Universidad Rey Juan Carlos

SPARQL RULES!

Axel Polleres

GIA TECHNICAL REPORT 2006-11-28
NOVEMBER 2006

Universidad Rey Juan Carlos
Tulipan s/n
28933, Móstoles, Madrid
ESPAÑA
http://www.ia.escet.urjc.es/

GIA TECHNICAL REPORT
GIA TECHNICAL REPORT 2006-11-28, NOVEMBER 2006

SPARQL RULES!

Axel Polleres
Universidad Rey Juan Carlos, Madrid, Spain

Abstract. As the data and ontology layers of the Semantic Web stack have achieved a certain level of maturity in standard recommendations such as RDF and OWL, the current focus lies on two related aspects. On the one hand, the definition of a suitable query language for RDF, SPARQL, seems to be close to candidate recommendation status within the W3C. The establishment of the rules layer on top of the existing stack on the other hand marks the next step to be taken, where especially languages with their roots in Logic Programming and Deductive Databases are receiving considerable attention. The purpose of this paper is threefold. First, we discuss the formal semantics of SPARQL extending recent results in several ways. Second, we provide translations from SPARQL to Datalog with stratified negation as failure. Third, we propose some useful and easy to implement extensions of SPARQL, based on this translation. As it turns out, the combination serves for direct implementations of SPARQL on top of existing rules engines as well as a basis for more general rules and query languages on top of RDF. A prototype implementation is available for evaluation of our approach.

[*]This report is an extended version of the paper titled "From SPARQL to Rules (and back)" to appear in WWW2007. It is a slight update of the original report published at `http://www.polleres.net/publications/GIA-TR-2006-11-28.pdf` on 2006-11-28 with minor typo fixes and encompasses reviewer comments for the conference version.

1 Introduction

After the data and ontology layers of the Semantic Web stack have achieved a certain level of maturity in standard recommendations such as RDF and OWL, the query and the rules layers seem to be the next building-blocks to be finalized. For the first part, SPARQL [21], W3C's proposed query language seems to be close to recommendation, though the Data Access working group is still struggling with defining aspects such as a formal semantics or layering on top of OWL and RDFS. As for the second part, the RIF working group [25], who is responsible for the rules layer, is just producing first concrete results. Besides aspects like business rules exchange or reactive rules, deductive rules languages on top of RDF and OWL are of special interest to the RIF group. One such deductive rules language is Datalog, which has been successfully applied in areas such as deductive databases and thus might be viewed as a query language itself. Let us briefly recap our starting points:

Datalog and SQL Analogies between Datalog and RDBMS query languages such as SQL are well-known and -studied. Both formalisms cover UCQ (unions of conjunctive, or select-project-join queries). Both can express set difference, by the keyword MINUS in SQL and using nonmonotonic negation (aka negation as failure) in a slightly generalized form of pure Datalog.

Datalog adds recursion, particularly unrestricted recursion involving nonmonotonic negation (aka unstratified negation as failure).

Still, SQL is often viewed to be more powerful in several respects: On the one hand, the lack of recursion has been partly solved in the SQL standard's 1999 version [23] with the introduction of recursive views which may involve stratified negation. On the other hand, aggregates or external function calls are missing in pure Datalog.

However, also developments on the Datalog side are evolving and with recent extensions of Datalog towards Answer Set Programming (ASP) – a logic programming paradigm extending and building on top of Datalog – lots of these issues have been solved, for instance by defining a declarative semantics for aggregates [11] or external predicates [10, 4].

The Semantic Web rules layer Remarkably, logic programming dialects such as Datalog with nonmonotonic negation which are covered by Answer Set Programming are often viewed as a natural basis for the Semantic Web rules layer [9]. Current ASP systems offer extensions for retrieving RDF data and querying OWL knowledge bases from the Web [10]. Particular concerns in the Semantic Web community exist with respect to adding rules including nonmonotonic negation [3] which involve a form of closed world reasoning on top of RDF and OWL which both adopt an open world assumption. Recent proposals for solving this issue suggest a "safe" use of negation as failure over finite contexts only for the Web which coined the term *scoped negation* [25, 20].

The Semantic Web query layer – SPARQL Since we base our considerations in this paper on the assumption that similar correspondences as between SQL and Datalog can be established for SPARQL, we have to observe that SPARQL inherits a lot from SQL, but there also remain substantial differences. On the one hand, SPARQL does not deal with nested queries or recursion, a detail which is indeed surprising by the fact that SPARQL is a graph query language on RDF where, typical recursive queries such as transitive closure of a property might seem very useful. Likewise, aggregation (such as count, average, etc.) of object values in RDF triples which might appear useful have not yet been included in the current standard. On the other hand, subtleties like blank nodes (aka bNodes), need to be taken into account, optional graph patterns, which have their counterpart (to some extent only, as we will see) in outer joins in SQL or relational algebra, are not straightforwardly translatable to Datalog, etc.

The goal of this paper is to shed light on the actual relation between declarative rules languages such as Datalog and SPARQL, and by this also provide valuable input for the currently ongoing discussions on the Semantic Web rules layer, in particular its integration with SPARQL, taking the likely direction into account that LP style rules languages will play a significant role in this context.

Although the SPARQL specification does not seem 100% stable at the current point, just having taken a step back from candidate recommendation to working draft, we do not think that it is too early for this exercise, since, as we will see, we gain valuable insights and positive side effects by our investigation. More precisely, the contributions of the present work are:

- We refine and extend a recent proposal to formalize the semantics of SPARQL from Pérez et al. [18], presenting three variants, namely c-joining, s-joining and b-joining semantics where the latter coincides with [18], and can thus be considered normative. We further discuss how aspects such compositionality, or idempotency of joins are treated in these semantics.
- Based on the three semantic variants, we provide translations from a large fragment of SPARQL queries to Datalog, which give rise to implementations of SPARQL on top of existing engines.
- We provide some straightforward extensions of SPARQL such as a set difference operator MINUS, and nesting of ASK queries in FILTER expressions.
- Finally, we discuss an extension towards recursion by allowing bNode-free-CONSTRUCT queries as part of the query dataset, which may be viewed as a light-weight, recursive rule language on top of of RDF.

The remainder of this paper is structured as follows: In Sec. 2 we first overview SPARQL, discuss some issues in the language (Sec. 2.1) and then define its formal semantics (Sec. 2.2). After introducing a general form of Datalog with negation as failure under the answer set semantics in Sec. 3, we proceed with the translations of SPARQL to Datalog in Sec. 4. We discuss our proposed language extensions and the use SPARQL as a rules language itself in Sec. 5. We point to a prototype implementation available for evaluation of our appraach in Sec. 6 before we conclude in Sec. 7. An appendix contains translated logic programs for all sample queries mentioned throughout the paper in order to exemplify the translation.

2 RDF and SPARQL

In examples, we will subsequently refer to the two RDF graphs in Fig. 1 which give some information about Bob and $Alice$. Such information about persons is common in so-called FOAF[1] files which are gaining popularity to describe personal data on the Web. Similarities with existing examples in [21] are on purpose. We assume the two RDF graphs given in TURTLE [2] notation and accessible via the IRIs `ex.org/bob` and `alice.org`[2]

```
# Graph: ex.org/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix bob: <ex.org/bob#> .

  <ex.org/bob> foaf:maker _:a.
  _:a a foaf:Person ; foaf:name "Bob";
           foaf:knows _:b.

  _:b a foaf:Person ; foaf:nick "Alice".
  <alice.org/> foaf:maker _:b
```

```
# Graph: alice.org

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <alice.org#> .

  alice:me a foaf:Person ; foaf:name "Alice" ;
           foaf:knows _:c.

  _:c  a foaf:Person ; foaf:name "Bob" ;
              foaf:nick "Bobby".
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT?Y ?X
FROM <alice.org>
FROM <ex.org/bob>
WHERE { ?Y foaf:name ?X .}
```

?X	?Y
"Bob"	_:a
"Bob"	_:c
"Alice"	alice.org#me

Figure 1: Two RDF graphs in TURTLE notation and a simple SPARQL query.

We assume the pairwise disjoint, infinite sets I, B, L and Var, which denote IRIs, Blank nodes, RDF literals, and variables respectively. In this paper, an *RDF Graph* is then a finite set, of triples from $I \cup B \cup L \times I \times I \cup B \cup L$,[3]

[1] see `http://www.foaf-project.org/`

[2] For reasons of legibility and conciseness, we normally omit the leading 'http://' or other schema identifiers in IRIs in this paper.

[3] Following SPARQL, we are slightly more general than the original RDF specification in that we allow literals in subject positions.

dereferenceable by an IRI. A SPARQL *query* is a quadruple $Q = (V, P, DS, SM)$, where V is a result form, P is a graph pattern, DS is a dataset, and SM is a set of solution modifiers. We refer to [21] for syntactical details and will explain these in the following as far as necessary. In this paper, we will ignore solution modifiers mostly, thus we will usually write queries as triples $Q = (V, P, DS)$, and will use the syntax for graph patterns introduced below.

Result Forms Since we will, to a large extent, restrict ourselves to SELECT queries, it is sufficient for our purposes to describe result forms by sets variables. Other result forms will be discussed in Sec. 5. For instance, let $Q = (V, P, DS)$ denote the query from Fig. 1, then $V = \{?X, ?Y\}$. Query results in SPARQL are given by partial, i.e. possibly incomplete, substitutions of variables in V by RDF terms. In traditional relational query languages, such incompleteness is usually expressed using null values. Using such null values we will write solutions as tuples where the order of columns is determined by *lexicographically ordering* the variables in V. Given a set of variables V, let $\overline{V}$ denote the tuple obtained from lexicographically ordering V.

The query from Fig. 1 with result form $\overline{V} = (?X, ?Y)$ then has solution tuples ("Bob", _:a), ("Alice", alice.org#me), ("Bob", _:c). We write substitutions in sqare brackets, so these tuples correspond to the substitutions [$?X \rightarrow$ "Bob", $?Y \rightarrow$ _:a], [$?X \rightarrow$ "Alice", $?Y \rightarrow$ alice.org#me], and [$?X \rightarrow$ "Bob", $?Y \rightarrow$ _:c], respectively.

Graph Patterns We follow the recursive definition of graph patterns P from [18]:

- a tuple (s, p, o) is a graph pattern where $s, o \in I \cup L \cup Var$ and $p \in I \cup Var$.[4]
- if P and P' are graph patterns then (P AND P'), (P OPT P'), (P UNION P'), (P MINUS P') are graph patterns.[5]
- if P is a graph pattern and $i \in I \cup Var$, then (GRAPH i P) is a graph pattern.
- if P is a graph pattern and R is a filter expression then (P FILTER R) is a graph pattern.

For any pattern P, we denote by $vars(P)$ the set of all variables occurring in P.

As *atomic filter expression*, SPARQL allows the unary predicates BOUND, isBLANK, isIRI, isLITERAL, binary equality predicates '$=$' for literals, and other features such as comparison operators, data type conversion and string functions which we omit here, see [21, Sec. 11.3] for details. *Complex filter expressions* can be built using the connectives '$\neg$','$\wedge$','$\vee$' and auxiliary parentheses.

Datasets The dataset $DS = (G, \{(g_1, G_1), \ldots (g_k, G_k)\})$ of a SPARQL query is defined by a default graph G plus a set of named graphs, i.e. pairs of IRIs and corresponding graphs. Without loss of generality (there are other ways to define the dataset such as in a SPARQL protocol query), we assume G given as the merge of the graphs denoted by the IRIs given in a set of FROM and FROM NAMED clauses. For instance, the query from Fig. 1 refers to the dataset which consists of the default graph obtained from merging `alice.org` $\uplus$ `ex.org/bob` plus an empty set of named graphs.

