexist in $\mathsf{Pep}_{\text{DAT}}[\Pi']$, for every Datalog program Π'. Thus, $T \notin \mathsf{Pep}[\text{DAT}]$. □

By providing a similar argument, we can show that

THEOREM 7.2. *Datalog*$^{\neg s, \bot}$ $\prec_{\mathsf{Pep}}$ TriQ-Lite 1.0.

Equipped with the above result, it is easy to show that TriQ-Lite 1.0 is more expressive (w.r.t. the program expressive power) than existing languages suitable for querying RDF graphs. Indeed, several query languages that enhance SPARQL with navigation capabilities and/or recursion mechanisms have been proposed, most notably nSPARQL [35], PSPARQL [2], recursive triple algebra [29], and NEMODEQ [39]. Each one of the above languages $\mathbb{L}$ is contained in Datalog$^{\neg s, \bot}$, in the sense

that every query in $\mathbb{L}$ can be expressed as a Datalog$^{\neg s,\perp}$ query. Thus, we can consider the Datalog version $\mathbb{L}^{\text{dat}}$ of $\mathbb{L}$ in order to compare the program expressive power of $\mathbb{L}$ and TriQ-Lite 1.0. From Theorem 7.2, we immediately conclude that

Corollary 7.3. *If $\mathbb{L}$ is nSPARQL, PSPARQL, recursive triple algebra or NEMODEQ, then $\mathbb{L}^{\text{dat}} \prec_{\text{Pep}}$* TriQ-Lite 1.0.

8 DISCUSSION

We considered the problem of bridging the gap between the existing RDF query languages and key features for querying RDF data such as reasoning capabilities, navigational capabilities, and a general form of recursion. A tractable Datalog-based query language has been proposed, called TriQ-Lite 1.0, which is expressive enough to encode every SPARQL query under the entailment regime for OWL 2 QL core. Moreover, this language allows us to formulate SPARQL queries in a simpler way, as it can easily encode a more natural notion of entailment.

Interestingly, the logical core of TriQ-Lite 1.0, that is, warded Datalog$^{\exists}$, has already found interesting applications in neighboring fields. Vadalog is a system for performing complex reasoning tasks such as those required in advanced knowledge graphs [9, 10]. It is Oxford's contribution to the VADA research project,[11] a joint effort of the universities of Oxford, Manchester, and Edinburgh, as well as around 20 industrial partners such as Meltwater, Banca d'Italia, and Neo4J. The logical core of the underlying Vadalog language is warded Datalog$^{\exists}$. Indeed, warded Datalog$^{\exists}$ turned out to be powerful enough for expressing all the tasks given by the industrial partners of VADA, while a recent analysis of it focusing on a practical implementation led to the reasoning algorithm around which the Vadalog system is built; for more details see [10]. Further investigation with the aim of identifying the space-efficient core of warded Datalog$^{\exists}$ has been recently carried out in [11].

8.1 Future Work

We are planning to investigate whether TriQ-Lite 1.0 is powerful enough to deal with the other two lightweight profiles of OWL 2, namely, OWL 2 EL and OWL 2 RL, and if not, how it can be extended in order to obtain a unique tractable Datalog-based language that can deal with all the three lightweight profiles of OWL 2 in a uniform way.

In this work, we considered the definition of SPARQL 1.0 that has been adopted in a large number of articles that formally study SPARQL under set semantics, starting from [34]. In the future we would like to consider SPARQL under bag semantics. A good starting point for such an investigation is the recent (still unpublished) work [12], which studies Datalog under bag semantics. Interestingly, it is shown that Datalog queries under bag semantics can be translated into warded Datalog$^{\exists}$ queries under set semantics, i.e., the logical core of TriQ-Lite 1.0 as studied in Section 6.

APPENDIX

A TRANSLATING SPARQL INTO DATALOG$^{\neg s}$: A FORMAL DESCRIPTION

Given a SPARQL graph pattern P, we show how P can be translated into a Datalog$^{\neg s}$ query $P_{\text{dat}} = (\Pi, \text{answer}_P)$, where Π is the union of three subprograms: $\tau_{\text{bgp}}(P)$ that encodes the basic graph patterns occurring in P, $\tau_{\text{opr}}(P)$ that represents the non-basic graph patterns occurring in P, and $\tau_{\text{out}}(P)$ that computes the output predicate answer_P.

[11] http://vada.org.uk/.

