The Expressive Power of SPARQL

Renzo Angles and Claudio Gutierrez

Technical Report TR/DCC-2008-5
Department of Computer Science, Universidad de Chile
{rangles,cgutierr}@dcc.uchile.cl

Abstract. This paper studies the expressive power of SPARQL. The main result is that SPARQL and non-recursive safe Datalog with negation have equivalent expressive power, and hence, by classical results, SPARQL is equivalent from an expressive point of view to Relational Algebra. We present explicit generic rules of the transformations in both directions. Among other findings of the paper are the proof that negation can be simulated in SPARQL, that non-safe filters are superfluous, and that current SPARQL W3C semantics can be simplified to a standard compositional one.

1 Introduction

Determining the expressive power of a query language is crucial for understanding its capabilities and complexity, that is, what queries a user is able to pose, and to understand how complex the evaluation of queries is, issues that are central considerations to take into account when designing a query language.

The query language for RDF, SPARQL, has recently become a W3C recommendation [9]. In the RDF Data Access Working Group (WG) were it was designed, expressiveness concerns generated ample debate. Among the topics discussed we can mention the issue of introducing nesting in SPARQL, of defining a proper semantics for new features like negation, and to understand the complexity of possible new extensions.

This paper studies in depth the expressive power of SPARQL. A first issue addressed in this paper is the incorporation of negation to SPARQL. SPARQL provides explicit operators for join and union of graph patterns, even for specifying optional graph patterns, but it does not define explicitly the difference of graph patterns. Although intuitively it can be emulated via a combination of optional patterns and filter conditions (like negation as failure in logic programming), we show that there are several non-trivial issues to be addressed if one likes to define it inside the language. Being one of the most natural operators, it is not yet present in the official specification.

A second expressiveness issue refers to patterns with non-safe filter (those patterns (P FILTER C) for which there are variables in C not present in P). It turns out that these type of patterns, which have non-desirable properties, can be simulated by safe ones (i.e. those patterns where every variable occurring in C also occurs in P). This innocent result has important consequences for defining

a clean semantics, in particular a compositional and context-free one. In fact, we prove that for the fragment of patterns which have all filters safe, the official W3C semantics coincides with a standard compositional one. After these results, one may assume a standard compositional semantics for the whole SPARQL.

Thirdly, we compare the expressive power of SPARQL and non-recursive safe Datalog with negation (nr-Datalog$^\neg$). To determine the expressive power of SPARQL, first, we show –using the above results– that the W3C specification is equivalent to a well behaved and studied formalization with compositional semantics, which we will denote in this paper SPARQL$_C$ [6]. Then we compare SPARQL$_C$ with nr-Datalog$^\neg$. First we show that SPARQL$_C$ is contained in nr-Datalog$^\neg$ by defining transformations (for databases, queries, and solutions) from SPARQL$_C$ to nr-Datalog$^\neg$, and we prove that the result of evaluating a SPARQL$_C$ query is equivalent, via the transformations, to the result of evaluating (in nr-Datalog$^\neg$) the transformed query. Second, we show that nr-Datalog$^\neg$ is contained in SPARQL$_C$ using a similar approach. These two results prove that nr-Datalog$^\neg$ and SPARQL$_C$ are equivalent. It is important to remark that the transformations used are explicit and simple, and in all steps bag semantics is considered.

Finally, and by far, the most important result of the paper, is the proof that SPARQL has the same expressive power of Relational Algebra under bag semantics (which is the one of SPARQL). Relational Algebra is probably one of the most studied languages, and has become, on one hand, a favorite by theoreticians because of a proper balance between expressiveness and complexity. In fact, it is well known that Relational Algebra has the same expressive power as Relational Calculus (First Order logic without functions) and of nr-Datalog$^\neg$ [1]. On the other hand, from a practical point of view, Relational Algebra is at the core of the most popular and successful database query language, SQL.

The result that SPARQL is equivalent in its expressive power to nr-Datalog$^\neg$, and hence equivalent to Relational Algebra, has important implications which are not discussed in this paper. Some examples are: the translation of some results from Relational Algebra into SPARQL, the settlement of several open questions about expressiveness of SPARQL, the expressive power added by the operator *bound* in combination with optional patterns.

Related Work. The W3C recommendation SPARQL is from January 2008. Then it is no surprise that little work has been done in the formal study of its expressive power. Several conjectures were raised during the WG sessions [1]. Furche et al. [3] surveyed expressive features of query languages for RDF including (old versions of) SPARQL in order to compare them systematically. But there is no particular analysis of the expressive power of SPARQL.

Related to our work here we can mention the following works.

Cyniak [2] presents a translation of SPARQL into Relational Algebra considering only a core fragment of SPARQL. His work is extremely useful to implement

[1] See `http://lists.w3.org/Archives/Public/public-rdf-dawg-comments/`, especially the years 2006 and 2007.

and optimize SPARQL in SQL engines. At the level of analysis of expressive issues it presents a list of problems that should be solved (many of which still persist), like the filter scope problem and the nested optional problem.

Polleres [8] proves the inclusion of the fragment of patterns with safe filters of SPARQL into Datalog by giving a precise and correct set of rules. Schenk [10] proposes a formal semantics for SPARQL based on Datalog, but concentrates on complexity more than expressiveness issues. Both works do not consider bag semantics of SPARQL in their translations.

The work of Perez et al. [6] and the technical report [7], that gave the formal basis for SPARQL_C compositional semantics, addressed several expressiveness issues, but no systematic study of expressive power of the language was done.

The paper is organized as follows. In Section 2 we present preliminary material to make more self contained the paper. Section 3 presents the study of negation. Section 4 studies non-safe filter patterns. Section 5 proves that SPARQL_{WG} and SPARQL_C have the same expressive power. Section 6 proves that SPARQL_C and nr-Datalog$^\neg$ have the same expressive power. Section 7 presents conclusions. Additionally, we included in this version three optional appendixes (which collects standard reference material that could save time to the reader).

2 Preliminaries

2.1 SPARQL

The essential results of this paper are comparisons of the expressive power of query languages. We will consider the following languages: SPARQL_{WG}, the W3C recommendation language SPARQL (it includes syntax and semantics); SPARQL^S_{WG}, the restriction of SPARQL_{WG} to filter-safe queries, that is, queries where for each occurrence of a pattern $(P \,\text{FILTER}\, C)$, it holds that every variable occurring in C also occurs in P; SPARQL_C, the formalization of SPARQL given in [6], with its algebraic syntax and compositional semantics; and non-recursive safe Datalog with negation (nr-Datalog$^\neg$), a standard language used in the Database community (see for example [1] and [5]).

We will follow in this paper the algebraic syntax of SPARQL_C, better suited to do formal analysis and processing than the syntax presented by the WG specification. There is an easy and intuitive way of translating back and forth between both syntax formalisms, which we will not detail here.

The two star languages in this study will be SPARQL_{WG} and SPARQL_C whose main difference resides in their respective semantics. SPARQL_C follows a compositional semantics, denoted by $[\![\cdot]\!]$. The SPARQL_{WG} semantics, denoted in this study by $\langle\!\langle \cdot \rangle\!\rangle$, is a mixture of compositional and operational semantics, where the meaning of certain expressions (patterns) depend on their context. We will explicitly indicate at each step, when necessary, which semantics is used.