The relation between names and graphs in SPARQL is defined solely in terms of that the IRI defines a resource which is represented by the respective graph. In this paper, we assume that the IRIs represent indeed network-accessible resources where the respective RDF-graphs can be retrieved from. This view has also be taken e.g. in [20]. Particularly, this treatment is not to be confused with so-called named graphs in the sense of [5]. We thus identify each IRI with the RDF graph available at this IRI and each set G of IRIs with the graph merge [15] over the graphs accessible at the respective IRIs in this set G. This allows us to identify the dataset by a pair of sets of IRIs $DS = (G, G_n)$ with $G = \{d_1, \ldots, d_n\}$ and $G_n = \{g_1, \ldots, g_k\}$ denoting the (merged) default graph and the set of named graphs, respectively. Hence, the following set of clauses

[4]We do not consider bNodes in patterns as these can be semantically equivalently replaced by variables in graph patterns [8].

[5]Note that AND and MINUS are not designated keywords in SPARQL, but we use them here for reasons of readability and in order to keep with the operator style definition of [18]. MINUS is syntactically not present at all, but we will suggest a syntax extension for this particular keyword in Sec. 5.

```
FROM <ex.org/bob>
FROM NAMED <alice.org>
```

defines the dataset $DS = (\{\texttt{ex.org/bob}\}, \{\texttt{alice.org}\})$.

2.1 Assumptions and Issues

In this section we will discuss some important issues about the current specification, and how we will deal with them here.

Assumptions and Issues about the Dataset. Note that the default graph if specified by name in a FROM clause is not counted among the named graphs automatically [21, section 8, definition 1]. An unbound variable in the GRAPH directive, means any of the named graphs in DS, but does NOT necessarily include the default graph.

Example 2.1 *This issue becomes obvious in the following query with dataset* $DS = (\{\texttt{ex.org/bob}\}, \emptyset)$ *which has an empty solution set.*

```
SELECT ?N WHERE {?G foaf:maker ?M .
            GRAPH ?G { ?X foaf:name ?N } }
```

We will sometimes find the following assumption convenient to avoid such arguably unintuitive effects:

Definition 2.1 *(*Dataset closedness assumption*) Given a dataset* $DS = (G, G_n)$, G_n *implicitly contains (i) all graphs mentioned in* G *and (ii) all IRIs mentioned explicitly in the graphs corresponding to* G.

Under this assumption, the previous query has both $("Alice")$ and $("Bob")$ in its solution set.

Assumptions and Issues about Filter expressions. Some more remarks are in place concerning FILTER expressions. According to the SPARQL specification *"Graph pattern matching creates bindings of variables [where] it is possible to further restrict solutions by constraining the allowable bindings of variables to RDF Terms [with FILTER expressions]."* However, it is not clearly specified how to deal with filter constraints referring to variables which do not appear in simple graph patterns. In this paper, for graph patterns of the form $(P$ FILTER $R)$ we tacitly assume *safe filter expressions*, i.e. that all variables used in a filter expression R also appear in the corresponding pattern P. This corresponds with the notion of safety in Datalog (see Sec.3), where the built-in predicates (which obviously correspond to filter predicates) do not suffice to safe unbound variables.

Moreover, the specification defines errors to avoid mistyped comparisons, or evaluation of built-in functions over unbound values, i.e. "any potential solution that causes an error condition in a constraint will not form part of the final results, but does not cause the query to fail." These errors propagate over the whole FILTER expression, also over negation, as shown by the following example.

Example 2.2 *Assuming the dataset does not contain triples for the* foaf : dummy *property, the example query*

```
SELECT ?X
WHERE { {?X a foaf:Person .
           OPTIONAL { ?X foaf:dummy ?Y . } }
         FILTER ( ¬(isLITERAL (?Y)) ) }
```

would discard any solution for ?X*, since the unbound value for* ?Y *causes an error in the isLITERAL expression and thus the whole FILTER expression returns an error.*

We will take special care for these errors, when defining the semantics of FILTER expressions later on.

Assumptions about bNodes. We do not consider bNodes in query patterns as these can be semantically equivalently replaced by variables in graph patterns [8]. However, bNodes may appear in the Dataset, and thus in query answers. Note that different graphs in the dataset might use the same identifiers for bNodes. For simplicity, we assume that such ambiguities are already resolved, i.e. that different graphs in the dataset use different bNode identifiers. By this assumption, we may treat bNode identifiers largely like normal constants and we do not need to rename bNodes identifiers in query results.

2.2 Formal Semantics of SPARQL

The semantics of SPARQL is still not formally defined in its current version. This lack of formal semantics has been tackled by a recent proposal of Pérez et al. [18]. We will base on this proposal, but suggest three variants thereof, namely (a) *bravely joining* (b-joining), (b) *cautiously-joining* (c-joining), and (c) *strictly-joining* (s-joining) semantics. Particularly, our definitions vary from [18] in the way we define joining unbound variables (represented by the distinct constant null in our approach). Moreover, we will refine their notion of FILTER satisfaction in order to deal with error propagation properly.

We denote by T_{null} the union $I \cup B \cup L \cup \{\mathsf{null}\}$, where null is a dedicated constant denoting the unknown value not appearing in any of I, B, or L, how it is commonly introduced when defining outer joins in relational algebra.

A *substitution* θ from Var to T_{null} is a partial function $\theta : Var \to T_{\mathsf{null}}$. We write substitutions in postfix notation: For a triple pattern $t = (s, p, o)$ we denote by $t\theta$ the triple $(s\theta, p\theta, o\theta)$ obtained by applying the substitution to all variables in t. The *domain* of θ, denoted by $dom(\theta)$, is the subset of Var where θ is defined. For a substitution θ and a set of variables $D \subseteq Var$ we define the substitution θ^D with domain D as follows:

$$x\theta^D = \begin{cases} x\theta \text{ if } x \in dom(\theta) \cap D \\ \mathsf{null} \text{ if } x \in D \setminus dom(\theta) \end{cases}$$

Let θ_1 and θ_2 be substitutions, then $\theta_1 \cup \theta_2$ is the substitution obtained as follows:

$$x(\theta_1 \cup \theta_2) = \begin{cases} x\theta_1 \text{ if } x\theta_1 \text{ defined and } x\theta_2 \text{ undefined} \\ \text{else: } x\theta_1 \text{ if } x\theta_1 \text{ defined and } x\theta_2 = \mathsf{null} \\ \text{else: } x\theta_2 \text{ if } x\theta_2 \text{ defined} \\ \text{else: undefined} \end{cases}$$

Thus, in the union of two substitutions defined values in one take precedence over null values the other substitution. For instance, given the substitutions $\theta_1 = [?X \to "Alice", ?Y \to \_{:}\mathsf{a}, ?Z \to \mathsf{null}]$ and $\theta_2 = [?U \to "Bob", ?X \to "Alice", ?Y \to \mathsf{null}]$ we get: $\theta_1 \cup \theta_2 = [?U \to "Bob", ?X \to "Alice", ?Y \to \_{:}\mathsf{a}, ?Z \to \mathsf{null}]$

Now, as opposed to [18], we define three notions of compatibility between substitutions:

- Two substitutions θ_1 and θ_2 are bravely compatible (*b-compatible*) when for all $x \in dom(\theta_1) \cap dom(\theta_2)$ either $x\theta_1 = \mathsf{null}$ or $x\theta_2 = \mathsf{null}$ or $x\theta_1 = x\theta_2$ holds. i.e., when $\theta_1 \cup \theta_2$ is a substitution over $dom(\theta_1) \cup dom(\theta_2)$.
- Two substitutions θ_1 and θ_2 are cautiously compatible (*c-compatible*) when they are b-compatible and for all $x \in dom(\theta_1) \cap dom(\theta_2)$ it holds that $x\theta_1 = x\theta_2$.
- Two substitutions θ_1 and θ_2 are strictly compatible (*s-compatible*) when they are c-compatible and for all $x \in dom(\theta_1) \cap dom(\theta_2)$ it holds that $x(\theta_1 \cup \theta_2) \neq \mathsf{null}$.

Analogously to [18] we define join, union, difference, and outer join between two sets of substitutions Ω_1 and Ω_2 over domains D_1 and D_2, respectively, all except union parameterized by $x \in \{$b,c,s$\}$:

$$\begin{aligned}
\Omega_1 \bowtie_x \Omega_2 &= \{\theta_1 \cup \theta_2 \mid \theta_1 \in \Omega_1, \theta_2 \in \Omega_2, \text{ are } x\text{-compatible}\} \\
\Omega_1 \cup \Omega_2 &= \{\theta \mid \exists \theta_1 \in \Omega_1 \text{ with } \theta = \theta_1^{D_1 \cup D_2} \text{ or} \\
&\qquad \exists \theta_2 \in \Omega_2 \text{ with } \theta = \theta_2^{D_1 \cup D_2}\} \\
\Omega_1 -_x \Omega_2 &= \{\theta \in \Omega_1 \mid \text{for all } \theta_2 \in \Omega_2,\ \theta \text{ and } \theta_2 \text{ not } x\text{-compatible}\} \\
\Omega_1 ⟕_x \Omega_2 &= (\Omega_1 \bowtie_x \Omega_2) \cup (\Omega_1 -_x \Omega_2)
\end{aligned}$$

The semantics of a graph pattern P over dataset $DS = (G, G_n)$, can now be defined recursively by the evaluation function returning sets of substitutions.

Definition 2.2 *(*Evaluation, extends [18, Def. 2]*) Let $t = (s, p, o)$ be a triple pattern, P, P_1, P_2 graph patterns, and $DS = (G, G_n)$ a dataset, then the x-joining evaluation $[[\cdot]]^x_{DS}$ is defined as follows:*

$$
\begin{array}{ll}
[[t]]^x_{DS} & = \{\theta \mid dom(\theta) = vars(P) \text{ and } t\theta \in G\} \\
[[P_1 \textsf{ AND } P_2]]^x_{DS} & = [[P_1]]^x_{DS} \bowtie_x [[P_2]]^x_{DS} \\
[[P_1 \textsf{ UNION } P_2]]^x_{DS} & = [[P_1]]^x_{DS} \cup [[P_2]]^x_{DS} \\
[[P_1 \textsf{ MINUS } P_2]]^x_{DS} & = [[P_1]]^x_{DS} -_x [[P_2]]^x_{DS} \\
[[P_1 \textsf{ OPT } P_2]]^x_{DS} & = [[P_1]]^x_{DS} ⟕_x [[P_2]]^x_{DS} \\
[[\textsf{GRAPH } i\ P]]^x_{DS} & = [[P]]^x_{(i,\emptyset)}, \text{ for } i \in G_n \\
[[\textsf{GRAPH } v\ P]]^x_{DS} & = \{\theta \cup [v \to g] \mid g \in G_n \text{ and } \theta \in [[\ P[v \to g]\ ]]^x_{(g,\emptyset)}\}, \text{ for } v \in Var \\
[[P \textsf{ FILTER } R]]^x_{DS} & = \{\theta \in [[P]]^x_{DS} \mid R\theta = \top\}
\end{array}
$$

Let R be a **FILTER** *expression, $u, v \in Var$, $c \in I \cup B \cup L$. The valuation of R on a substitution θ, written $R\theta$ takes one of the three values $\{\top, \bot, \varepsilon\}$[6] and is defined as follows.*
$R\theta = \top$, if:

(1) $R = \textsf{BOUND}(v)$ with $v \in dom(\theta) \ \wedge\ v\theta \neq \textsf{null}$;
(2) $R = \textsf{isBLANK}(v)$ with $v \in dom(\theta) \ \wedge\ v\theta \in B$;
(3) $R = \textsf{isIRI}(v)$ with $v \in dom(\theta) \ \wedge\ v\theta \in I$;
(4) $R = \textsf{isLITERAL}(v)$ with $v \in dom(\theta) \ \wedge\ v\theta \in L$;
(5) $R = (v = c)$ with $v \in dom(\theta) \ \wedge\ v\theta = c$;
(6) $R = (u = v)$ with $u, v \in dom(\theta) \ \wedge\ u\theta = v\theta \ \wedge\ u\theta \neq \textsf{null}$;
(7) $R = (\neg R_1)$ with $R_1\theta = \bot$;
(8) $R = (R1 \vee R2)$ with $R_1\theta = \top \ \vee\ R_2\theta = \top$;
(9) $R = (R1 \wedge R2)$ with $R_1\theta = \top \ \wedge\ R_2\theta = \top$.