A.1 The Program $\tau_{\text{bgp}}(P)$: Encoding the Basic Graph Patterns

Assume first that $P = \{t_1, \ldots, t_n\}$ is a basic graph pattern such that $t_i = (u_i, v_i, w_i)$ for every $i \in [1, n]$, and $\text{var}(P) = \{?X_1, \ldots, ?X_k\}$. Then $\tau_{\text{bgp}}(P)$ is the Datalog program defined as follows. Assume that ς is a substitution such that for every symbol u occurring in P, $\varsigma(u) = u$ if $u \in (\mathbf{U} \cup \mathbf{V})$, and $\varsigma(u)$ is a fresh variable if u is a blank node. Then $\tau_{\text{bgp}}(P)$ is defined as

$$\text{triple}(\varsigma(u_1), \varsigma(v_1), \varsigma(w_1)), \ldots, \text{triple}(\varsigma(u_n), \varsigma(v_n), \varsigma(w_n)) \rightarrow \text{query}_P(?X_1, \ldots, ?X_k).$$

For example, if P_2 is the basic graph pattern mentioned in Example 5.1 (see Section 5.1), then $\tau_{\text{bgp}}(P_2)$ consists of the rule (7). Now assume that P is a graph pattern. Then $\tau_{\text{bgp}}(P)$ is the Datalog program consisting of the rules $\tau_{\text{bgp}}(P_i)$ for every basic graph pattern P_i occurring in P. For example, if P_3 is the graph pattern mentioned in Example 5.1, then $\tau_{\text{bgp}}(P_3)$ consists of the rules (8) and (9).

A.2 The Program $\tau_{\text{opr}}(P)$: Encoding the SPARQL Operators

Program $\tau_{\text{opr}}(P)$ is used to encode the semantics of the SPARQL operators occurring in P. For example, if P_3 is the graph pattern mentioned in Example 5.1 (see Section 5.1), then $\tau_{\text{opr}}(P_3)$ consists of the rules (10), (11), and (12). Before defining $\tau_{\text{opr}}(P)$, we need to introduce some auxiliary terminology. The set of free variables of P, denoted by $\text{fvar}(P)$, is recursively defined as follows:

- If P is a basic graph pattern, then $\text{fvar}(P) = \text{var}(P)$.
- If P is either $(P_1 \text{ AND } P_2)$ or $(P_1 \text{ OPT } P_2)$ or $(P_1 \text{ UNION } P_2)$, then $\text{fvar}(P) = \text{fvar}(P_1) \cup \text{fvar}(P_2)$.
- if P is $(P_1 \text{ FILTER } R)$, then $\text{fvar}(P) = \text{fvar}(P_1)$.
- If P if $(\text{SELECT } W\ P_1)$, then $\text{fvar}(P) = \text{fvar}(P_1) \cap W$.

Moreover, given a tuple of variables $(?X_1, \ldots, ?X_k)$, where $k \geq 1$, and given $\mathcal{I} \subseteq [1, k]$ such that $[1, k] \setminus \mathcal{I} = \{i_1, i_2, \ldots, i_j\}$ with $1 \leq i_1 < i_2 < \cdots < i_j \leq k$, define $\rho_{\mathcal{I}}(?X_1, \ldots, ?X_k)$ as the tuple of variables $(?X_{i_1}, ?X_{i_2}, \ldots, ?X_{i_j})$; notice that if $\mathcal{I} = [1, k]$, then $\rho_{\mathcal{I}}(?X_1, \ldots, ?X_k)$ is the empty tuple.

We are now ready to define $\tau_{\text{opr}}(P)$. If P is a basic graph pattern, then $\tau_{\text{opr}}(P) = \varnothing$. Assume now that P_1 and P_2 are graph patterns such that $\text{fvar}(P_1) = \{?X_1, \ldots, ?X_\ell, ?Y_1, \ldots, ?Y_m\}$, $\text{fvar}(P_2) = \{?X_1, \ldots, ?X_\ell, ?Z_1, \ldots, ?Z_n\}$, and $\{?X_1, \ldots, ?X_\ell\} = \text{fvar}(P_1) \cap \text{fvar}(P_2)$. We proceed by considering the five different syntactic forms that P can have:

(1) If $P = (P_1 \text{ AND } P_2)$, then $\tau_{\text{opr}}(P)$ is the program consisting of the rules in $(\tau_{\text{opr}}(P_1) \cup \tau_{\text{opr}}(P_2))$ together with the following rule for every $\mathcal{I}_1 \subseteq [1, \ell + m]$ and $\mathcal{I}_2 \subseteq [1, \ell + n]$:

$$\text{query}^{\mathcal{I}_1}_{P_1}(\rho_{\mathcal{I}_1}(?X_1, \ldots, ?X_\ell, ?Y_1, \ldots, ?Y_m)),$$
$$\text{query}^{\mathcal{I}_2}_{P_2}(\rho_{\mathcal{I}_2}(?X_1, \ldots, ?X_\ell, ?Z_1, \ldots, ?Z_n)) \rightarrow$$
$$\text{query}^{\mathcal{I}}_{P}(\rho_{\mathcal{I}}(?X_1, \ldots, ?X_\ell, ?Y_1, \ldots, ?Y_m, ?Z_1, \ldots, ?Z_n)),$$

where $\mathcal{I} = (\mathcal{I}_1 \cap \mathcal{I}_2 \cap [1, \ell]) \cup (\mathcal{I}_1 \cap [\ell + 1, \ell + m]) \cup \{m + k \mid k \in \mathcal{I}_2 \cap [\ell + 1, \ell + n]\}$.

(2) If $P = (P_1 \text{ UNION } P_2)$, then $\tau_{\text{opr}}(P)$ consists of the rules in $(\tau_{\text{opr}}(P_1) \cup \tau_{\text{opr}}(P_2))$ together with the following rules for every $\mathcal{I}_1 \subseteq [1, \ell + m]$ and $\mathcal{I}_2 \subseteq [1, \ell + n]$:

$$\text{query}^{\mathcal{I}_1}_{P_1}(\rho_{\mathcal{I}_1}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m)) \rightarrow \text{query}^{\mathcal{I}'_1}_{P}(\rho_{\mathcal{I}'_1}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m,?Z_1,\ldots,?Z_n))$$

and

$$\text{query}^{\mathcal{I}_2}_{P_2}(\rho_{\mathcal{I}_2}(?X_1,\ldots,?X_\ell,?Z_1,\ldots,?Z_n)) \rightarrow \text{query}^{\mathcal{I}'_2}_{P}(\rho_{\mathcal{I}'_2}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m,?Z_1,\ldots,?Z_n)),$$

where $\mathcal{I}'_1 = \mathcal{I}_1 \cup [\ell+m+1, \ell+m+n]$ and $\mathcal{I}'_2 = (\mathcal{I}_2 \cap [1,\ell]) \cup [\ell+1,\ell+m] \cup \{m+k \mid k \in \mathcal{I}_2 \cap [\ell+1,\ell+n]\}$.

(3) If $P = (P_1 \text{ OPT } P_2)$, then $\tau_{\text{opr}}(P)$ consists of the rules in $(\tau_{\text{opr}}(P_1) \cup \tau_{\text{opr}}(P_2))$ together with the following rules for every $\mathcal{I}_1 \subseteq [1, \ell+m]$ and $\mathcal{I}_2 \subseteq [1, \ell+n]$:

$$\text{query}^{\mathcal{I}_1}_{P_1}(\rho_{\mathcal{I}_1}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m)), \text{query}^{\mathcal{I}_2}_{P_2}(\rho_{\mathcal{I}_2}(?X_1,\ldots,?X_\ell,?Z_1,\ldots,?Z_n)) \rightarrow \text{query}^{\mathcal{I}}_{P}(\rho_{\mathcal{I}}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m,?Z_1,\ldots,?Z_n)), \text{compatible}^{(\mathcal{I}_1\cap[1,\ell])}_{P_1}(\rho_{(\mathcal{I}_1\cap[1,\ell])}(?X_1,\ldots,?X_\ell)),$$

and

$$\text{query}^{\mathcal{I}_1}_{P_1}(\rho_{\mathcal{I}_1}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m)), \neg\text{compatible}^{(\mathcal{I}_1\cap[1,\ell])}_{P_1}(\rho_{(\mathcal{I}_1\cap[1,\ell])}(?X_1,\ldots,?X_\ell)) \rightarrow \text{query}^{\mathcal{I}_1}_{P_1}(\rho_{\mathcal{I}_1}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m)),$$

where $\mathcal{I} = (\mathcal{I}_1 \cap \mathcal{I}_2 \cap [1,\ell]) \cup (\mathcal{I}_1 \cap [\ell+1,\ell+m]) \cup \{m+k \mid k \in \mathcal{I}_2 \cap [\ell+1,\ell+n]\}$.