In this paper we restrict to outputs in the form of SELECT queries. Recall that SPARQL has three other possible outputs: ASK which returns a yes or no if a query pattern has a solution, CONSTRUCT which outputs a graph constructed

using the solution values from graph pattern matching, and DESCRIBE which returns a single result RDF graph containing RDF data about resources. It is not difficult to see that this restriction does not harm the generality of the results.

We also will avoid patterns with blank nodes in order to concentrate on the fragment most common in practice and to avoid features still not well developed theoretically nor tested extensively.

Finally, for the sake of theoretical cleanness, we will leave out diverse small features (types, XML literals, some atomic filter constraints, etc.) which do not affect the essence of our results.

Syntax of SPARQL$_C$. Assume the existence of an infinite set V of variables disjoint from T. A tuple from $(T \cup V) \times (I \cup V) \times (T \cup V)$ is called a *triple pattern*. We denote by $\text{var}(\alpha)$ the function which returns the set of variables occurring in the structure α.

A *SPARQL Query* is a tuple (R, F, P)[2] where R is a result query form, F is a set –possibly empty– of dataset clauses, and P is a graph pattern. We will assume the *safe result condition*, that is, if $?X \in \text{var}(R)$ then $?X \in \text{var}(P)$. Next we define each component:

(1) If $W \subset V$ is a set of variables and H is a set of triple patterns (called a *graph template*), the expressions SELECT W, CONSTRUCT H, and ASK are *result query forms*.
(2) If $u \in I$ then FROM u and FROM NAMED u are *dataset clauses*.
(3) A *filter constraint* is defined recursively as follows:
   – If $?X, ?Y \in V$ and $u \in I \cup L$, then $?X = u$, $?X = ?Y$, bound($?X$), isIRI($?X$), isLiteral($?X$), and isBlank($?X$) are *atomic filter constraints*.[3]
   – If C_1 and C_2 are filter constraints then $(\neg C_1)$, $(C_1 \wedge C_2)$, and $(C_1 \vee C_2)$ are *complex filter constraints*.
(4) A *graph pattern* is defined recursively as follows:
   – A triple pattern is a graph pattern.
   – If P_1 and P_2 are graph patterns then $(P_1$ AND $P_2)$, $(P_1$ UNION $P_2)$, and $(P_1$ OPT $P_2)$ are graph patterns.
   – If P is a graph pattern and C is a filter constraint, then $(P$ FILTER $C)$ is a graph pattern.
   – If P is a graph pattern and $u \in I \cup V$ then $(u$ GRAPH $P)$ is a graph pattern.

2.2 Comparing Expressive Power of Languages

By the *expressive power* of a query language, we understand the set of all queries expressible in that language [1,5]. In order to determine the expressive power of a query language L, usually one chooses a well-studied query language L' and

[2] In this paper we do not consider solution modifiers.
[3] For a complete list of atomic filter constraints see [9].

compares L and L' in their expressive power. Two query languages have the same expressive power if they express exactly the same set of queries.

A given query language is defined as a triple $(\mathcal{Q}, \mathcal{D}, \mathcal{S}, \text{eval})$, where $\mathcal{Q}$ is a set of queries, $\mathcal{D}$ is a set of databases, $\mathcal{S}$ is a set of solutions, and $\text{eval} : \mathcal{Q} \times \mathcal{D} \to \mathcal{S}$ is the evaluation function. The evaluation of a query $Q \in \mathcal{Q}$ on a database $D \in \mathcal{D}$ is denoted $\text{eval}(Q, D)$ (usually $\text{eval}(Q, D)$ is simply denoted $Q(D)$ if no confusion arises).

Given a language $L = (\mathcal{Q}, \mathcal{D}, \mathcal{S}, \text{eval})$, two queries Q_1, Q_2 of L are *equivalent*, denoted $Q_1 \equiv Q_2$, if they return the same answer for all input databases, i.e., $\text{eval}(Q_1, D) = \text{eval}(Q_2, D)$ for every $D \in \mathcal{D}$. Let L_1, L_2 be two fragments of L. We say that L_1 is *contained* in L_2, if and only if for every query Q_1 in L_1 there exists a query Q_2 in L_2 such that $Q_1 \equiv Q_2$.

To compare two query languages with different syntax and semantics require having a common data and language setting to do the comparison. Let $L_1 = (\mathcal{Q}_1, \mathcal{D}_1, \mathcal{S}_1, \text{eval}_1)$ and $L_2 = (\mathcal{Q}_2, \mathcal{D}_2, \mathcal{S}_2, \text{eval}_2)$ be two query languages. We now say that L_1 is *contained* in L_2 if and only if there are bijective data transformations $\mathcal{T}_D : \mathcal{D}_1 \to \mathcal{D}_2$, and $\mathcal{T}_S : \mathcal{S}_1 \to \mathcal{S}_2$ and query transformation $\mathcal{T}_Q : \mathcal{Q}_1 \to \mathcal{Q}_2$, such that for all $Q_1 \in \mathcal{Q}_1$ and $D_1 \in \mathcal{D}_1$ it holds

$$\mathcal{T}_S(\text{eval}_1(Q_1, D_1)) = \text{eval}_2(\mathcal{T}_Q(Q_1), \mathcal{T}_D(D_1)).$$

We say that two query languages L_1, L_2 are *equivalent* if and only if L_1 is contained in L_2 and L_2 is contained in L_1.

3 Expressing Difference of Patterns in SPARQL$_{WG}$

We will start the study of expressive capabilities of SPARQL by the difference operator.

The SPARQL specification indicates that it is possible to test if a graph pattern does not match ([9] Sec. 11.4.1) via a combination of optional patterns and filter conditions (like negation as failure in logic programming). Nevertheless, it seems that this feature has not been studied formally. In this section we analyze in depth the scope and limitations of this approach.

Let P_1 and P_2 be two graph patterns and consider a dataset D having active graph G. We will denote by $(P_1 \,\text{MINUS}\, P_2)$ the "difference" of P_1 and P_2. The informal meaning for $\langle\langle P_1 \,\text{MINUS}\, P_2 \rangle\rangle_G^D$ would be "the set of mappings that match P_1 and does not match P_2". Formally:

Definition 1 (Semantics of the MINUS **operator).** *Let P_1 and P_2 be graph patterns. The evaluation of a graph pattern $(P_1 \,\text{MINUS}\, P_2)$ over a dataset G with active graph G is defined as follows*

$$\langle\langle P_1 \,\text{MINUS}\, P_2 \rangle\rangle_G^D = \langle\langle P_1 \rangle\rangle_G^D \setminus \langle\langle P_2 \rangle\rangle_G^D,$$

where the operator $\setminus$ between two sets of mappings Ω_1 and Ω_2 is defined as follows:

$$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \textit{ for all } \mu_2 \in \Omega_2,\ \mu_1 \textit{ and } \mu_2 \textit{ are not compatible}\}.$$

A naive implementation of the difference operator by means of the rest of SPARQL operators would be the graph pattern $((P_1 \,\mathrm{OPT}\, P_2)\,\mathrm{FILTER}\, C)$ where C is a clause asserting that there is no mapping matching P_2. This means, for each mapping $\mu \in \langle\!\langle P_1 \,\mathrm{OPT}\, P_2 \rangle\!\rangle_G^D$ at least one variable $?X$ in P_2 does not match (i.e., $?X$ is unbounded). We have two problems with this solution:

- The variable $?X$ cannot be an arbitrary variable. For example, P_2 could be in turn an optional pattern $(P_3 \,\mathrm{OPT}\, P_4)$ where only the variables in P_3 are relevant.
- If $\mathrm{var}(P_2) \setminus \mathrm{var}(P_1) = \emptyset$ we have no variable $?X$ to check unboundedness.