$R\theta = \varepsilon$, if:

(1) $R = \textsf{isBLANK}(v)$, $R = \textsf{isIRI}(v)$, $R = \textsf{isLITERAL}(v)$,
or $R = (v = c)$ with $v \notin dom(\theta) \ \vee\ v\theta = \textsf{null}$;
(2) $R = (u = v)$ with $u \notin dom(\theta) \ \vee\ u\theta = \textsf{null}$
$\vee\ v \notin dom(\theta) \ \vee\ v\theta = \textsf{null}$;
(3) $R = (\neg R_1)$ and $R_1\theta = \varepsilon$;
(4) $R = (R_1 \vee R_2)$ and $(R_1\theta \neq \top \ \wedge\ R_2\theta \neq \top) \ \wedge$
$(R_1\theta = \varepsilon \ \vee\ R_2\theta = \varepsilon)$;
(5) $R = (R1 \wedge R2)$ and $R_1\theta = \varepsilon \ \vee\ R_2\theta = \varepsilon$.

$R\theta = \bot$ otherwise.

We will now exemplify the three different semantics defined above, namely bravely joining (b-joining), cautiously joining (c-joining), and strictly-joining (s-joining) semantics. When taking a closer look to the **AND** and **MINUS** operators, one will realize that all three semantics take a slightly differing view only when joining null. Indeed, the **AND** operator behaves as the traditional natural join operator $\bowtie$ in relational algebra, when no null values are involved.

Example 2.3 *Take for instance, $DS = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$ and*

$P = ((?X, \texttt{name}, ?Name) \textsf{ AND } (?X, \texttt{knows}, ?Friend))$.

When viewing each solution set as a relational table with variables denoting attribute names, we can write:

[6]$\top$ stands for "true", $\bot$ stands for "false" and ε stands for errors, see [21, Sec. 11.3] and Example 2.2 for details.

?X	?Name
`_:a`	"Bob"
alice.org#me	"Alice"
`_:c`	"Bob"

$\bowtie$

?X	?Friend
`_:a`	`_:b`
alice.org#me	`_:c`

$=$

?X	?Name	?Friend
`_:a`	"Bob"	`_:b`
alice.org#me	"Alice"	`_:c`

Differences between the three semantics appear when joining over null-bound variables, as shown in the next example.

Example 2.4 *Let DS be as in Example 2.3 and assume the following query which might be considered a naive attempt to ask for pairs of persons $?X1$, $?X2$ who share the same name and nickname where both, name and nickname are optional:*

$$P = (\ ((?X1, a, \texttt{Person})\ \textit{OPT}\ (?X1, \texttt{name}, ?N))\ \textit{AND}$$
$$((?X2, a, \texttt{Person})\ \textit{OPT}\ (?X2, \texttt{nick}, ?N))\ )$$

Again, we consider the tabular view of the resulting join:

?X1	?N
`_:a`	"Bob"
`_:b`	null
`_:c`	"Bob"
alice.org#me	"Alice"

$\bowtie_x$

?X2	?N
`_:a`	null
`_:b`	"Alice"
`_:c`	"Bobby"
alice.org#me	null

Now, let us see what happens when we evaluate the join $\bowtie_x$ with respect to the different semantics. The following result table lists in the last column which tuples belong to the result of b-, c- and s-join, respectively.

$=$

?X1	?N	X2	
`_:a`	"Bob"	`_:a`	b
`_:a`	"Bob"	alice.org#me	b
`_:b`	null	`_:a`	b,c
`_:b`	"Alice"	`_:b`	b
`_:b`	"Bobby"	`_:c`	b
`_:b`	null	alice.org#me	b,c
`_:c`	"Bob"	`_:a`	b
`_:c`	"Bob"	alice.org#me	b
alice.org#me	"Alice"	`_:a`	b
alice.org#me	"Alice"	`_:b`	b,c,s
alice.org#me	"Alice"	alice.org#me	b

Leaving aside the question whether the query formulation was intuitively broken, we remark that only the s-join would have the expected result. At the very least we might argue, that the liberal behavior of b-joins might be considered surprising in some cases. The c-joining semantics acts a bit more cautious in between the two, treating null *values as normal values, only unifiable with other* null *values.*

From the point of view how joins over incomplete relations are treated in common relational database systems, the s-joining semantics might be considered the intuitive behavior of the join operator above. Another interesting divergence (which would rather suggest to adopt the c-joining semantics) shows up when we consider a simple idempotent join.

Example 2.5 *Let us consider the following single triple dataset $DS = (\{(\texttt{alice.org\#me}, a, \texttt{Person})\}, \emptyset)$ and the following simple query pattern:*

$$P = ((?X, a, \texttt{Person})\ \textit{UNION}\ (?Y, a, \texttt{Person}))$$

Clearly, this pattern, has the solution set

$$[[P]]^x_{DS} = \{(\texttt{alice.org\#me}, \textsf{null}), (\textsf{null}, \texttt{alice.org\#me})\}$$

under all three semantics. Somewhat surprisingly however, $P' = (P\ \textit{AND}\ P)$ has different solution sets for the different semantics. First, $[[P']]^c_{DS} = [[P]]^x_{DS}$, but $[[P']]^s_{DS} = \emptyset$, since null *values are not compatible under the s-joining semantics. Finally,*

$[[P']]^b_{DS} = \{(\texttt{alice.org\#me}, \mathsf{null}), (\mathsf{null}, \texttt{alice.org\#me}),$
$(\texttt{alice.org\#me}, \texttt{alice.org\#me})\}$

As shown by this example, under the reasonable assumption, that the join operator is idempotent, i.e., $(P \bowtie P) \equiv P$, only the c-joining semantics behaves correctly.

However, the brave b-joining behavior is advocated by the current SPARQL document, and we might also think of examples where this obviously makes a lot of sense. Especially, when considering no explicit joins, but the implicit joins within the OPT operator:

Example 2.6 *This example is a slight variant of a query from [6]. Let* $DS = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$ *and assume the following query for all persons and some name for these persons, where preferably the* $\texttt{foaf : name}$ *is taken, and, if not specified, the* $\texttt{foaf : nick}$.

$P = ((((?X, a, \texttt{Person})\ \mathit{OPT}\ (?X, \texttt{name}, ?XNAME))$
$\mathit{OPT}\ (?X, \texttt{nick}, ?XNAME))$

Only $[[P]]^b_{DS}$ *contains the expected solution* $(\_:\texttt{b}, "Alice")$ *for the bNode* $\_:\texttt{b}$.

All three semantics may be considered as variations of the original definitions in [18], for which the authors proved complexity results and various desirable features, such as semantics-preserving normal form transformations and compositionality. The following proposition shows that all these results carry over to the normative b-joining semantics:

Proposition 2.1 *Given a dataset* DS *and a pattern* P *which does not contain GRAPH patterns, the solutions of* $[[P]]_{DS}$ *as defined in [18] and* $[[P]]^b_{DS}$ *are in one-to-one correspondence.*

Proof: Given DS and P each substitution θ obtained by evaluation $[[P]]^b_{DS}$ can be reduced to a substitution θ' obtained from the evaluation $[[P]]_{DS}$ in [18] by dropping all mappings of the form $v \rightarrow \mathsf{null}$ from θ. Likewise, each substitution θ' obtained from $[[P]]_{DS}$ can be extended to a substitution $\theta = \theta'^{vars(P)}$ for $[[P]]^b_{DS}$. □

Following the definitions from the SPARQL specification, in fact, the b-joining semantics is the only admissible definition, which is why [18] does not consider null values at all. There are still advantages for gradually defining alternatives towards traditional treatment of joins involving nulls. On the one hand, as we have seen in the examples above, the brave view on joining unbound variables might have partly surprising results, on the other hand, as we will see, the c- and s-joining semantics allow for a more efficient implementation in terms of Datalog rules.

Let us now take a closer look on some properties of the three defined semantics.

Compositionality and Equivalences

As shown in [18], some implementations have a non-compositional semantics, leading to undesired effects such as non-commutativity of the join operator, etc. A semantics is called *compositional* if for each P' sub-pattern of P the result of evaluating P' can be used to evaluate P. Obviously, all three the c-, s- and b-joining semantics defined here retain this property, since all three semantics are defined recursively, and independent of the evaluation order of the sub-patterns.

The following proposition summarizes equivalences which hold for all three semantics, showing some interesting additions to the results of Pérez et al.

Proposition 2.2 (extends [18, Prop. 1]) *The following equivalences hold or do not hold in the different semantics as indicated after each law:*

(1) AND, UNION are associative and commutative. (b,c,s)
(2) $(P_1\ \mathit{AND}\ (P_2\ \mathit{UNION}\ P_3))$
$\equiv ((P_1\ \mathit{AND}\ P_2)\ \mathit{UNION}\ (P_1\ \mathit{AND}\ P_3)).$ *(b)*

(3) $(P_1 \textsf{\textit{OPT}}\, (P_2 \textsf{\textit{UNION}}\, P_3))$
$\equiv ((P_1 \textsf{\textit{OPT}}\, P_2) \textsf{\textit{UNION}}\, (P_1 \textsf{\textit{OPT}}\, P_3)).$ *(b)*
(4) $((P_1 \textsf{\textit{UNION}}\, P_2) \textsf{\textit{OPT}}\, P_3)$
$\equiv ((P_1 \textsf{\textit{OPT}}\, P_3) \textsf{\textit{UNION}}\, (P_2 \textsf{\textit{OPT}}\, P_3)).$ *(b)*
(5) $((P_1 \textsf{\textit{UNION}}\, P_2) \textsf{\textit{FILTER}}\, R)$
$\equiv ((P_1 \textsf{\textit{FILTER}}\, R) \textsf{\textit{UNION}}\, (P_2 \textsf{\textit{FILTER}}\, R)).$ *(b,c,s)*
(6) AND is idempotent, i.e. $(P \textsf{\textit{AND}}\, P) \equiv P$. *(c)*

Proof: [Sketch.] (1-5) for the b-joining semantics are proven in [18], so we only consider the other semantics in more detail.

(1): for c-joining and s-joining follows straight from the definitions.

(2)-(4): the substitution sets $[[P_1]]^{c,s} = \{[?X \to a, ?Y \to b]\}$, $[[P_2]]^{c,s} = \{[?X \to a, ?Z \to c]\}$, $[[P_3]]^{c,s} = \{[?Y \to b, ?Z \to c]\}$ provide counterexamples for c-joining and s-joining semantics for all three equivalences (2)-(4).

(5): The semantics of FILTER expressions and UNION is exactly the same for all three semantics. Thus, the result for the b-joining semantics carries over to all three semantics.[7]

(6): follows from the observations in Example 2.5. □

Ideally, we would like to identify a subclass of programs, where the three semantics coincide. Obviously, this is the case for any query involving neither UNION nor OPT operators. Pérez et al. [18] define a bigger class of programs, including "well-behaving" optional patterns:

Definition 2.3 *(*[18, Def. 4]*) A UNION-free graph pattern P is* well-designed *if for every occurrence of a sub-pattern $P' = (P_1 \textsf{\textit{OPT}}\, P_2)$ of P and for every variable v occurring in P, the following condition holds: if v occurs both in P_2 and outside P' then it also occurs in P_1.*

As may be easily verified by the reader, neither Example 2.4 nor Example 2.6, which are both UNION-free, satisfy the well-designedness condition. Since in the general case the equivalences for Prop. 2.2 do not hold, we also need to consider nested UNION patterns as a potential source for null bindings which might affect join results. We extend the notion of well-designedness, which direclty leads us to another correspondence in the subsequent proposition.