(4) If $P = (P_1 \text{ FILTER } R)$, then $\tau_{\text{opr}}(P)$ is defined as follows. First, let Σ_{eq} be a Datalog program consisting of the following three rules:

$$\text{triple}(?X,?Y,?Z) \rightarrow \text{eq}(?X,?X),$$
$$\text{triple}(?X,?Y,?Z) \rightarrow \text{eq}(?Y,?Y),$$
$$\text{triple}(?X,?Y,?Z) \rightarrow \text{eq}(?Z,?Z).$$

Now, assume that R is in DNF, that is,

$$R = \bigvee_{i=1}^{s} (L_{i,1} \wedge \cdots \wedge L_{i,k_i}),$$

where $L_{i,j}$ $(1 \leq i \leq s \text{ and } 1 \leq j \leq k_i)$ is an atomic built-in condition or the negation of an atomic built-in condition. For every $\mathcal{I} \subseteq [1, \ell+m]$, define $\Delta_{\mathcal{I}}$ as the set of variables $?U$ such that $?U$ is mentioned in the tuple $\rho_{\mathcal{I}}(?X_1,\ldots,?X_\ell,?Y_1,\ldots,?Y_m)$, and for every

$i \in \{1, \ldots, s\}$ and $j \in \{1, \ldots, k_i\}$, define $\tau_{\mathcal{I}}(L_{i,j})$ as follows:

$$\tau_{\mathcal{I}}(L_{i,j}) = \begin{cases} \text{eq}(?U, c) & \text{if } L_{i,j} = (?U = c) \text{ and } ?U \in \Delta_{\mathcal{I}}, \\ \mathit{false} & \text{if } L_{i,j} = (?U = c) \text{ and } ?U \notin \Delta_{\mathcal{I}}, \\ \text{eq}(?U, ?V) & \text{if } L_{i,j} = (?U = ?V), ?U \in \Delta_{\mathcal{I}} \text{ and } ?V \in \Delta_{\mathcal{I}}, \\ \mathit{false} & \text{if } L_{i,j} = (?U = ?V), \text{ and } ?U \notin \Delta_{\mathcal{I}} \text{ or } ?V \notin \Delta_{\mathcal{I}}, \\ \mathit{true} & \text{if } L_{i,j} = \text{bound}(?U) \text{ and } ?U \in \Delta_{\mathcal{I}}, \\ \mathit{false} & \text{if } L_{i,j} = \text{bound}(?U) \text{ and } ?U \notin \Delta_{\mathcal{I}}, \\ \neg\text{eq}(?U, c) & \text{if } L_{i,j} = \neg(?U = c) \text{ and } ?U \in \Delta_{\mathcal{I}}, \\ \mathit{true} & \text{if } L_{i,j} = \neg(?U = c) \text{ and } ?U \notin \Delta_{\mathcal{I}}, \\ \neg\text{eq}(?U, ?V) & \text{if } L_{i,j} = \neg(?U = ?V), ?U \in \Delta_{\mathcal{I}} \text{ and } ?V \in \Delta_{\mathcal{I}}, \\ \mathit{true} & \text{if } L_{i,j} = \neg(?U = ?V), \text{ and } ?U \notin \Delta_{\mathcal{I}} \text{ or } ?V \notin \Delta_{\mathcal{I}}, \\ \mathit{false} & \text{if } L_{i,j} = \neg\,\text{bound}(?U) \text{ and } ?U \in \Delta_{\mathcal{I}}, \\ \mathit{true} & \text{if } L_{i,j} = \neg\,\text{bound}(?U) \text{ and } ?U \notin \Delta_{\mathcal{I}}. \end{cases}$$

The program $\tau_{\text{opr}}(P)$ consists of the rules in $(\tau_{\text{opr}}(P_1) \cup \Sigma_{\text{eq}})$ together with the following set of rules for every $\mathcal{I}_1 \subseteq [1, \ell + m]$:

$$\Big\{ \text{query}_{P_1}^{\mathcal{I}_1}(\rho_{\mathcal{I}_1}(?X_1, \ldots, ?X_\ell, ?Y_1, \ldots, ?Y_m)), \tau_{\mathcal{I}_1}(L_{i,j_1}), \ldots, \tau_{\mathcal{I}_1}(L_{i,j_p}) \rightarrow \text{query}_P^{\mathcal{I}_1}(\rho_{\mathcal{I}_1}(?X_1, \ldots, ?X_\ell, ?Y_1, \ldots, ?Y_m)) \;\Big|\; 1 \le i \le s, \tau_{\mathcal{I}_1}(L_{i,q}) \neq \mathit{false} \text{ for every } q \in [1, k_i], 1 \le j_1 < \cdots < j_p \le k_i, \text{ and } \{j_1, \ldots, j_p\} = \{q \in [1, k_i] \mid \tau_{\mathcal{I}_1}(L_{i,q}) \neq \mathit{true}\} \Big\}.$$