The above two problems motivate the introduction of the notions of non-optional variables and copy patterns.

The set of *non-optional variables* of a graph pattern P, denoted $\mathrm{nov}(P)$, is a subset of the variables of P defined recursively as follows: $\mathrm{nov}(t) = \mathrm{var}(t)$ for a triple pattern t; $\mathrm{nov}(P_1 \,\mathrm{AND}\, P_2) = \mathrm{nov}(P_1) \cup \mathrm{nov}(P_2)$; $\mathrm{nov}(P_1 \,\mathrm{UNION}\, P_2) = \mathrm{nov}(P_1) \cup \mathrm{nov}(P_2)$; $\mathrm{nov}(P_1 \,\mathrm{OPT}\, P_2) = \mathrm{nov}(P_1)$; $\mathrm{nov}(n \,\mathrm{GRAPH}\, P_1)$ is either $\mathrm{nov}(P_1)$ when $n \in I$ or $\mathrm{nov}(P_1) \cup \{n\}$ when $n \in V$; and $\mathrm{nov}(P_1 \,\mathrm{FILTER}\, C) = \mathrm{nov}(P_1)$. Intuitively, a variable in $\mathrm{nov}(P)$ is a variable of P which does not occur in an optional pattern of P, i.e., $\mathrm{nov}(P)$ indicates the variables that necessarily must be bounded in any mapping of P.

Let $\phi : V \to V$ be a variable-renaming function. Given a graph pattern P, assume that $\phi(P)$ denotes an isomorphic copy of P whose variables have been renamed according to ϕ and satisfying that $\mathrm{var}(P) \cap \mathrm{var}(\phi(P)) = \emptyset$. Then we say that $\phi(P)$ is a *copy pattern* of P.

Theorem 1 (Difference of graph patterns). *Let P_1 and P_2 be graph patterns. Then:*

$$(P_1 \,\mathrm{MINUS}\, P_2) = ((P_1 \,\mathrm{OPT}((P_2 \,\mathrm{AND}\, \phi(P_2))\,\mathrm{FILTER}\, C_1))\,\mathrm{FILTER}\, C_2), \quad (1)$$

where:

- *C_1 is the filter constraint $(?X_1 = ?X'_1 \wedge \cdots \wedge ?X_n = ?X'_n)$ where $?X_i \in \mathrm{var}(P_2)$ and $?X'_i = \phi(X_i)$ for $1 \leq i \leq n$.*
- *C_2 is the filter constraint $\neg(\mathrm{bound}(?X'_1) \wedge \cdots \wedge \mathrm{bound}(?X'_m))$ where $X'_j \in \mathrm{nov}(\phi(P_2))$ for $1 \leq j \leq m$, such that for each $X'_j = \phi(X_j)$ it satisfies that $X_j \in \mathrm{nov}(P_2) \setminus \mathrm{var}(P_1)$.*

Proof. Let P be the graph pattern $(P_1 \,\mathrm{MINUS}\, P_2)$ and P' be the right hand side of (1). We will prove that $\langle\!\langle P \rangle\!\rangle = \langle\!\langle P' \rangle\!\rangle$.

(a) *Evaluation of $\langle\!\langle P \rangle\!\rangle$:* By definition, a mapping μ_1 is in $\langle\!\langle P \rangle\!\rangle$ if and only if $\mu_1 \in P_1$ and for every mapping $\mu_2 \in \langle\!\langle P_2 \rangle\!\rangle$, μ_1 and μ_2 are not compatible.

(b) *Evaluation of $\langle\!\langle P' \rangle\!\rangle$:* To simplify the idea of the proof, we will reduce P' to the pattern $((P_1 \,\mathrm{OPT}\, P_2)\,\mathrm{FILTER}\, C_2)$ where C_2 is $\neg(\mathrm{bound}(?X_1) \wedge \cdots \wedge \mathrm{bound}(?X_m))$ for $?X_i \in \mathrm{nov}(P_2)$. [4]

[4] This reduction does not diminish the generality of the proof because $\phi(P_2)$ and C_1 occurs into P' to solve the case when $\mathrm{nov}(P_2) \setminus \mathrm{var}(P_1) = \emptyset$ (See Note 1).

Then we have that a mapping μ is in $\langle\langle P' \rangle\rangle$ if and only if $\mu \in \langle\langle P_1 \,\text{OPT}\, P_2 \rangle\rangle$ and $\mu \models C_2$. Given $\mu_1 \in \langle\langle P_1 \rangle\rangle$, we have that $\mu \in \langle\langle P_1 \,\text{OPT}\, P_2 \rangle\rangle$ if and only if either (i) $\mu = \mu_1 \cup \mu_2$ when μ_1 is compatible with $\mu_2 \in \langle\langle P_2 \rangle\rangle$; or (ii) $\mu = \mu_1$ when for all $\mu_2 \in \langle\langle P_2 \rangle\rangle$, μ_1 and μ_2 are not compatible.
Note that, in case (i) $\mu(?X)$ is bounded for each variable $?X \in \text{nov}(P_2) \setminus \text{var}(P_1)$, whereas in case (ii) $\mu(?X)$ is unbounded. Given that C_2 contains the negation of a conjunction of clauses $\text{bound}(?X)$ for each variable $?X \in \text{nov}(P_2) \setminus \text{var}(P_1)$, we have that only case (ii) satisfies $\mu \models C_2$ (Note that here is critical the fact that in C_2 are only variables in $\text{nov}(P_2)$).
In case (ii) we have that $\mu = \mu_1$ and we can assure that $\mu_1 \in \langle\langle P_1 \rangle\rangle$ and for every mapping $\mu_2 \in \langle\langle P_2 \rangle\rangle$, μ_1 and μ_2 are not compatible, that is, $\mu_1 \in \langle\langle P' \rangle\rangle$. Therefore, $\langle\langle P' \rangle\rangle$ has exactly the same mappings as the evaluation of $\langle\langle P \rangle\rangle$ showed in (a), and this concludes the proof.

Note 1 (Why the copy pattern $\phi(P)$ is necessary?).
Consider to reduce the graph pattern presented in Theorem 1 by avoiding the copy pattern $\phi(P_2)$ and the filter constraint C_1.[5] Then we have the graph pattern $((P_1 \,\text{OPT}\, P_2) \,\text{FILTER}\, C_2)$ where C_2 is a clause asserting that there is no mapping matching P_2. Such restriction can be expressed in SPARQL with the filter constraint $\neg(\text{bound}(?X_1) \wedge \cdots \wedge \text{bound}(?X_m))$ such that $X_j \in \text{var}(P_2) \setminus \text{var}(P_1)$, that is, "there exists no mapping bounding each variable $?X_j$ occurring in P_2 but not occurring in P_1".