Definition 2.4 *A graph pattern P is* well-designed *if the condition from Def. 2.3 holds and additionally for every occurrence of a sub-pattern $P' = (P_1 \textsf{\textit{UNION}}\, P_2)$ of P and for every variable v occurring in P', the following condition holds: if v occurs outside P' then it occurs in both P_1 and P_2.*

Proposition 2.3 *On well-designed graph patterns the c-, s-, and b-joining semantics coincide.*

Proof: [Sketch.] Follows directly from the observation that all variables which are re-used outside P' must be bound to a value unequal to null in P' due to the well-designedness condition, and thus cannot generate null bindings which might carry over to joins. □

Likewise, we can identify "dangerous" variables in graph patterns, which might cause semantic differences:

Definition 2.5 *Let P' a sub-pattern of P of either the form $P' = (P_1 \textsf{\textit{OPT}}\, P_2)$ or $P' = (P_1 \textsf{\textit{UNION}}\, P_2)$. Any variable v in P' which violates the well-designedness-condition is called* possibly-null-binding *in P.*

Note that, so far we have only defined the semantics in terms of a pattern P and dataset DS, but not yet taken the result form V of query $Q = (V, P, DS)$ into account.

We now formally define solution tuples that were informally introduced in Sec. 2. Recall that by $\overline{V}$ we denote the tuple obtained from lexicographically ordering a set of variables in V. The notion $\overline{V[V' \to \mathsf{null}]}$ means that, after ordering V all variables from a subset $V' \subseteq V$ are replaced by null.

[7]We remark here, that transformation (5) possibly has problematic effects with respect to the assumption of safe filter expressions. This however only affects the translation in Sec. 4 and not the overall semantics.

Definition 2.6 *(*Solution Tuples*) Let $Q = (V, P, DS)$ be a* SPARQL *query, and θ a substitution in $[[P]]^x_{DS}$, then we call the tuple $V[(V \setminus vars(P)) \rightarrow \mathsf{null}]\theta$ a solution tuple of Q with respect to the x-joining semantics.*

Let us remark at this point, that as for the discussion of intuitivity of the different join semantics discussed in Examples 2.4–2.6, we did not yet consider combinations of different join semantics, e.g. using b-joins for OPT and c-joins for AND patterns. We leave this for further work.

3 Datalog and Answer Sets

In this paper we will use a very general form of Datalog commonly referred to as Answer Set Programming (ASP), i.e. function-free logic programming (LP) under the answer set semantics [1, 13]. ASP is widely proposed as a useful tool for various problem solving tasks in e.g. Knowledge Representation and Deductive databases. ASP extends Datalog with useful features such as negation as failure, disjunction in rule heads, aggregates [11], external predicates[10], etc. [8]

Let $Pred$, $Const$, Var, $exPr$ be sets of predicate, constant, variable symbols, and external predicate names, respectively. Note that we assume all these sets except $Pred$ and $Const$ (which may overlap), to be disjoint. In accordance with common notation in LP and the notation for external predicates from [9] we will in the following assume that $Const$ and $Pred$ comprise sets of numeric constants, string constants beginning with a lower case letter, or '`"`' quoted strings, and strings of the form ⟨quoted-string⟩^^⟨IRI⟩, ⟨quoted-string⟩@⟨valid-lang-tag⟩,[9] Var is the set of string constants beginning with an upper case letter. Given $p \in Pred$ an *atom* is defined as $p(t_1, \dots, t_n)$, where n is called the arity of p and $t_1, \dots, t_n \in Const \cup Var$.

Moreover, we define a fixed set of external predicates $exPr = \{rdf, isBLANK, isIRI, isLITERAL, =, \mathrel{!=}\}$ All external predicates have a fixed semantics and fixed arities, distinguishing *input* and *output terms*. The atoms $isBLANK[c](val)$, $isIRI[c](val)$, $isLITERAL[c](val)$ test the input term $c \in Const \cup Var$ (in square brackets) for being valid string representations of a Blank nodes, IRI References or RDF literals, returning an output value $val \in \{\mathtt{t}, \mathtt{f}, \mathtt{e}\}$, representing truth, falsity or an error, following the semantics defined in [21, Sec. 11.3]. For the rdf predicate we write atoms as $rdf[i](s, p, o)$ to denote that $i \in Const \cup Var$ is an input term, whereas $s, p, o \in Const \cup Var$ are output terms which may be bound by the external predicate. The external atom $rdf[i](s, p, o)$ is true if (s, p, o) is an RDF triple *entailed* by the RDF graph which is accessibly at IRI i. For the moment, here we consider simple RDF entailment [15] only. Finally, we write comparison atoms '$t_1 = t_2$' and '$t_1 \mathrel{!=} t_2$' in infix notation with $t_1, t_2 \in Const \cup Var$ and the obvious semantics of (lexicographic or numeric) (in)equality. Here, for $=$ either t_1 or t_2 is an output term, but at least one is an input term, and for $\mathrel{!=}$ both t_1 and t_2 are input terms.

Definition 3.1 *Finally, a* rule *is of the form*

$$h \;\; \text{:-} \;\; b_1,\; \dots,\; b_m,\; \mathtt{not}\; b_{m+1},\; \dots\; \mathtt{not}\; b_n. \tag{1}$$

where h and b_i ($1 \leq i \leq n$) are atoms, b_k ($1 \leq k \leq m$) are either atoms or external atoms, and `not` *is the symbol for negation as failure.*

We use $H(r)$ to denote the head atom h and $B(r)$ to denote the set of all body literals $B^+(r) \cup B^-(r)$ of r, where $B^+(r) = \{b_1, \dots, b_m\}$ and $B^-(r) = \{b_{m+1}, \dots, b_n\}$.

The notion of input and output terms in external atoms described above denotes the binding pattern. More precisely, we assume the following condition which extends the standard notion of safety (cf. [24]) in Datalog with negation: Each variable appearing in a rule must appear in a non-negated body atom or as an output term of an external atom.

Definition 3.2 *A* (logic) program Π *is defined as a set of safe rules r of the form (1).*

[8]We consider ASP, more precisely a simplified version of ASP with so-called HEX-programs [10] here, since it is up to date the most general extension of Datalog.

[9]Actually, for the purpose of this paper, we will disregard language tags and datatype tags, so we will not treat the last two types specially. We remark that for a full coverage of SPARQL we'd also need to consider variables in language and datatype tags.

The *Herbrand base* of a program Π, denoted HB_Π, is the set of all possible ground versions of atoms and external atoms occurring in Π obtained by replacing variables with constants from $Const$, where we define for our purposes by $Const$ the union of the set of all constants appearing in Π as well as all the literals, IRIs, and distinct constants for each blank node occurring in each RDF graph identified[10] by one of the IRIs in the (recursively defined) set I, where I is defined by the recursive closure of all IRIs appearing in Π and all RDF graphs identified by IRIs in I.[11] As long as we assume that the Web is finite the grounding of a rule r, $ground(r)$, is defined by replacing each variable with the possible elements of HB_Π, and the grounding of program Π is $ground(\Pi) = \bigcup_{r\in\Pi} ground(r)$.

An *interpretation relative to* Π is any subset $\mathcal{I} \subseteq HB_\Pi$ containing only atoms. We say that $\mathcal{I}$ is a *model* of atom $a \in HB_\Pi$, denoted $\mathcal{I} \models a$, if $a \in \mathcal{I}$. With every external predicate name $\text{lg} \in exPr$ with arity n we associate an $(n+1)$-ary Boolean function f_{lg} (called *oracle function*) assigning each tuple $(\mathcal{I}, t_1 \ldots, t_n)$ either 0 or 1. [12] We say that $\mathcal{I} \subseteq HB_\Pi$ is a *model* of a ground external atom $a = g[t_1, \ldots, t_m](t_{m+1}, \ldots, t_n)$, denoted $\mathcal{I} \models a$, iff $f_{\text{lg}}(\mathcal{I}, t_1, \ldots, t_n) = 1$.

The semantics we use here generalizes the answer-set semantics [13][13], and is defined using the *FLP-reduct* [11], which is more elegant than the traditional Gelfond-Lifschitz reduct of stable model semantics and ensures minimality of answer sets also in presence of external atoms.

Let r be a ground rule. We define (i) $\mathcal{I} \models B(r)$ iff $\mathcal{I} \models a$ for all $a \in B^+(r)$ and $\mathcal{I} \not\models a$ for all $a \in B^-(r)$, and (ii) $\mathcal{I} \models r$ iff $I \models H(r)$ whenever $\mathcal{I} \models B(r)$. We say that $\mathcal{I}$ is a *model* of a program Π, denoted $\mathcal{I} \models \Pi$, iff $\mathcal{I} \models r$ for all $r \in ground(\Pi)$.

The *FLP-reduct* [11] of Π with respect to $\mathcal{I} \subseteq HB_\Pi$, denoted $\Pi^{\mathcal{I}}$, is the set of all $r \in ground(\Pi)$ such that $\mathcal{I} \models B(r)$. $\mathcal{I} \subseteq HB_\Pi$ is an *answer set of* Π iff $\mathcal{I}$ is a minimal model of $\Pi^{\mathcal{I}}$.

We did not consider further extensions common to many ASP dialects here, namely disjunctive rule heads, strong negation [13]. We note that for non-recursive programs, i.e. programs where the predicate dependency graph is acyclic, the answer set is unique. For the pure translation which we will give in Sec. 4 where we will produce such non-recursive programs from SPARQL queries, we could equally take other semantics such as the well-founded [12] semantics into account, which coincides with the answer set semantics on non-recursive programs.

4 From SPARQL to Datalog

We are now ready to define a translation from SPARQL to Datalog which can serve straightforwardly to implement SPARQL within existing rules engines. We start with a translation for c-joining semantics, which we will extend thereafter towards s-joining and b-joining semantics.

4.1 Translation Π_Q^c

Let $Q = (V, P, DS)$, where $DS = (G, G_n)$ as defined above. We translate this query to a logic program Π_Q^c defined as follows.

$$\begin{aligned}\Pi_Q^c = & \{\texttt{triple}(S,P,O,\texttt{default})\ \texttt{:-}\ \texttt{rdf}[d](S,P,O).\ \mid\ d \in G\} \\ & \cup \{\texttt{triple}(S,P,O,g)\ \texttt{:-}\ \texttt{rdf}[g](S,P,O).\ \mid\ g \in G_n\} \\ & \cup\, \tau(V, P, \texttt{default}, 1)\end{aligned}$$

The first two rules serve to import the relevant RDF triples from the dataset into a 4-ary predicate `triple`. Under the dataset closedness assumption (see Def. 2.1) we may replace the second rule set, which imports the named graphs,

[10]By "identified" we mean here that the IRIs denote network accessible resources which correspond to RDF graphs/documents.

[11]Without loss of generality we may assume the number of accessible IRIs finite.

[12]The notion of an oracle function reflects the intuition that external predicates compute (sets of) outputs for a particular input, depending on the interpretation. The dependence on the interpretation is necessary for instance for defining the semantics of external predicates querying OWL [10] or computing aggregate functions.