(5) Finally, if $P = (\text{SELECT } W\ P_1)$, then $\tau_{\text{opr}}(P)$ consists of the rules in $\tau_{\text{opr}}(P_1)$ together with the following rule for every $\mathcal{I}_1 \subseteq [1, \ell + m]$:

$$\text{query}_{P_1}^{\mathcal{I}_1}(\rho_{\mathcal{I}_1}(?X_1, \ldots, ?X_\ell, ?Y_1, \ldots, ?Y_m)) \rightarrow \text{query}_P^{\mathcal{I}_1'}(\rho_{\mathcal{I}_1'}(?X_1, \ldots, ?X_\ell, ?Y_1, \ldots, ?Y_m)),$$

where $\mathcal{I}_1' = \mathcal{I}_1 \cup \{i \mid i \in [1, \ell] \text{ and } ?X_i \notin \text{fvar}(P_1) \cap W\} \cup \{\ell + j \mid j \in [1, m] \text{ and } ?Y_j \notin \text{fvar}(P_1) \cap W\}$.

A.3 The Program $\tau_{\text{out}}(P)$: Computing the Output Predicate

By construction, some atoms of the form query_P^J, where J is a set of indices, appear in the head of some rules of $\tau_{\text{opr}}(P)$. For example, if P_3 is the graph pattern in Example 5.1 (see Section 5.1), then $\text{query}_P(?X, ?Y, ?Z)$ and $\text{query}_P^{\{3\}}(?X, ?Y)$ occur in $\tau_{\text{opr}}(P_3)$; if $J = \varnothing$, then we simply write $\text{query}_P(?X_1, \ldots, ?X_k)$ instead of $\text{query}_P^{\varnothing}(?X_1, \ldots, ?X_k)$. Then for every atom $\text{query}_P^J(?X_1, \ldots, ?X_k)$ occurring in $\tau_{\text{opr}}(P)$, the following rule is added to $\tau_{\text{out}}(P)$:

$$\text{query}_P^J(?X_1, \ldots, ?X_k) \rightarrow \text{answer}_P(t_1, \ldots, t_{m_P}),$$

where m_P is the arity of the predicate query_P, t_i is the special constant $\star$ if $i \in J$, and after eliminating all the occurrences of $\star$ from $(t_1, \ldots, t_{m_P})$ the tuple $(?X_1, \ldots, ?X_k)$ is obtained. For example, due to the atom $\text{query}_{P_3}^{\{3\}}$ occurring in $\tau_{\text{opr}}(P_3)$, where P_3 is the graph pattern in Example 5.1,

$$\text{query}_{P_3}^{\{3\}}(?X, ?Y) \rightarrow \text{answer}_{P_3}(?X, ?Y, \star)$$