Note that such implementation would be valid when $\text{var}(P_2) \setminus \text{var}(P_1) \neq \emptyset$, i.e., it would work when *there exist variables* to check unboundedness. However a problem arises when $\text{var}(P_2) \setminus \text{var}(P_1) = \emptyset$. For example, consider the patterns $P_1 = (?X, \text{name}, ?N)$, $P_2 = (?X, \text{lastname}, \text{"Perez"})$. The above for $(P_1 \,\text{MINUS}\, P_2)$ will give a pattern with filter condition $C_2 = \emptyset$ because there are no variables in $\text{var}(P_2) \setminus \text{var}(P_1)$, thus, loosing the filtering:

$$(?X, \text{name}, ?N) \,\text{OPT}(?X, \text{lastname}, \text{"Perez"}).$$

Note that, it is not possible to check if variable $?X$ is unbounded in the pattern $(?X, \text{lastname}, \text{"Perez"})$, because –to satisfy the entire pattern– variable $?X$ must have already been bound in the pattern $(?X, \text{name}, ?N)$.

To avoid this problem, the graph pattern P_2 is replaced by the graph pattern $((P_2 \,\text{AND}\, \phi(P_2)) \,\text{FILTER}\, C_1)$ where $\phi(P_2)$ is a copy of P_2 where variables have been renamed and whose relations of equality with the original ones are defined in C_1. Then, the implementation of $(P_1 \,\text{MINUS}\, P_2)$ in the example will be

$$\begin{aligned}&(((?X, \text{name}, ?N) \,\text{OPT}\\ &\qquad(((?X, \text{lastname}, \text{"Perez"}) \,\text{AND}(?X', \text{lastname}, \text{"Perez"}))\\ &\qquad\qquad\text{FILTER}(?X = ?X'))) \,\text{FILTER}(\neg\, \text{bound}(?X'))).\end{aligned}$$

[5] To simplify the example, we will assume that P_2 does not contain optional patterns, hence the issue of optional variables does not matter here.

where the filter constraint $C_2 = (\neg\,\text{bound}(?X'))$ has been defined using variables from $\phi(P_2) = (?X', \text{lastname}, \text{“Perez”})$.

Note that the inclusion of copy patterns could introduce an exponential blow-up in the size of the pattern. A possible optimization (still inside the syntax of SPARQL) is to replace $\phi(P_2)$ by a “universal” pattern P_2^u. Here “universal” means that a variable $?X$ matches any triple. An implementation of universal pattern for a variable $?X$ would be

$$u(?X) = ((?X, ?Y_1, ?Y_2)\,\text{UNION}(?Y_3, ?X, ?Y_4)\,\text{UNION}(?Y_5, Y_6, ?X)),$$

where $?Y_j$ are fresh variables. Then, the universal pattern P_2^u will be:

$$(u(?X_1)\,\text{AND}\cdots\text{AND}\,u(?X_n)) \text{ where } ?X_i \in \text{var}(P_2).$$

Note 2 (Why non-optional variables?).
We justify the use of non-optional variables by showing that the naive implementation does not work in general. For example, consider the graph pattern

$$P = ((?X, \text{name}, ?N)\,\text{MINUS}((?X, \text{knows}, ?Y)\,\text{OPT}(?Y, \text{mail}, ?Z)))$$

and the RDF graph

$$G = \{\ (\text{a,name,}n_a),\ (\text{b,name,}n_b),\ (\text{b,knows,c}),\ (\text{b,mail,}m_b),\ (\text{c,name,}n_c),\ (\text{c,knows,d}),\ (\text{d,name,}n_d),\ (\text{d,mail,}m_d)\ \}$$

The evaluation of P over graph G is the set of mappings:

?X	?N
a	n_a
d	n_d

(*)

Now, consider the naive implementation of difference between graph patterns without enforcing the restriction of non-optional variables presented in Theorem 1, that is, the restriction $X'_j \in \text{nov}(\phi(P_2))$ in filter constraint C_2. Let us denote $P_1 = (?X, \text{name}, ?N)$, $P_2 = (?X, \text{knows}, ?Y)$, and $P_3 = (?Y, \text{mail}, ?Z)$, and $P_4 = (P_2\,\text{OPT}\,P_3)$. We would get the following pattern:

$$P' = ((P_1\,\text{OPT}(P_2\,\text{OPT}\,P_3))\,\text{FILTER}(\neg(\text{bound}(?Y) \wedge \text{bound}(?Z))))^6.$$

Note that the restriction of non-optional variables has been violated here, because although variable $?Z$ is optional in pattern $(P_2\,\text{OPT}\,P_3)$, it occurs in the filter constraint $(\neg(\text{bound}(?Y) \wedge \text{bound}(?Z))$.

The evaluation of P' over graph G is described step-by-step as follows:

$[\![P_1]\!]_G =$

?X	?N
a	n_a
b	n_b
c	n_c
d	n_d

$[\![P_2]\!]_G =$

?X	?Y
b	c
c	d

$[\![P_3]\!]_G =$

?Y	?Z
b	m_b
d	m_d

$[\![P_4]\!]_G =$

?X	?Y	?Z
b	c	
c	d	m_d

[6] For simplicity we do not consider the copy pattern $\phi(P_4)$.

$[\![(P_1 \text{ OPT } P_4)]\!]_G =$

?X	?N	?Y	?Z
a	n_a		
b	n_b	c	
c	n_c	d	m_d
d	n_d		

$[\![P']\!]_G =$

?X	?N	?Y	?Z
a	n_a		
b	n_b	c	
d	n_d		

It is not difficult to see that the evaluation of P' differs from P for variables $?X$ and $?N$. To show the problem, consider the following informal semantics: a mapping μ matches pattern P' if and only if μ matches P_1 and μ does not match P_4 (recall that P_4 is $((?X, \text{knows}, ?Y) \text{ OPT}(?Y, \text{mail}, ?Z))$). This latter condition means: it is false that every variable in $\text{var}(P_4) \setminus \text{var}(P_1)$ is bounded. But to say "every variable" is not correct in this context, because P_4 contains the optional pattern (?Y,mail,?Z).

In fact, consider the mapping $\mu_1(?X) = b$, $\mu_1(?N) = n_b$, and $\mu_1(?Y) = c$. This mapping does not match $(P_1 \text{ MINUS } P_4)$, because it matches P_4, since it matches (?X,knows,?Y) although it does not match the optional pattern (?Y,mail,?Z). On the other hand, we have that μ_1 matches P' because it matches $(P_1 \text{ OPT } P_4)$ and μ_1 satisfies the filter constraint $\neg(\text{bound}(?Y) \wedge \text{bound}(?Z))$.

The problem is produced by the expression $\text{bound}(?Z)$, because the bounding state of variable $?Z$ introduces noise when testing if pattern P_4 gets matched. In fact, this variable is optional for pattern P_4.