[13]In fact, we use slightly simplified definitions from [9] for so-called HEX-programs, with the sole difference that we restrict ourselves to a fixed set of external predicates.

$$
\begin{array}{lcll}
\tau(V,(s,p,o),D,i) & = & \mathtt{answer_i}(\overline{V},D)\ \mathtt{:\!-}\ \mathtt{triple}(s,p,o,D). & (1)\\
\tau(V,(P'\ \textsf{AND}\ P''),D,i) & = & \tau(vars(P'),P',D,2*i)\ \cup\ \tau(vars(P''),P'',D,2*i+1)\ \cup & \\
 & & \mathtt{answer_i}(\overline{V},D)\ \mathtt{:\!-}\ \mathtt{answer_{2*i}}(\overline{vars(P')},D),\ \mathtt{answer_{2*i+1}}((\overline{vars(P'')},D). & (2)\\
\tau(V,(P'\ \textsf{UNION}\ P''),D,i) & = & \tau(vars(P'),P',D,2*i)\ \cup\ \tau(vars(P''),P'',D,2*i+1)\ \cup & \\
 & & \mathtt{answer_i}(\overline{V[(V\setminus vars(P'))\rightarrow \textsf{null}]},D)\ \mathtt{:\!-}\ \mathtt{answer_{2*i}}(\overline{vars(P')},D). & (3)\\
 & & \mathtt{answer_i}(\overline{V[(V\setminus vars(P''))\rightarrow \textsf{null}]},D)\ \mathtt{:\!-}\ \mathtt{answer_{2*i+1}}(\overline{vars(P'')},D). & (4)\\
\tau(V,(P'\ \textsf{MINUS}\ P''),D,i) & = & \tau(vars(P'),P',D,2*i)\ \cup\ \tau(vars(P''),P'',D,2*i+1)\ \cup & \\
 & & \mathtt{answer_i}(\overline{V[(V\setminus vars(P'))\rightarrow \textsf{null}]},D)\ \mathtt{:\!-}\ \mathtt{answer_{2*i}}(\overline{vars(P')},D), & (5)\\
 & & \qquad\qquad\qquad\qquad \mathtt{not}\ \mathtt{answer_{2*i}}'(\overline{vars(P')\cap vars(P'')},D). & \\
 & & \mathtt{answer_{2*i}}'(\overline{vars(P')\cap vars(P'')},D)\ \mathtt{:\!-}\ \mathtt{answer_{2*i+1}}(\overline{vars(P'')},D).\ \} & (6)\\
\tau(V,(P'\ \textsf{OPT}\ P''),D,i) & = & \tau(V,(P'\ \textsf{AND}\ P''),D,i)\ \cup\ \tau(V,(P'\ \textsf{MINUS}\ P''),D,i) & \\
\tau(V,(P\ \textsf{FILTER}\ R),D,i) & = & \tau(vars(P),P,D,2*i)\ \cup & \\
 & & LT(\mathtt{answer_i}(\overline{V},D)\ \mathtt{:\!-}\ \mathtt{answer_{2*i}}(\overline{vars(P)},D),R.) & (7)\\
\tau(V,(\textsf{GRAPH}\ g\ P),D,i) & = & \tau(V,P,g,i)\text{ for } g\in V\cup I & \\
 & & \mathtt{answer_i}(\overline{V},D)\ \mathtt{:\!-}\ \mathtt{answer_i}(\overline{V},g),\ \mathtt{isIRI}(g),\ \mathtt{not}\ g=\mathtt{default}. & (8)
\end{array}
$$

Alternate rules replacing (5)+(6):

$$
\begin{array}{lcll}
\mathtt{answer_i}(\overline{V[(V\setminus vars(P'))\rightarrow \textsf{null}]},D) & \mathtt{:\!-} & \mathtt{answer_{2*i}}(\overline{vars(P')},D),\ \mathtt{not}\ \mathtt{answer_{2*i}}'(\overline{vars(P')},D) & (5')\\
\mathtt{answer_{2*i}}'(\overline{vars(P')},D) & \mathtt{:\!-} & \mathtt{answer_{2*i}}(\overline{vars(P')},D),\ \mathtt{answer_{2*i+1}}(\overline{vars(P'')},D). & (6')
\end{array}
$$

Figure 2: Translation Π_Q^c from SPARQL queries semantics to Datalog.

by[14]:

$$\mathtt{triple}(S,P,O,G)\ \mathtt{:\!-}\ \mathtt{rdf}[G](S,P,O),HU(G),\mathtt{isIRI}(G).$$

Here, the predicate HU stands for "Herbrand universe", where we use this name a bit sloppily, with the intention to cover all the relevant part of $\mathcal{C}$, recursively importing all possible IRIs in order to emulate the dataset closedness assumption. HU, can be computed recursively over the input triples, i.e.

$$
\begin{array}{ll}
HU(X)\ \mathtt{:\!-}\ \mathtt{triple}(X,P,O,D). & HU(X)\ \mathtt{:\!-}\ \mathtt{triple}(S,X,O,D).\\
HU(X)\ \mathtt{:\!-}\ \mathtt{triple}(S,P,X,D). & HU(X)\ \mathtt{:\!-}\ \mathtt{triple}(S,P,O,X).
\end{array}
$$

The remaining program $\tau(V,P,\mathtt{default},1)$ represents the actual query translation, where τ is defined recursively as shown in Fig. 2.

As for FILTER expressions, one might miss details of the translation of complex filter expressions R in Fig. 2: By $LT(\cdot)$ we mean the set of rules resulting from disassembling complex filter expressions (involving '$\neg$','$\wedge$','$\vee$') according to the rewriting defined by Lloyd and Topor [17] where we have to obey the semantics for errors, following Definition 2.2. In a nutshell, the rewriting $LT-rewrite(\cdot)$ proceeds as follows: Complex filters involving $\neg$ are transformed by standard normal form transformations into negation normal form such that negation only occurs in front of atomic filter expressions. Note that this step might result in an exponential blowup of the resulting program size.[15] Conjunctions of filter expressions are simply disassembled to conjunctions of body literals, disjunctions are handled by splitting the respective rule for both alternatives in the standard way. The resulting rules involve possibly negated atomic filter expressions in the bodies. Here, for $v\in V$ the unary predicate $\textsf{BOUND}(v)$ is translated to $v=\textsf{null}$, $\neg\textsf{BOUND}(v)$ to $v\,!\!=\textsf{null}$. $\textsf{isBLANK}(v)$, $\textsf{isIRI}(v)$, $\textsf{isLITERAL}(v)$ and their negated forms are replaced by their corresponding external atoms (see Sec. 3) $\textsf{isBLANK}[v](\mathtt{t})$ or $\textsf{isBLANK}[v](\mathtt{f})$, etc., respectively.

The final program Π_Q^c implements the c-joining semantics in the following sense:

[14]The rule would be safe as defined in Sec. 3 even without adding $HU(G)$ to the body. Remember, we assumed the number of accessible IRIs and thus the extension of the external predicate $\mathtt{isIRI}$ finite. However, we defined the dataset closedness assumption as considering the recursive closure of IRIs occcurring within the dataset, which we emulate by HU.

[15]Lloyd and Topor can avoid this potential exponential blowup by introducing new auxiliary predicates. However, we cannot do the same trick, mainly for reasons of preserving safety of external predicates as defined in Sec. 3.

Proposition 4.1 (Soundness and completeness of Π_Q^c) *For each atom of the form* $\texttt{answer}_1(\vec{s}, \texttt{default})$ *in the unique answer set* M *of* Π_Q^c*,* $\vec{s}$ *is a solution tuple of* Q *with respect to the c-joining semantics, and all solution tuples of* Q *are represented by the extension of predicate* $\texttt{answer}_1$ *in* M*.*

Without giving a proof, we remark that the result follows if we convince ourselves that $\tau(V, P, D, i)$ emulates exactly the recursive definition of $[[P]]^x_{DS}$. Moreover, together with Proposition 2.3, we obtain soundness and completeness of Π_Q for b-joining and s-joining semantics as well for well-designed query patterns.

Corollary 4.2 *If the graph pattern* P *is well-designed, the extension of predicate* $\texttt{answer}_1$ *in the unique answer set* M *of* Π_Q^c *represents all and only the solution tuples for* $Q = (V, P, DS)$ *with respect to the* x*-joining semantics, for* $x \in \{b, c, s\}$*.*

4.2 Generalizing Π_Q^c to s-joining and b-joining semantics

Now, in order to obtain a proper translation for arbitrary, not necessarily well-designed patterns, we obviously need to focus our attention on the possibly-null-binding variables within the query pattern P. Let $vnull(P)$ denote the possibly-null-binding variables in a (sub)pattern P. We need to consider all rules in Fig. 2 which involve x-joins, i.e. the rules of the forms (2),(5) and (6). Since rules (5) and (6) do not make this join explicit, we will replace them by the equivalent rules (5') and (6') for the extended translations Π_Q^s and Π_Q^b, respectively. The "extensions" to s-joining and b-joining semantics can then be achieved by rewriting the rules (2) and (6').

The idea is to rename variables and add proper FILTER expressions to these rules in order to realize the b-joining and s-joining behavior for the variables in $V_N = vnull(P) \cap vars(P') \cap vars(P'')$.

4.2.1 Translation Π_Q^s

The more restrictive s-joining behavior can be achieved by adding FILTER expressions

$$R^s = (\bigwedge_{v \in V_N} BOUND(v))$$

to the rule bodies of (2) and (6'). The resulting rules are again subject to the LT-rewriting as discussed above for the rules of the form (7).

This is sufficient to filter out any joins involving null values, thus achieving s-joining semantics, and we denote the program rewritten that way as Π_Q^s.

4.2.2 Translation Π_Q^b

Obviously, b-joining semantics is more tricky to achieve, since we now have to relax the allowed joins in order to allow null bindings to join with *any* other value. We will again achieve this result by modifying rules (2) and (6') where we first do some variable renaming and then add respective FILTER expressions to these rules.

Step 1. We rename each variable $v \in V_N$ in the respective rule bodies to v' or v'', respectively, in order to disambiguate the occurrences originally from sub-pattern P' or P'', respectively. That is, for each rule (2) or (6'), we rewrite the body to:

$$\texttt{answer}_{2*\texttt{i}}(\overline{vars(P')[V_N \to V'_N]}, D), \texttt{answer}_{2*\texttt{i}+1}(\overline{vars(P'')[V_N \to V''_N]}, D).$$

Step 2. We now add the following FILTER expressions $R^b_{(2)}$ and $R^b_{(6')}$, respectively, to the resulting rule bodies which "emulate" the relaxed b-compatibility:

$$\begin{aligned} R^b_{(2)} = \textstyle\bigwedge_{v \in VN}(\quad & ((v = v') \land (v' = v'')) \lor ((v = v') \land \neg BOUND(v'')) \lor \\ & ((v = v'') \land \neg BOUND(v'))) \\ R^b_{(6')} = \textstyle\bigwedge_{v \in VN}(\quad & ((v = v') \land (v' = v'')) \lor ((v = v') \land \neg BOUND(v'')) \lor \\ & ((v = v') \land \neg BOUND(v'))) \end{aligned}$$

The rewritten rules are again subject to the LT rewriting. Note that, strictly speaking the filter expression introduced here does not fulfill the assumption of safe filter expressions, since it creates new bindings for the variable v. However, these can safely be allowed here, since the translation only creates valid input/output term bindings for the external Datalog predicate '$=$'.

The subtle difference between $R^b_{(2)}$ and $R^b_{(6')}$ lies in the fact that $R^b_{(2)}$ preferably "carries over" bound values from v' or v'' to v whereas $R^b_{(6')}$ always takes the value of v'. The effect of this becomes obvious in the translation of Example 2.6 in the Appendix.

In a real implementation we could avoid the potential exponential (with respect to $|V_N|$) blowup of the program size by unfolding the filter expressions in according to the LT rewriting by a cascading definition of predicates $\mathtt{join}_n$ which "emulate" b-joining behavior.

$$\begin{array}{l}
\mathtt{join}(X,X,X)\ \text{:-}\ HU(X).\quad \mathtt{join}(X,\mathsf{null},X)\ \text{:-}\ HU(X).\quad \mathtt{join}(\mathsf{null},X,X)\ \text{:-}\ HU(X).\\
\mathtt{join_1}(X_1',X_1'',X_1)\ \text{:-}\ \mathtt{join}(X_1',X_1'',X_1).\\
\mathtt{join_2}(X_1',X_2',X_1'',X_2'',X_1,X_2)\ \text{:-}\ \mathtt{join_1}(X_1',X_1'',X_1),\mathtt{join}(X_2',X_2'',X_2).\\
\mathtt{join_3}(X_1',X_2',X_3',X_1'',X_2'',X_3'',X_1,X_2,X_3)\ \text{:-}\ \mathtt{join_2}(X_1',X_2',X_1'',X_2'',X_1,X_2),\mathtt{join}(X_3',X_3'',X_3).\\
\vdots\\
\mathtt{join}_n(X_1',\ldots,X_n',X_1'',\ldots,X_n'',X_1,\ldots,X_n)\ \text{:-}\ \mathtt{join_{n-1}}(X_1',\ldots,X_{n-1}',X_1'',\ldots,X_{n-1}'',X_1,\ldots,X_{n-1}),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{join}(X_n',X_n'',X_n).
\end{array}$$

Instead of the FILTER expressions $R^b_{(2)}$ and $R^b_{(6')}$, we can now directly add the atom $\mathtt{join}_{|V_N|}(\overline{V_N'},\overline{V_N''},\overline{V_N})$ to the rewritten rule bodies from Step 1. The slightly different behavior of $R^b_{(2)}$ and $R^b_{(6')}$ is then accounted for by replacing variables changing the head of rules (6') to

$$\mathtt{answer_{2*i}}'(\overline{vars(P')[V_N \to V_N']},D)$$

The predicate HU used in the bodies of the first three rules is defined as in Sec. 4.1, and serves for making the rules safe.