is added to $\tau_{\text{out}}(P_3)$.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for many helpful comments.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
[2] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. 2009. Extending SPARQL with regular expression patterns (for querying RDF). *Journal of Web Semantics* 7, 2 (2009), 57–73.
[3] Renzo Angles and Claudio Gutierrez. 2008. The expressive power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference*. 114–129.
[4] 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*. 14–26.
[5] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. 2009. Foundations of RDF databases. In *Reasoning Web*. 158–204.
[6] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On rules with existential variables: Walking the decidability line. *Artificial Intelligence* 175, 9–10 (2011), 1620–1654.
[7] Pablo Barceló. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 175–188.
[8] Catriel Beeri and Moshe Y. Vardi. 1981. The implication problem for data dependencies. In *Proceedings of the 8th International Colloquium on Automata, Languages and Programming*. 73–85.
[9] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2017. Swift logic for big data and knowledge graphs. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 2–10.
[10] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog system: Datalog-based reasoning for knowledge graphs. *Proceedings of the VLDB Endowment* 11, 9 (2018), 975–987.
[11] Gerald Berger, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2018. The space-efficient core of Vadalog. In *Proceedings of the 38th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. To Appear.
[12] Leopoldo E. Bertossi, Georg Gottlob, and Reinhard Pichler. 2018. Datalog: Bag semantics via set semantics. *CoRR* abs/1803.06445 (2018). arxiv:1803.06445 http://arxiv.org/abs/1803.06445.
[13] Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research* 48 (2013), 115–174.
[14] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. 2010. Datalog+/−: A family of logical knowledge representation and query languages for new applications. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*. 228–242.
[15] Andrea Calì, Georg Gottlob, and Andreas Pieris. 2012. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence* 193 (2012), 87–128.
[16] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2007. Tractable reasoning and efficient query answering in description logics: The DL-lite family. *Journal of Automated Reasoning* 39, 3 (2007), 385–429.
[17] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1990. *Logic Programming and Databases*. Springer.
[18] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. 2009. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering* 68, 10 (2009), 973–1000.
[19] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* 33, 3 (2001), 374–425.
[20] Valeria Fionda, Claudio Gutierrez, and Giuseppe Pirrò. 2012. Semantic navigation on the web of data: Specification of routes, web fragments and actions. In *Proceedings of the 21st World Wide Web Conference*. 281–290.
[21] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. 2006. RDF querying: Language constructs and evaluation methods compared. In *Reasoning Web*. 1–52.
[22] Birte Glimm and Chimezie Ogbuji. 2013. SPARQL 1.1 Entailment Regimes. Retrieved March 21, 2013 from, http://www.w3.org/TR/sparql11-entailment/.
[23] Georg Gottlob and Andreas Pieris. 2015. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*. 2999–3007.
[24] Georg Gottlob, Sebastian Rudolph, and Mantas Simkus. 2014. Expressiveness of guarded existential rule languages. In *Proceedings of the 33rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 27–38.
[25] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language. Retrieved March 21, 2013 from http://www.w3.org/TR/sparql11-query/.
[26] André Hernich, Clemens Kupke, Thomas Lukasiewicz, and Georg Gottlob. 2013. Well-founded semantics for extended datalog and ontological reasoning. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 225–236.

[27] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. 2014. Everything you always wanted to know about blank nodes. *Journal of Web Semenatics* 27 (2014), 42–69.
[28] Ilianna Kollia, Birte Glimm, and Ian Horrocks. 2011. SPARQL query answering over OWL ontologies. In *Proceedings of the 8th Extended Semantic Web Conference, Part I.* 382–396.
[29] Leonid Libkin, Juan L. Reutter, and Domagoj Vrgoc. 2013. Trial for RDF: Adapting graph query languages for RDF data. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 201–212.
[30] Deborah L. McGuinness and Frank van Harmelen. 2004. OWL Web Ontology Language Overview. Retrieved on February 10, 2004 from http://www.w3.org/TR/owl-features/.
[31] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. 2012. OWL 2 Web Ontology Language Profiles (2nd ed.). Retrieved December 11, 2012 from http://www.w3.org/TR/owl2-profiles/.
[32] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. 2012. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (2nd ed.). Retrieved December 11, 2012 from http://www.w3.org/TR/owl2-syntax/.
[33] Peter F. Patel-Schneider and Boris Motik. 2012. OWL 2 Web Ontology Language Mapping to RDF Graphs (2nd ed.). Retrieved 11 December 2012 from http://www.w3.org/TR/owl2-mapping-to-rdf/.
[34] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (2009), 16:1–16:45.
[35] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2010. nSPARQL: A navigational language for RDF. *Journal of Web Semantics* 8, 4 (2010), 255–270.
[36] Axel Polleres. 2007. From SPARQL to rules (and back). In *Proceedings of the 16th International Conference on World Wide Web*. 787–796.
[37] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF. Retrieved January 15, 2008 from http://www.w3.org/TR/rdf-sparql-query/.
[38] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc. 2015. Recursion in SPARQL. In *Proceedings of the 14th International Semantic Web Conference*. 19–35.
[39] Sebastian Rudolph and Markus Krötzsch. 2013. Flag & check: Data access with monadically defined queries. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 151–162.
[40] Simon Schenk. 2007. A SPARQL semantics based on Datalog. In *Proceedings of the 30th Annual German Conference on Artificial Intelligence*. 160–174.
[41] W3C OWL Working Group. 2012. OWL 2 Web Ontology Language Document Overview (2nd ed.). Retrieved December 11, 2012 from http://www.w3.org/TR/owl2-overview/.

Received April 2017; revised May 2018; accepted July 2018