Now, if we consider the condition of "being optional" of variables when transforming P, we have that in this case $?Y$ is the unique non-optional variable occurring in P_4 but not occurring in P_1, i.e., variable $?Y$ works exactly as the test to check if a mapping matching P_1 matches P_4 as well. Hence, instead of P', the graph pattern

$$P'' = ((P_1 \text{ OPT } P_4) \text{ FILTER}(\neg \text{ bound}(?Y))$$

is the one that expresses faithful the graph pattern $(P_1 \text{ MINUS } P_4)$, and in fact, the evaluation of P'' will be exactly the same set of mappings for P.

(Note that this type of patterns –double nested optionals– will arise with double-difference $P_1 \text{ MINUS}(P_2 \text{ MINUS } P_3)$.)

4 Avoiding Unsafe Patterns in SPARQL$_{WG}$

One influential point in the design of the evaluation of patterns in the SPARQL semantics is the behavior of *filters*. What is the scope of a filter? What is the meaning of a filter condition having variables that do not occur in the graph pattern to be filtered?

In [6] it was proposed that, for reasons of simplicity for the user, and cleanness of the semantics, the scope of filters should be the expression which they filter, and free variables should be disallowed in the filter condition. Formally, given a pattern of the form $(P \text{ FILTER } C)$, it is said to be *safe* if $\text{var}(C) \subseteq \text{var}(P)$. In [6] only safe filter expressions were allowed in the syntax, and hence the scope of the filter C is the pattern P which defines the filter expression. The evaluation

thus is the natural one: $\langle\langle P \,\text{FILTER}\, C \rangle\rangle = \{\mu \in \langle\langle P \rangle\rangle \mid \mu \models C\}$. This approach is further supported by the fact that non-safe filter are rare in practice.

The WG decided to follow a different approach, and defined that the scope of a condition C in a filter expression is a case-by-case and context-dependent feature:

1. The scope of a filter is defined as follows: a filter "is a restriction on solutions over the whole group in which the filter appears".
2. There is one exception, though, when filters combine with optionals. If $\text{FILTER}\, C$ belongs to the group graph pattern of an optional, the scope of C is local to the group where the optional belongs to.

The complexities that this approach brings were recognized in the discussion of the WG, and can be witnessed by the reader by following the specification of the WG evaluation of patterns.

In what follows we will show that in the frame of the WG semantics, non-safe filters are superfluous, and hence the above non-standard and case-by-case semantics can be avoided. In fact, we will prove that non-safe filters do not add expressiveness to the language, or in other words, that SPARQL_{WG} and SPARQL^{S}_{WG} have the same expressive power, that is, for each query q there is a filter-safe query q' which computes exactly the same results as q.

The transformation is given by Algorithm 1. This algorithm works as the identity for most patterns. The key part is the treatment of patterns which combine filters and optionals. Lines 8, 9, 10 and 11 are exactly the codification of the WG evaluation of filters inside optionals. For non-safe filters, it replaces the filter condition where the free variable occurs by a logical value of false.

Note 3 (On Algorithm 1). The complex pattern in lines 8, 9, 10 and 11 can be simplified to $T(P) \leftarrow (T(P_1)\,\text{OPT}((T(P_1)\,\text{AND}\,T(P_2))\,\text{FILTER}\,C))$ if one would need only set-semantics. Lines 14 and 15 work similarly for the atomic filter expressions isIRI, isLiteral and isBlank.

Lemma 1. *For every pattern P, the pattern $T(P)$ defined by Algorithm 1 is filter-safe and it holds $\langle\langle P \rangle\rangle = \langle\langle T(P) \rangle\rangle$.*

Proof. We present the proof for the most relevant cases.

Note 4. In this proof we use concepts defined in the SPARQL specification (See Appendix I).

1. If $P = (P_1$ AND $\mid$ UNION $\mid$ OPT $(P_2\,\text{FILTER}\,C))$ and $?X$ is a variable in $\text{var}(C)$ and not in $\text{var}(P_1) \cup \text{var}(P_2)$.
   We consider the following semantics defined in the SPARQL specification [9]:
   - Apart from bound$(\cdot)$, all functions and operators operate on RDF Terms and will produce a type error if any arguments are unbound (Sec. 11.2).
   - Function bound(var) returns true if var is bound to a value. Returns false otherwise (Sec. 11.4.1).
   - Let Ω be a set of solution mappings and expr be an expression.

Algorithm 1 Transformation of patterns to safe patterns

if $P = (P_1 \text{ AND } P_2)$ **then**
  $T(P) \leftarrow (T(P_1) \text{ AND } T(P_2))$
else if $P = P_1 \text{ UNION } P_2$ **then**
  $T(P) \leftarrow (T(P_1) \text{ UNION } T(P_2))$
else if $P = (P_1 \text{ FILTER } C)$ and $\text{var}(C) \subseteq \text{var}(P_1)$ **then**
  $T(P) \leftarrow (T(P_1) \text{ FILTER } C)$
else if $P = (P_1 \text{ OPT}(P_2 \text{ FILTER } C))$ and $\text{var}(C) \subseteq \text{var}(P_1) \cup \text{var}(P_2)$ **then**
  $T(P \leftarrow ((T(P_1) \text{ AND } T(P_2)) \text{ FILTER } C) \text{ UNION}$
    $(T(P_1) \text{ MINUS } T(P_2)) \text{ UNION}$
    $((T(P_1) \text{ MINUS}(T(P_1) \text{ MINUS } T(P_2)))$
      $\text{MINUS}((T(P_1) \text{ AND } T(P_2)) \text{ FILTER } C))$
else if $P = (P_1 \text{ AND} \mid \text{UNION} \mid \text{OPT } (P_2 \text{ FILTER } C)$ **then**
  for all variable $?X$ in $\text{var}(C)$ and not in $\text{var}(P_1) \cup \text{var}(P_2)$ **do**
    for all expression $(?X = a)$ or $(?X = ?Y)$ or $\text{bound}(?X)$ occurring in C **do**
      Replace in C the expression by $false$
    end for
  end for
else if $P = (P1 \text{ OPT } P_2)$ **then**
  $T(P) \leftarrow (T(P1) \text{ OPT } T(P_2))$
end if

$$\text{Filter}(\text{expr}, \Omega) = \{\mu \mid \mu \in \Omega \text{ and } \text{expr}(\mu) \text{ is an expression that has an effective boolean value of true}\} \text{ (Section 12.4)}$$

Given a variable $?X \in \text{var}(C)$ but $?X \notin \text{var}(P_2)$, we have that $?X$ is not bounded for every mapping $\mu \in \langle\langle P_2 \rangle\rangle$, then $\langle\langle P_2 \text{ FILTER } C \rangle\rangle = \emptyset$. To attain the same result, we replace C by the logical value of false.