We remark that this translation only solves the problem of exponential blowup at the surface, the evaluation of the $\mathtt{join}_n$ rules above in a bottom-up Datalog engine would still be exponential in the size of $|V_N|$. This is not surprising though, given the negative complexity results in [18]. See also the discussion on the translation of Example 2.5 in the appendix for more details.

In total, we obtain a program which Π^b_Q which reflects the normative b-joining semantics. Consequently, we get sound and complete query translations for all three semantics:

Corollary 4.3 (Soundness and completeness of Π^x_Q) *Given an arbitrary graph pattern P, the extension of predicate* $\mathtt{answer_1}$ *in the unique answer set M of Π^x_Q represents all and only the solution tuples for $Q = (V,P,DS)$ with respect to the x-joining semantics, for $x \in \{b,c,s\}$.*

In the following, we will drop the superscript x in Π_Q implicitly refer to the normative b-joining translation/semantics.

5 Possible Extensions

As it turns out, the embedding of SPARQL in the rules world opens a wide range of possibilities for combinations. In this section, we will first discuss some straightforward extensions of SPARQL which come practically for free with the translation to Datalog provided before. We will then discuss the use of SPARQL itself as a simple RDF rules language which allows to combine RDF fact bases with implicitly specified further facts and discuss the semantics thereof briefly. We will conclude this section with revisiting the open issue of entailment regimes covering RDFS or OWL semantics in SPARQL.

5.1 Additional Language Features

In this section we present two additional language features for SPARQL as a suggestion to the Data Access working group, which – though easy to add – would increase usability of SPARQL considerably in the opinion of the author.

5.1.1 Set Difference

As mentioned before, set difference is not present in the current SPARQL specification syntactically, though hidden, and would need to be emulated via a combination of OPTIONAL and FILTER constructs. As we defined the MINUS operator here in a completely modular fashion, it could be added straightforwardly as a separate keyword without affecting the semantics definition.

5.1.2 Nested ASK Queries

Nested queries are a distinct feature of SQL not present in SPARQL. We suggest a simple, but useful form of nested queries to be added: Boolean queries $Q_{\mathsf{ASK}} = (\emptyset, P_{\mathsf{ASK}}, DS_{\mathsf{ASK}}))$ with an empty result form (denoted by the keyword ASK) can be safely allowed within FILTER expressions as an easy extension fully compatible with our translation. Given query $Q = (V, P, DS)$, with sub-pattern $(P_1 \text{ FILTER } (\mathsf{ASK} Q_{\mathsf{ASK}}))$ we can modularly translate such subqueries by extending Π_Q with $\Pi_{Q'}$ where $Q' = (vars(P_1) \cap vars(P_{\mathsf{ASK}}), P_{\mathsf{ASK}}, DS_{\mathsf{ASK}}))$. Moreover, we have to rename predicate names $\mathtt{answer}_i$ to $\mathtt{answer^{Q'}}_i$ in $\Pi_{Q'}$. Some additional considerations are necessary in order to combine this within arbitrary complex filter expressions, and we probably need to impose well-designedness for variables shared between P and P_{ASK} similar to Def. 2.4. We leave more details as future work.

5.2 Result Forms and Solution Modifiers

We have covered only SELECT queries so far. As shown in the previous section, we can consider ASK queries equally by simply allowing empty result forms. A limited form of the CONSTRUCT result form, which allows to construct new triples could be emulated in our approach as well. Namely, we can allow queries of the form

$$Q_{\mathsf{C}} = (\mathsf{CONSTRUCT} P_{\mathsf{C}}, P, DS)$$

where P_{C} is a graph pattern consisting only of bNode-free triple patterns. We can model these by adding a rule

$$\mathtt{triple}(s, p, o, \mathsf{C}) \;\; \mathtt{:-} \;\; \mathtt{answer_1}(\overline{vars(P_{\mathsf{C}})}, \mathtt{default}),\ \mathsf{isIRI}[p](\mathtt{t}),\ s \mathrel{!=} \mathsf{null},\ o \mathrel{!=} \mathsf{null}. \qquad (2)$$

to Π_Q foreach triple pattern (s, p, o) in P_{C}. The result graph is then naturally represented in the answer set of the program extended that way in the extension of the predicate triple.[16]

5.3 SPARQL as a Rules Language

As it turns out with the extensions defined in the previous subsections, SPARQL itself may be viewed as an expressive rules language on top of RDF. CONSTRUCT statements have an obvious similarity with view definitions in SQL, and thus may be seen as rules themselves.

Intuitively, in the translation of CONSTRUCT we "stored" the new triples in a new triple outside the dataset DS, defining a new "context" C. We can imagine a similar construction in order to define the semantics of queries over datasets mixing such CONSTRUCT statements with RDF data in the same turtle file.

Let us assume such a mixed file containing CONSTRUCT rules and RDF triples web-accessible at IRI g, and a query $Q = (V, P, DS)$, with $DS = (G, G_n)$. The semantics of a query over a dataset containing g may then be defined by recursively adding $\Pi_{Q_{\mathsf{C}}}$ to Π_Q for any CONSTRUCT query Q_{C} in g plus the rules (2) above with their head changed to $\mathtt{triple}(s, p, o, g)$. We further need to add a rule

$$\mathtt{triple}(s, p, o, default) \;\; \mathtt{:-} \;\; \mathtt{triple}(s, p, o, g).$$

[16] As a sidenote, we remark that if implemented on top of a logic programming engine beyond Datalog, allowing for function symbols, we could also allow bNodes in the CONSTRUCT graph pattern, by replacing these with Skolem-functions.

for each $g \in G$, in order not to omit any of the implicit triples defined by such "CONSTRUCT rules". Analogously to the considerations for nested ASK queries, we need to rename the $\texttt{answer}_i$ predicates and $default$ constants in every subprogram Π_{Q_C} defined this way.

Naturally, the resulting programs possibly involve recursion, and, even worse, recursion over negation as failure. Fortunately, the general answer set semantics, which we use, can cope with this. For some important aspects on the semantics of such distributed rules and facts bases, we refer to [20], where we also outline an alternative semantics based on the well-founded semantics. A more in-depth investigation of the complexity and other semantic features of such a combination is on our agenda.

5.4 Revisiting Entailment Regimes

The current SPARQL specification does not treat entailment regimes beyond RDF simple entailment. Strictly speaking, even RDF entailment is already problematic as a basis for SPARQL query evaluation; a simple query pattern like $P = (?X, \mathsf{rdf{:}type}, \mathsf{rdf{:}Property})$ would have infinitely many solutions even on the empty (sic!) dataset by matching the infinitely many axiomatic triples in the RDF(S) semantics.

On the contrary, finite rule sets which approximate the RDF(S) semantics in terms of positive Datalog rules [20] have been implemented in systems like TRIPLE[17] or JENA[18]. Similarly, fragments and extensions of OWL [14, 3, 16] definable in terms of Datalog rule bases have been proposed in the literature. Such rule bases can be parametrically combined with our translations, implementing what one might call RDFS$^-$ or OWL$^-$ entailment at least. It remains to be seen whether the SPARQL working group will define such reduced entailment regimes.

More complex issues arise when combining a nonmonotonic query language like SPARQL with monotonic ontology languages such as OWL. Our embedding of SPARQL into a nonmonotonic rules language might provide valuable insights here, since it opens up a whole body of work done on combinations of such rules languages with first-order logic based ontology languages [9, 22].

6 Prototype

A prototype of the presented translation has been implemented on top of the dlvhex system, a flexible framework for developing extensions for the declarative Logic Programming Engine DLV[19]. The prototype is available as a plugin at `http://con.fusion.at/dlvhex/` for download. The web-page also provides an online interface for evaluation at `http://con.fusion.at/dlvhex/sparql-query-evaluation.php`, where the reader can check translation results for various example queries. We also added listings obtained by translating all sample queries from this paper in the appendix below.

Note that the output of the prototype might sligthly differ from the versions obtained from strictly implementing the translation outlined in this paper. For instance, in the current implementation indices may be named differently and joins of simple triple patterns are collapsed into single rules. We currently implement the c-joining and b-joining semantics and we plan to gradually extend the prototype towards the features mentioned in Sec. 5, in order to query mixed RDF+SPARQL rule and fact bases. Implementation of further extensions, such as the integration of aggregates typical for database query language – and recently defined for recursive Datalog programs in a declarative way compatible with the answer set semantics [11] – are on our agenda.

7 Conclusions & Related Work

In this paper, we presented three possible semantics for SPARQL based on Pérez et al. [18] which differ mainly in their treatment of joins and provided translations of all three semantics to Datalog with nonmonotonic negation.

We slightly extended Pérez et al.'s semantics in missing parts such as a more complete treatment of complex FILTER expressions and graph patterns. Let us remark that a very recent technical report by the same authors [19]

[17]`http://triple.semanticweb.org/`
[18]`http://jena.sourceforge.net/`
[19]`http://www.dlvsystem.com/`

encompasses some very similar extensions towards a more complete coverage of SPARQL. We discussed intuitive behavior of joins in the different semantics in several examples.

As it turned out, the s-joining semantics which is close to traditional treatment of joins over incomplete relations and the c-joining semantics are nicely embeddable into Datalog. The b-joining semantics which reflects the normative behavior as described by the current SPARQL specification is most difficult to translate. Finally, we suggested some extension of SPARQL, based on the provided translations.

We hope to have contributed to clarifying the relationships between the Query, Rules and Ontology layers of the Semantic Web architecture with the present work. A prototype implementation of the presented translation is available. We are currently not aware of any other engine implementing the full semantics defined in [18].

As for related work, we should finally not forget to remark that also earlier work by Cyganiak [6] was trying to embed SPARQL in more traditional relational algebra, in order to implement SPARQL on top of existing SQL engines, which we consider complementary to what we were doing here.

8 Acknowledgments

Special thanks go to Jos de Bruijn and Reto Krummenacher for discussions on earlier versions of this document, to Bijan Parsia, Jorge Pérez, and Andy Seaborne for valuable insights gained through various email-discussions, to Roman Schindlauer for his invaluable help on prototype implementation on top of dlvhex, and to the anonymous reviewers of the conference version of this report for various useful comments. This work is partially supported by the Spanish MEC under the project TIC-2003-9001 and by the EC funded projects TripCom (FP6-027324) and KnowledgeWeb (IST 507482).

References

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[2] D. Beckett. Turtle - Terse RDF Triple Language, Tech. Report, 4 Apr. 2006. `http://www.dajobe.org/2004/01/turtle/`

[3] J. de Bruijn, A. Polleres, R. Lara, D. Fensel. OWL DL vs. OWL Flight: Conceptual modeling and reasoning for the semantic web. In *Proc. WWW-2005*, 2005.

[4] F. Calimeri and G. Ianni. External sources of computation for Answer Set Solvers. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proc. LPNMR'05, Diamante, Italy*, volume 3662 of *LNCS*, pp. 105–118. Springer, 2005.

[5] J. Carroll, C. Bizer, P. Hayes, P. Stickler. Named graphs. *Journal of Web Semantics*, 3(4), 2005.

[6] R. Cyganiak. A relational algebra for sparql. Technical Report HPL-2005-170, HP Labs, Sept. 2005.