2. Let P be the pattern $(P_1 \text{ OPT}(P_2 \text{ FILTER } C))$ and P' be the pattern:
   $((P_1 \text{ AND } P_2) \text{ FILTER } C))$ UNION $(P_1 \text{ MINUS } P_2)$ UNION
   $((P_1 \text{ MINUS}(P_1 \text{ MINUS } P_2)) \text{ MINUS}((P_1 \text{ AND } P_2) \text{ FILTER } C))$
   We need to prove that for every dataset D with active graph G, it satisfies that $\langle\langle P \rangle\rangle = \langle\langle P' \rangle\rangle$.

   [Evaluation of P] .
   Transforming P in a SPARQL algebra expression:
   $\text{Transform}(P) = \text{LeftJoin}(P_1, P_2, \text{expr})$
   Suppose that $\Omega_1 = \text{eval}(D(G), P_1)$ and $\Omega_2 = \text{eval}(D(G), P_2)$. The evaluation of $\text{Transform}(P)$ over dataset D with active graph G is defined as

$$\text{eval}(D(G), \text{Filter}(\text{expr}, \text{Join}(P_1, P_2))) \ \cup \ \text{eval}(D(G), \text{Diff}(P_1, P_2, \text{expr})).$$

   where:
   - $\text{eval}(D(G), \text{Filter}(\text{expr}, \text{Join}(P_1, P_2)))$ defines the set:
     $\{\text{merge}(\mu_1, \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are compatible, and } \text{expr}(\text{merge}(\mu_1, \mu_2)) \text{ is true}\}$
   - $\text{eval}(D(G), \text{Diff}(P_1, P_2, \text{expr}))$ defines the union of sets:
     $\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\} \cup$
     $\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2 \text{ compatible with } \mu_1, \text{expr}(\text{merge}(\mu_1, \mu_2)) \text{ is false}\}$

[Evaluation of P'] .
The SPARQL algebra expression for P' will be (to simplify the expression we represent the operator Union by the symbol $\cup$):

$\text{Filter}(\text{expr}, \text{Join}(P_1, P_2)) \ \cup \ \text{Diff}(P_1, P_2, true) \cup$
$\text{Diff}(\text{Diff}(P_1, \text{Diff}(P_1, P_2, true), true), \text{Filter}(\text{expr}, \text{Join}(P_1, P_2)), true)^{(\star)}$

The evaluation of the above expression over dataset D with active graph G is the union of the following evaluations:

- $\text{eval}(D(G), \text{Filter}(\text{expr}, \text{Join}(P_1, P_2)))$ which defines the set
  $\{\text{merge}(\mu_1, \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1$ and μ_2 are compatible, and $\text{expr}(\text{merge}(\mu_1, \mu_2))$ is true$\}$
- $\text{eval}(D(G), \text{Diff}(P_1, P_2, true))$ which defines the union of sets
  $\{\mu_1 \in \Omega_1 \mid$ for all $\mu_2 \in \Omega_2, \mu_1$ and μ_2 are not compatible$\} \cup$
  $\{\mu_1 \in \Omega_1 \mid$ for all $\mu_2 \in \Omega_2$ compatible with μ_1, $\text{expr}(\text{merge}(\mu_1, \mu_2))$ is false$\}$
  Note that second set is empty because the condition expr is true.
- $\text{eval}(D(G), \star)$ which defines the set
  $\{\mu_1 \in \Omega_1 \mid$ for all $\mu_2 \in \Omega_2$ compatible with μ_1, $\text{expr}(\text{merge}(\mu_1, \mu_2))$ is false$\}$
  Let M_1 be the set of mappings Ω_1, M_2 be the subset of mappings in Ω_1 that are incompatible with some mapping in Ω_2, M_3 be the subset of mappings in Ω_1 that are compatible with some mapping in Ω_2 and that satisfies the filter condition C, and M be the subset of mappings in Ω_1 that are compatible with some mapping of Ω_2 and that does not satisfy the filter condition C.
  It is clear that $M = (M_1 - M_2) - (M_3)$.
  If we consider that M_2 is $\text{Diff}(P_1, P_2, true)$ and M_3 is $\text{Filter}(C, \text{Join}(P_1, P_2))$.
  Then $\text{Diff}(\text{Diff}(P_1, \text{Diff}(P_1, P_2, true), true), \text{Filter}(C, \text{Join}(P_1, P_2)), true)$ represents the set of mapping M.

Note that the sets that conforms the evaluation of P are the same in the evaluation of P'. Then, patterns P and P' are equivalent.

Thus we proved:

Theorem 2. *$SPARQL_{WG}$ and $SPARQL^{S}_{WG}$ have the same expressive power.*

5 Expressive power of SPARQL_{WG} is equivalent to SPARQL_{C}

As we have been showing, the semantics that the WG gave to SPARQL departed in some aspects from a compositional semantics. We also indicated that there is an alternative formalization, with a standard algebraic syntax and a compositional semantics, which was called SPARQL_C [6].

The good news is that, albeit these apparent differences, these languages are equivalent in expressive power, that is, although differing in the syntax and semantics, they compute the same class of queries. In fact, the only differences are produced by the treatment of non-safe filters in patterns and the evaluation of filters inside optional. For this reason and because SPARQL^{S}_{WG} and SPARQL_C do not have non-safe filters, the following result can be proved.

Theorem 3. *$SPARQL^S_{WG}$ is equivalent to $SPARQL_C$ under bag semantics.*

Proof. We have to show that for each safe-filter graph pattern P, it holds that for the corresponding pattern P' in SPARQL_C (recall, the syntactic translation) $[\![P']\!] = \langle\!\langle P \rangle\!\rangle$. This is an induction on the structure of patterns.

By comparing both evaluations, it is easy to see that the only non-evident case where the semantics of SPARQL^S_{WG} and SPARQL_C differ, is a particular evaluation of filters inside optionals, specifically: given a graph pattern $P = (P_1 \,\text{OPT}(P_2 \,\text{FILTER}\, C))$, we have that SPARQL^S_{WG} evaluates the algebra expression $\text{LeftJoin}(P_1, P_2, C)$, whereas SPARQL_C evaluates $[\![P]\!]$ to the expression $[\![P_1]\!] \overset{\cdot}{\bowtie} [\![P_2 \,\text{FILTER}\, C]\!]$, which clearly is the same as $\text{LeftJoin}(P_1, \text{Filter}(C, P_2), \mathit{true})$ in the SPARQL_{WG} formalism.

Next, we show in detail that for a pattern $P = (P_1 \,\text{OPT}(P_2 \,\text{FILTER}\, C))$ where $\text{var}(C) \subseteq \text{var}(P_2)$ (i.e., P is filter safe) it satisfies that $\langle\!\langle P \rangle\!\rangle^D_G = [\![P]\!]^D_G$, for every dataset D with active graph G.

[Evaluation in sparql^s_{WG}*]* .
Following the SPARQL specification, we transform P in the algebra expression:

$$\text{Transform}(P) = \text{LeftJoin}(P_1, P_2, \text{expr}).$$

Suppose that $\Omega_1 = \text{eval}(D(G), P_1)$ and $\Omega_2 = \text{eval}(D(G), P_2)$. The evaluation of $\text{Transform}(P)$ over dataset D with active graph G is defined as

$$\text{eval}(D(G), \text{Filter}(\text{expr}, \text{Join}(P_1, P_2))) \;\cup\; \text{eval}(D(G), \text{Diff}(P_1, P_2, \text{expr}))$$

where:

- $\text{eval}(D(G), \text{Filter}(\text{expr}, \text{Join}(P_1, P_2)))$ which defines the set
  - (a) $\{\text{merge}(\mu_1, \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1$ and μ_2 are compatible, and $\text{expr}(\text{merge}(\mu_1, \mu_2))$ is true$\}$
- $\text{eval}(D(G), \text{Diff}(P_1, P_2, \text{expr}))$ which defines the union of the sets
  - (b) $\{\mu_1 \in \Omega_1 \mid$ for all $\mu_2 \in \Omega_2, \mu_1$ and μ_2 are not compatible$\}$
  - (c) $\{\mu_1 \in \Omega_1 \mid$ for all $\mu_2 \in \Omega_2$ compatible with μ_1, $\text{expr}(\text{merge}(\mu_1, \mu_2))$ is false$\}$

  If we redefine (b) by changing the universal quantifier by an existential quantifier: $\{\mu_1 \in \Omega_1 \mid \nexists \mu_2 \in \Omega_2$ satisfying that μ_1 and μ_2 are compatible $\}$.
  Now, we have two cases: ($\diamond$) if $\Omega_2 = \emptyset$, then there exists no $\mu_2 \in \Omega_2$; ($\diamond\diamond$) if $\Omega_2 \neq \emptyset$, then for all $\mu_2 \in \Omega_2$, μ_1 is not compatible with μ_2 and either $\mu_2 \models C$ or $\mu_2 \not\models C$. Then, we have that (b) encodes three cases:
  (b_1) $\{\mu_1 \mid \mu_1 \in \Omega_1, \nexists \mu_2 \in \Omega_2\}$
  (b_2) $\{\mu_1 \mid \mu_1 \in \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1$ is not compatible with $\mu_2, \mu_2 \models C\}$
  (b_3) $\{\mu_1 \mid \mu_1 \in \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1$ is not compatible with $\mu_2, \mu_2 \not\models C\}$

[Evaluation in sparql_C*]* .
The evaluation of P in SPARQL_C is given by the expression

$$[\![P_1]\!]^D_G \overset{\cdot}{\bowtie} [\![P_2 \,\text{FILTER}\, C]\!]^D_G.$$

Suppose that $\Omega_1 = [\![P_1]\!]_G^D$, $\Omega_2 = [\![P_2]\!]_G^D$, and $\Omega_3 = \{\mu_2 \in \Omega_2 \mid \mu_2 \models C\}$, then $[\![P]\!]_G^D$ is given by the set-union of two sets:

(1) $\{\mu_1 \cup \mu_3 \mid \mu_1 \in \Omega_1, \mu_3 \in \Omega_3, \mu_1 \text{ and } \mu_3 \text{ are compatible}\}$
(2) $\{\mu_1 \in \Omega_1 \mid \text{for all } \mu_3 \in \Omega_3, \mu_1 \text{ and } \mu_3 \text{ are not compatible}\}$

If we redefine (1) by solving μ_3, we have that:
$\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are compatible, and } \mu_2 \models C\}$
Considering that μ_1 is compatible with μ_2 we can write
(1.1) $\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are compatible, and } (\mu_1 \cup \mu_2) \models C\}$
If we redefine (2) by changing the universal quantifier by an existential quantifier: $\{\mu_1 \mid \mu_1 \in \Omega_1, \nexists \mu_3 \in \Omega_3, \mu_1 \text{ and } \mu_3 \text{ are compatible}\}$.
Now, we have two cases:
(i) When $\Omega_3 = \emptyset$. It implies two cases: ($\star$) $\nexists \mu_2 \in \Omega_2$; and ($\star\star$) $\forall \mu_2 \in \Omega_2$, $\mu_2 \not\models C$ (and either μ_1 is compatible with μ_2 or μ_1 is not compatible with μ_2).
(ii) When $\Omega_3 \neq \emptyset$. Then $\forall \mu_2 \in \Omega_2$, $\mu_2 \models C$, μ_1 and μ_2 are not compatible.
Considering ($\star$), ($\star\star$), and (ii), we have that (2) encodes four cases:
(2.1) $\{\mu_1 \mid \mu_1 \in \Omega_1, \nexists \mu_2 \in \Omega_2\}$
(2.2) $\{\mu_1 \mid \mu_1 \in \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are compatible, and } \mu_2 \not\models C\}$
(2.3) $\{\mu_1 \mid \mu_1 \in \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible, and } \mu_2 \models C\}$
(2.4) $\{\mu_1 \mid \mu_1 \in \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible, and } \mu_2 \not\models C\}$
Considering that μ_1 is compatible with μ_2 we redefine (2.2) into
(2.2.1) $\{\mu_1 \mid \mu_1 \in \Omega_1, \forall \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are compatible, and } (\mu_1 \cup \mu_2) \not\models C\}$

Finally, we have that (1.1) is (a), (2.1) is (b_1), (2.3) is (b_2), (2.4) is (b_3), and (2.2.1) is (c). Then, we have proven the claim.

6 Expressive Power of SPARQL$_C$

In this section we study the expressive power of SPARQL$_C$ by comparing it against non recursive safe Datalog with negation (just Datalog from now on). (For Datalog details see Appendix III). Informally, Datalog facts correspond to RDF graphs, Datalog rules correspond to graph patterns, goal queries correspond to SELECT clauses, and the set of substitutions returned by a Datalog query corresponds to the set of mappings returned by a SPARQL$_C$ query.

6.1 From SPARQL$_C$ to Datalog

Note that because SPARQL$_C$ and Datalog programs have different type of input and output formats, we have to normalize them to be able to do the comparison.

Let $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$ and $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$ be the SPARQL$_C$ language and the Datalog language respectively. To prove that L_s is contained in L_d, we define transformations $\mathcal{T}_Q : \mathcal{Q}_s \to \mathcal{Q}_d$, $\mathcal{T}_D : \mathcal{D}_s \to \mathcal{D}_d$, and $\mathcal{T}_S : \mathcal{S}_s \to \mathcal{S}_d$. That is, $\mathcal{T}_Q$ transforms a SPARQL$_C$ query into a Datalog query, $\mathcal{T}_D$ transforms a SPARQL$_C$ dataset into a set of Datalog facts, and $\mathcal{T}_S$ transforms a set of SPARQL$_C$ solution mappings into a set of Datalog substitutions.

The rough idea of the transformations is the following. For transforming RDF Datasets, a function $\mathcal{T}_D$ is defined, which uses the unary predicates *iri*, *blank*, and *literal* to encode IRIs, blank nodes, and literals respectively. The unary predicate $Null$ encodes the *null* value [7]. The IRI of each named graph is encoded using the predicate *graph*. A fact $triple(u, v_1, v_2, v_3)$ models a triple (v_1, v_2, v_3) which occurs in the graph identified by IRI u. The unary predicate *term* encodes the domain of values.

With respect to solutions, note that naturally sets of mappings in SPARQL correspond bijectively to sets of substitutions in Datalog.