[7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[8] J. de Bruijn, E. Franconi, S. Tessaris. Logical reconstruction of normative RDF. *OWL: Experiences and Directions Workshop (OWLED-2005)*, 2005.

[9] T. Eiter, G. Ianni, A. Polleres, R. Schindlauer, H. Tompits. Reasoning with rules and ontologies. *Reasoning Web 2006*, volume 4126 of *LNCS*, pp. 93–127. Springer, 2006.

[10] T. Eiter, G. Ianni, R. Schindlauer, H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. *Int.l Joint Conf. on Art. Intelligence (IJCAI)*, 2005.

[11] W. Faber, N. Leone, G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. *Proc. of the 9th European Conf. on Art. Intelligence (JELIA 2004)*, 2004. Springer

[12] A. V. Gelder, K. Ross, J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *7th ACM Symp. on Principles of Database Systems*, 1988.

[13] M. Gelfond, V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[14] B. N. Grosof, I. Horrocks, R. Volz, S. Decker. Description logic programs: Combining logic programs with description logics. *Proc. WWW-2003*, 2003.

[15] P. Hayes. RDF semantics. W3C Recommendation, 10 Feb. 2004. `http://www.w3.org/TR/rdf-mt/`

[16] H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, 3(2), July 2005.

[17] J. W. Lloyd, R. W. Topor. Making prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[18] J. Pérez, M. Arenas, C. Gutierrez. Semantics and complexity of SPARQL. *The Semantic Web – ISWC 2006*, 2006. Springer.

[19] J. Pérez, M. Arenas, and C. Gutierrez. Semantics of SPARQL. Technical Report TR/DCC-2006-17, Universidad de Chile), May 2006.

[20] A. Polleres, C. Feier, A. Harth. Rules with contextually scoped negation. *Proc. 3rd European Semantic Web Conf. (ESWC2006)*, 2006. Springer.

[21] E. Prud'hommeaux, A. S. (ed.). SPARQL Query Language for RDF. *W3C Working Draft*, 4 Oct. 2006. `http://www.w3.org/TR/rdf-sparql-query/`

[22] R. Rosati. Reasoning with Rules and Ontologies. *Reasoning Web 2006*, volume 4126 of *LNCS*, pp. 128–151. Springer, 2006.

[23] SQL-99. Information Technology - Database Language SQL- Part 3: Call Level Interface (SQL/CLI). Technical Report INCITS/ISO/IEC 9075-3, INCITS/ISO/IEC, Oct. 1999. Standard specification.

[24] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[25] W3C. Rule Interchange Format Working Group, since 2005. `http://www.w3.org/2005/rules/wg`.

Appendix: Translations of Sample Queries

In this appendix we provide corresponding programs and answers for some sample queries mentioned in this paper, as well as some queries from the SPARQL spec [21] in order to exemplify the translation. We will, for each query, first give the query in SPARQL syntax as specified in [21], then provide the pattern in our notation and, finally provide the translated logic programs Π^c_Q, Π^s_Q,Π^b_Q together with the answers obtained in each of the three semantics.

For reasons of clarity, we explicitly specify the dataset in the form of FROM and FROM NAMED clauses in SPARQL syntax. As before, we omit and the leading 'http://' or other schema identifiers in IRIs and do not use namespace prefixes except in SPARQL syntax.

Query 1

We start with the query from Figure 1.

```
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?Y ?X
FROM <alice.org>
FROM <ex.org/bob>
WHERE { ?Y foaf:name ?X . }
```

This corresponds to $Q_1 = (V_1, P_1, DS_1)$ with $V_1 = \{?X, ?Y\}$, $DS_1 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, and

$$P_1 = (?Y,\ name,\ ?X)$$

$\Pi^c_{Q_1} = \Pi^s_{Q_1} = \Pi^b_{Q_1} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
answer1(X,Y,default) :- triple(Y,"name",X,default).
```

The query delivers the following answers:

```
{ answer1("Bob","_:a",default),
  answer1("Bob","_:c",default),
  answer1("Alice","alice.org#me",default) }
```

Query 2

We continue with the query from Example 2.1.

```
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?N
FROM <ex.org/bob>
WHERE { ?G foaf:maker ?M .
        GRAPH ?G { ?X foaf:name ?N } }
```

This corresponds to $Q_2 = (V_2, P_2, DS_2)$ with $V_1 = \{?N\}$, $DS_2 = (\{\texttt{ex.org/bob}\}, \emptyset)$, and

$$P_2 = ((?G,\ maker,\ ?M)\ \textsf{AND}\ (\textsf{GRAPH}\ ?G\ (?X,\ name,\ ?N)))$$

$\Pi^c_{Q_2} = \Pi^s_{Q_2} = \Pi^b_{Q_2} =$

```
  triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
  answer1(N,default) :- answer2(G,M,default), answer3(G,N,X,default).
  answer2(G,M,default) :- triple(G,"maker",M,default).
  answer3(G,N,X, default) :- answer3(G,N,X,G), isIRI(G), not G = default.
  answer3(G,N,X,G) :- triple(X,"maker",N,G).
```

This query delivers no answers, i.e. the extension of predicate `answer1` is empty in the (unique) answer set.
If we adopt the dataset closedness assumption, we need to add the rules

```
triple(S,P,O,G) :- rdf[G](S,P,O), HU(G), isIRI(G).
HU(X) :- triple(X,P,O,D).
HU(X) :- triple(S,X,O,D).
HU(X) :- triple(S,P,X,D).
HU(X) :- triple(S,P,O,X).
```

and would obtain the following answers:

```
{ answer1("_:b",default), answer1("_:a",default) }
```

Query 3

The next query is the one from Example 2.2.

```
  PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
  SELECT ?X
  FROM <ex.org/bob>
  WHERE { { ?X a foaf:Person .
            OPTIONAL { ?X foaf:dummy ?Y . }
          }
          FILTER ( !( isLITERAL (?Y) ) ) }
```

This corresponds to $Q_3 = (V_3, P_3, DS_3)$ with $V_3 = \{?X\}$, $DS_3 = (\{\texttt{ex.org/bob}\}, \emptyset)$, and

$$P_3 = (((?X,\ a, Person)\ \textsf{OPT}\ (?X,\ dummy,\ ?Y))\textsf{FILTER}(\neg(\textsf{isLITERAL}(?Y))))$$

$\Pi^c_{Q_3} = \Pi^s_{Q_3} = \Pi^b_{Q_3} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
answer1(X_X,default) :- answer2(X_X,X_Y,default), isLITERAL[Y](f).
answer2(X_X,X_Y, default) :- answer4(X_X,default), answer5(X_X,X_Y,default).
answer2(X_X,null,default) :- answer4(X_X,default), not answer5'(X_X,default).
answer5'(X_X,default) :- answer5(X_X,X_Y,default).
answer4(X_X,default)      :- triple(X_X,"a","Person",default).
answer5(X_X,X_Y,default) :- triple(X_X,"dummy",X_Y,default).
```

Note that unlike the usual Lloyd-Topor rewriting, the negation symbol is not rewritten by introducing a new predicate symbol and negating it with negation as failure `not` but to simply testing whether the filter expresion evaluates to `f`, i.e. $\perp$, see Definition 2.2. Thus, the query delivers no answers.

Query 4

We proceed with the query from Example 2.3.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
FROM <alice.org>
FROM <ex.org/bob>
WHERE { ?X foaf:name ?Name .
        ?X foaf:knows ?Friend }
```

This corresponds to $Q_4 = (V_4, P_4, DS_4)$ with $V_4 = \{?Friend, ?Name, ?X\}$, $DS_4 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, and

$$P_4 = ((?X, name, ?Name)\mathsf{AND}(?X, knows, ?Friend))$$

Again, all three translations remain the same, as the pattern is well-designed, i.e. $\Pi^c_{Q_4} = \Pi^s_{Q_4} = \Pi^b_{Q_4} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
answer1(Friend,Name,X_X,default) :- answer2(Friend,X,default),
                                    answer3(Name,X,default)
answer2(Friend,X,default) :- triple(X,"knows",Friend,default).
answer3(Name,X,default) :- triple(X,"name",Name,default).
```

We obtain the following answers:

```
{ answer1("_:b","Bob","_:a", default),
   answer1("_:c","Alice","alice.org#me") }
```

Compared with the result table in Example 2.3, we see that the "column order" has changed due to lexicographically ordering the variables.

Query 5

Looking at the query from Example 2.4, we get to the first example where the three translations actually diverge.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
FROM <alice.org>
FROM <ex.org/bob>
WHERE { { ?X1 a foaf:Person . OPTIONAL { ?X1 foaf:name ?N } }
        { ?X2 a foaf:Person . OPTIONAL { ?X2 foaf:nick ?N } } }
```

This corresponds to $Q_5 = (V_5, P_5, DS_5)$ with $V_5 = \{?N, ?X1, ?X2\}$, $DS_5 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, and

$$P_5 = (\ ((?X1, a, \texttt{Person})\ \mathsf{OPT}\ (?X1, \texttt{name}, ?N))\ \mathsf{AND}$$
$$((?X2, a, \texttt{Person})\ \mathsf{OPT}\ (?X2, \texttt{nick}, ?N))\ )$$

$\Pi^c_{Q_5} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,default) :- answer2(N,X1,default), answer3(N,X2,default).

answer2(N,    X1,default) :- answer4(X1,default), answer5(N,X1,default).
answer2(null,X1,default) :- answer4(X1,default), not answer5'(X1,default).
answer5'(X1,default)   :- answer5(N,X1,default).
```

```
answer4(X1,default)   :- triple(X1,"a","Person",default).
answer5(N,X1,default) :- triple(X1,"name",N,default).

answer3(N,   X2,default) :- answer6(X2,default), answer7(N,X2,default).
answer3(null,X2,default) :- answer6(X2,default), not answer7'(X2,default).
answer7'(X2,default) :- answer7(N,X2,default).

answer6(X2,default)   :- triple(X2,"a","Person",default).
answer7(N,X2,default) :- triple(X2,"nick",N,default).
```

Under c-joining semantics we obtain the following answers:

```
{ answer1(null,"_:b","_:a", default),
  answer1(null,"_:b","alice.org#me", default),
  answer1("Alice","alice.org#me","_:b", default) }
```

$\Pi^s_{Q_5} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,default) :- answer2(N,X1,default), answer3(N,X2,default), N != null.

answer2(N,   X1,default) :- answer4(X1,default), answer5(N,X1,default).
answer2(null,X1,default) :- answer4(X1,default), not answer5'(X1,default).
answer5'(X1,default)     :- answer4(X1,default), answer5(N,X1,default).

answer4(X1,default)   :- triple(X1,"a","Person",default).
answer5(N,X1,default) :- triple(X1,"name",N,default).

answer3(N,   X2,default) :- answer6(X2,default), answer7(N,X2,default).
answer3(null,X2,default) :- answer6(X2,default), not answer7'(X2,default).
answer7'(X2,default) :- answer6(X2,default), answer7(N,X2,default).

answer6(X2,default)   :- triple(X2,"a","Person",default).
answer7(N,X2,default) :- triple(X2,"nick",N,default).
```

Note here, that the only shared possibly-null-binding variable in a join is N at the top-level, so only the rule for `answer1` needs the additional filter condition. Under s-joining semantics we obtain the single answer:

```
{ answer1("Alice","alice.org#me","_:b", default) }
```

$\Pi^b_{Q_5} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).

answer1(N,X1,X2,default) :- answer2(N',X1,default), answer3(N'',X2,default),
                            N = N',  N' = N''.
answer1(N,X1,X2,default) :- answer2(N',X1,default), answer3(N'',X2,default),
                            N = N',  N'' = null.
answer1(N,X1,X2,default) :- answer2(N',X1,default), answer3(N'',X2,default),
                                 N = N'', N' = null.

answer2(N,   X1,default) :- answer4(X1,default), answer5(N,X1,default).
answer2(null,X1,default) :- answer4(X1,default), not answer5'(X1,default).
answer5'(X1,default)   :- answer4(X1,default), answer5(N,X1,default).

answer4(X1,default)    :- triple(X1,"a","Person",default).
answer5(N,X1,default) :- triple(X1,"name",N,default).

answer3(N,   X2,default) :- answer6(X2,default), answer7(N,X2,default).
answer3(null,X2,default) :- answer6(X2,default), not answer7'(X2,default).
answer7'(X2,default) :- answer6(X2,default), answer7(N,X2,default).

answer6(X2,default)    :- triple(X2,"a","Person",default).
answer7(N,X2,default) :- triple(X2,"nick",N,default).
```

Under b-joining semantics we obtain the following answers:

```
{ answer1(null,"_:b","_:a",default),
  answer1(null,"_:b","alice.org#me",default),
  answer1("Bobby","_:b","_:c",default),
  answer1("Bob","_:a","_:a",default),
  answer1("Bob","_:a","alice.org#me",default),
  answer1("Bob","_:c","_:a",default),
  answer1("Bob","_:c","alice.org#me",default),
  answer1("Alice","_:b","_:b",default),
  answer1("Alice","alice.org#me","_:b",default),
  answer1("Alice","alice.org#me","_:a",default),
  answer1("Alice","alice.org#me","alice.org#me",default) }
```

In the translated program $\Pi^b_{Q_5}$ above the complex FILTER expression

$$R^b_{(2)} = (\quad ((N = N') \wedge (N' = N'')) \ \vee \ ((N = N') \ \wedge \ \neg BOUND(N'')) \vee \\ ((N = N'') \ \wedge \ \neg BOUND(N')) \ )$$

is already decomposed. Actually, the interested reader might realize that we can write the three rules for predicate `answer1` equivalently, and avoiding variable renaming completely as follows:

```
answer1(N,X1,X2,default) :- answer2(N,X1,default), answer3(N,X2,default).
answer1(N,X1,X2,default) :- answer2(N,X1,default), answer3(null,X2,default).
answer1(N,X1,X2,default) :- answer2(null,X1,default), answer3(N,X2,default).
```

Our implemented prototype rather creates this form of rewriting, and we will use this simplified form of rewriting the b-joining FILTERs in the following.

Query 6

Let us now draw our attention to the second query from Example 2.5, which is concernde with idempotency:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
FROM <alice.org/singleTriple.rdf>
WHERE { { { ?X a foaf:Person } UNION { ?Y a foaf:Person } }
        { { ?X a foaf:Person } UNION { ?Y a foaf:Person } } }
```

This corresponds to $Q_6 = (V_6, P_6, DS_6)$ with $V_6 = \{?X, ?Y\}$, $DS_6 = (\{(\texttt{alice.org\#me}, a, \texttt{Person})\}, \emptyset)$, and the query pattern:

$$P_6 = (((?X, a, \texttt{Person}) \text{ UNION } (?Y, a, \texttt{Person})) \text{ AND } \\ ((?X, a, \texttt{Person}) \text{ UNION } (?Y, a, \texttt{Person})))$$

$\Pi^c_{Q_6} =$

```
triple(S,P,O,default) :- rdf["alice.org/singleTriple.rdf"](S,P,O).
answer1(X,Y,default) :- answer2(X,Y,default),answer3(X,Y,default).
answer2(X,null,default) :- triple(X,"a","Person",default).
answer2(null,Y,default) :- triple(Y,"a","Person",default).
answer3(X,null,default) :- triple(X,"a","Person",default).
answer3(null,Y,default) :- triple(Y,"a","Person",default).
```

Under c-joining semantics we obtain the following answers:

```
{ answer1(null,"alice.org/singleTriple.rdf#me",default),
  answer1("alice.org/singleTriple.rdf#me",null,default) }
```

$\Pi^s_{Q_6} =$

```
triple(S,P,O,default) :- rdf["alice.org/singleTriple.rdf"](S,P,O).
answer1(X,Y,default) :- answer2(X,Y,default),answer3(X,Y,default),
                        X != null, Y != null.
answer2(X,null,default) :- triple(X,"a","Person",default).
answer2(null,Y,default) :- triple(Y,"a","Person",default).
answer3(X,null,default) :- triple(X,"a","Person",default).
answer3(null,Y,default) :- triple(Y,"a","Person",default).
```

Under the s-joining semantics this query does not deliver any answers, proving that idempotency does not hold in general in this semantics.

```
{ answer1(null,"alice.org/singleTriple.rdf#me",default),
  answer1("alice.org/singleTriple.rdf#me",null,default) }
```

$\Pi^b_{Q_6} =$

```
triple(S,P,O,default) :- rdf["alice.org/singleTriple.rdf"](S,P,O).
answer1(X,Y,default) :- answer2(X,Y,default), answer3(X,Y,default).
answer1(X,Y,default) :- answer2(X,Y,default), answer3(X,null,default).
answer1(X,Y,default) :- answer2(X,null,default), answer3(X,Y,default).
answer1(X,Y,default) :- answer2(X,Y,default), answer3(null,Y,default).
answer1(X,Y,default) :- answer2(X,Y,default), answer3(null,null,default).
answer1(X,Y,default) :- answer2(X,null,default), answer3(null,Y,default).
answer1(X,Y,default) :- answer2(null,Y,default), answer3(X,Y,default).
answer1(X,Y,default) :- answer2(null,Y,default), answer3(X,null,default).
answer1(X,Y,default) :- answer2(null,null,default), answer3(X,Y,default).
answer2(X,null,default) :- triple(X,"a","Person",default).
answer2(null,Y,default) :- triple(Y,"a","Person",default).
answer3(X,null,default) :- triple(X,"a","Person",default).
answer3(null,Y,default) :- triple(Y,"a","Person",default).
```

With this example we can observe the potential exponential blowup of the translation under the b-joining semantics. Acually, in this particular example, we can observe that we could save some of the rules for `answer1`, e.g.

```
answer1(X,Y,default) :- answer2(null,null,default), answer3(X,Y,default).
answer1(X,Y,default) :- answer2(X,Y,default), answer3(null,null,default).
```

These rules could be safely dropped from $\Pi^{b}_{Q_6}$, since in neither of the involved UNION patterns both X and Y can be bound to null. Formalizing and implementing such optimizations is currently on our agenda, but, due to the complexity results in [18], we may not expect that in the worst-case an exponential number of rules needs to be added to emulate b-joins.

More precisely, the paper by Pérez et al. proofs worst case PSPACE query complexity for non-well-designed SPARQL queries. There is a similar result, i.e. PSPACE program complexity, for nonrecursive Datalog with negation [7]. Actually, we have shown in Sec. 4.2.2 a rewriting which does not have exponential blowup, by means of auxiliary $\texttt{join}_n$ predicates, definable for an upper bound of the number of possibly-null-binding variables in joins. An upper bound for this number of variables is always given by the size of the query pattern itself. However, for typical bottom-up Datalog engines such as dlvhex this does not make a difference since these engines instantiate programs prior to evaluation (called "grounding", such that the problem would only be shifted to the grounding. For this reason, we decided to go with the translation shown above in the current implementation.

The answers delivered under b-joining semantics also show the typical non-idempotency of this semantics.

```
{ answer1(null,"alice.org#me",default),
  answer1("alice.org#me",null,default),
  answer1("alice.org#me","alice.org#me",default) }
```

Query 7

We conclude with the query from Example 2.6.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
FROM <alice.org>
FROM <ex.org/bob>
WHERE { ?X a foaf:Person .
        OPTIONAL { ?X foaf:name ?XNAME }
        OPTIONAL { ?X foaf:nick ?XNAME } }
```

This corresponds to $Q_7 = (V_7, P_7, DS_7)$ with $V_7 = \{?X, ?Y\}$, $DS_7 = (\{\texttt{ex.org/bob}, \texttt{alice.org}\}, \emptyset)$, and the query pattern:

$$P_7 = ((((?X, a, \texttt{Person})\ \mathsf{OPT}\ (?X, \texttt{name}, ?XNAME))\\ \quad \mathsf{OPT}\ (?X, \texttt{nick}, ?XNAME))$$

$\Pi^{c}_{Q_7} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
answer1(X,XNAME,default) :- answer2(X,XNAME,default), answer3(X,XNAME,default).
answer1(X,XNAME,default) :- answer2(X,XNAME,default), not answer3′(X,XNAME,default).
answer3′(X,XNAME,default) :- answer3(X,XNAME,default).

answer2(X,XNAME,default) :- answer4(X,default), answer5(X,XNAME,default).
answer2(X,null,default) :- answer4(X,default), not answer5′(X,default).
answer5′(X,default) :- answer5(X,XNAME,default).

answer3(X,XNAME,default) :- triple(X,"nick",XNAME,default).
answer4(X,default) :- triple(X,"a","Person",default).
answer5(X,XNAME,default) :- triple(X,"name",XNAME,default).
```

Under the c-joining semantics, we obtain the following answers:

```
{ answer1("_:b",null,default),
  answer1("_:a","Bob",default),
  answer1("_:c","Bob",default),
  answer1("alice.org#me","Alice",default) }
```

$\Pi^s_{Q_7} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
answer1(X,XNAME,default) :- answer2(X,XNAME,default), answer3(X,XNAME,default).
answer1(X,XNAME,default) :- answer2(X,XNAME,default), not answer3'(X,XNAME,default).
answer3'(X,XNAME,default) :- answer2(X,XNAME,default), answer3(X,XNAME,default),
                             XNAME != null.

answer2(X,XNAME,default) :- answer4(X,default), answer5(X,XNAME,default).
answer2(X,null,default) :- answer4(X,default), not answer5'(X,default).
answer5'(X,default) :- answer4(X,default), answer5(X,XNAME,default).

answer3(X,XNAME,default) :- triple(X,"nick",XNAME,default).
answer4(X,default) :- triple(X,"a","Person",default).
answer5(X,XNAME,default) :- triple(X,"name",XNAME,default).
```

Under the s-joining semantics, this query delivers the same answers as for c-joining semantics:

```
{ answer1("_:b",null,default),
  answer1("_:a","Bob",default),
  answer1("_:c","Bob",default),
  answer1("alice.org#me","Alice",default) }
```

$\Pi^b_{Q_7} =$

```
triple(S,P,O,default) :- rdf["ex.org/bob"](S,P,O).
triple(S,P,O,default) :- rdf["alice.org"](S,P,O).
answer1(X,XNAME,default) :- answer2(X,XNAME,default), answer3(X,XNAME,default).
answer1(X,XNAME,default) :- answer2(X,null,default), answer3(X,XNAME,default).
answer1(X,XNAME,default) :- answer2(X,XNAME,default), answer3(X,null,default).
answer1(X,XNAME,default) :- answer2(X,XNAME,default), not answer3'(X,XNAME,default).
answer3'(X,XNAME,default) :- answer2(X,XNAME,default), answer3(X,XNAME,default).
answer3'(X,null,default) :- answer2(X,null,default), answer3(X,XNAME,default).
answer3'(X,XNAME,default) :- answer2(X,XNAME,default), answer3(X,null,default).

answer2(X,XNAME,default) :- answer4(X,default), answer5(X,XNAME,default).
answer2(X,null,default) :- answer4(X,default), not answer5'(X,default).
answer5'(X,default) :- answer4(X,default), answer5(X,XNAME,default).

answer3(X,XNAME,default) :- triple(X,"nick",XNAME,default).
answer4(X,default) :- triple(X,"a","Person",default).
answer5(X,XNAME,default) :- triple(X,"name",XNAME,default).
```

Under the b-joining semantics, this query delivers the following answers, including the expected result for `_:b`:

```
{ answer1("_:b","Alice",default),
  answer1("_:a","Bob",default),
  answer1("_:c","Bob",default),
  answer1("alice.org#me","Alice",default) }
```

For more example translations, we suggest to try out our prototype by either downloading dlvhex from `http://con.fusion.at/dlvhex/` or directly access the online interface for our prototype available at: `http://con.fusion.at/dlvhex/sparql-query-evaluation.php`