RDF data as Datalog facts . Let $D = \{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ be an RDF dataset. We denote by $\mathcal{T}_D(D)$ the function which transforms a dataset D into a set of Datalog facts. The function $\mathcal{T}_D$ is defined by the following algorithm:

$F \leftarrow \emptyset$
for each $u \in \text{term}(D) \cap I$ **do**
  add the fact $iri(u)$ to F
end for
for each $b \in \text{term}(D) \cap B$ **do**
  add the fact $blank(b)$ to F
end for
for each $l \in \text{term}(D) \cap L$ **do**
  add the fact $literal(l)$ to F
end for
Add the Datalog fact $Null(null)$ to F
for each triple $(v_1, v_2, v_3) \in G_0$ **do**
  Add the fact $triple(g_0, v_1, v_2, v_3)$ to F
end for
for each named graph $\langle u, G\rangle \in D$ **do**
  Add the fact $graph(u)$ to F
  for each triple $(v_1, v_2, v_3) \in G$ **do**
    Add the fact $triple(u, v_1, v_2, v_3)$ to F
  end for
end for
Add to F the rules
  $term(X) \leftarrow iri(X)$
  $term(X) \leftarrow blank(X)$
  $term(X) \leftarrow literal(X)$
return F

SPARQL$_C$ solutions (solution mappings) as Datalog solutions (substitutions) . Given a mapping μ, define the notion of *extended domain* of μ, denoted $\text{extdom}(\mu)$, as a finite set of variables containing its domain. Given a set of mappings Ω, the set of substitutions obtained from Ω, denoted $\mathcal{T}_S(\Omega)$, is

[7] We use null values to represent unbounded values.

defined as follows: for each mapping $\mu \in \Omega$ there exists a substitution $\theta \in \mathcal{T}_S(\Omega)$ satisfying that, for each $x \in \text{extdom}(\mu)$ there exists $x/t \in \theta$ such that $t = \mu(x)$ when $\mu(x)$ is bounded and $t = null$ otherwise.

Graph patterns as Datalog rules. Let P be a graph pattern to be evaluated against a dataset D with active graph identified by g. We denote by $\delta(P, g)_D$ (or simply $\delta(P, g)$ when D is clear from the context) the function which transforms the graph pattern P into a set of Datalog facts. A predicate *comp* implements the notion of compatible mappings: $comp(X, X, X) \leftarrow term(X)$, $comp(X, null, X) \leftarrow \text{term}(X)$, $comp(null, X, X) \leftarrow \text{term}(X)$, and $comp(X, X, X) \leftarrow Null(X)$.

The transformation follows essentially the intuitive transformation presented by Polleres [8] with the improvement of the necessary code to support faithful translation of bag semantics. Specifically we changed the transformations for complex filter expressions by simulating them with double negation.

The complete rules work as follows. Let p, p_1, p_2 be predicate names for graph patterns P, P_1 and P_2 respectively. Additionally, we denote by $\overline{\text{var}}(P)$, a tuple of variables obtained from a lexicographical ordering of the variables in the graph pattern P. Then, the function $\delta(P, g)_D$ is defined recursively as follows:

(1) If P is a triple pattern (x_1, x_2, x_3), then $\delta(P, g)_D$ is:
$p(\overline{\text{var}}(P)) \leftarrow triple(g, x_1, x_2, x_3)$

(2) If P is $(P_1 \,\text{AND}\, P_2)$, then $\delta(P, g)_D$ is:
$\delta(P_1, g)_D$
$\delta(P_2, g)_D$
$p(\overline{\text{var}}(P)) \leftarrow \nu_1(p_1(\overline{\text{var}}(P_1))) \ \wedge \ \nu_2(p_2(\overline{\text{var}}(P_2)))$
$\bigwedge_{x \in \text{var}(P_1) \cap \text{var}(P_2)} comp(\nu_1(x), \nu_2(x), x)$
where $\nu_j : V \rightarrow V$ is a variable-renaming function, and $\nu_j(L)$ renames the variables in the literal L according to ν_j. Additionally, it satisfies that $\text{dom}(\nu_1) = \text{dom}(\nu_2) = \text{var}(P_1) \cap \text{var}(P_2)$ and $\text{range}(\nu_1) \cap \text{range}(\nu_2) = \emptyset$.

(3) If P is $(P_1 \,\text{UNION}\, P_2)$, then $\delta(P, g)_D$ is:
$\delta(P_1, g)_D$
$\delta(P_2, g)_D$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x)$
$p(\overline{\text{var}}(P)) \leftarrow p_2(\overline{\text{var}}(P_2)) \bigwedge_{x \in \text{var}(P_1) \wedge x \notin \text{var}(P_2)} Null(x)$

(4) If P is $(P_1 \,\text{OPT}\, P_2)$.
Let $P_3 = (P_1 \,\text{AND}\, P_2)$. Then $\delta(P, g)_D$ is:
$\delta(P_1, g)_D$
$\delta(P_2, g)_D$
$\delta(P_3, g)_D$
$p'_1(\overline{\text{var}}(P_1)) \leftarrow p_3(\overline{\text{var}}(P_3))$
$p(\overline{\text{var}}(P)) \leftarrow p_3(\overline{\text{var}}(P_3))$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'_1(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x)$

(5) If P is $(n \,\text{GRAPH}\, P_1)$

(5.1) When $n \in I$ then $\delta(P, g)_D$ is
$\delta(P_1, n)_D$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1))$

(5.2) When $n \in V$ then $\delta(P, g)_D$ is
$\delta(P_{11}, u_1)_D$
$p(\overline{\text{var}}(P)) \leftarrow p_{11}(\overline{\text{var}}(P_{11})) \wedge graph(n) \wedge n = u_1$
$\vdots$
$\delta(P_{1n}, u_n)_D$
$p(\overline{\text{var}}(P)) \leftarrow p_{1n}(\overline{\text{var}}(P_{1n})) \wedge graph(n) \wedge n = u_n$
where $u_i \in \text{names}(D)$, each P_{1i} is a copy of P_1, and p_{1i} is its respective predicate.

(6) If P is $(P_1 \,\text{FILTER}\, C)$ and

(6.1) C is an atomic filter constraint, then $\delta(P, g)_D$ is:
$\delta(P_1, g)_D$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge cond$
where *cond* is defined as follows. If $?X, ?Y \in V$ and $u \in I \cup L$ then, *cond* is C when C is either (?X = u) or (?X=?Y); *cond* is $iri(?X)$ when C is (isIRI($?X$)); *cond* is $literal(?X)$ when C is (isLiteral($?X$)); *cond* is $blank(?X)$ when C is (isBlank($?X$)).

(6.2) C is $(\neg C_1)$, then $\delta(P, g)_D$ is:
$\delta(P_1, g)_D$
$\delta(P'_1, g)_D$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'_1(\overline{\text{var}}(P_1))$
where $P'_1 = (P_1 \,\text{FILTER}\, C_1)$

(6.3) C is $(C_1 \wedge C_2)$, then $\delta(P, g)_D$ is:
$\delta(P_{11}, g)_D$
$\delta(P_{12}, g)_D$
$p''(\overline{\text{var}}(P_1)) \leftarrow p_{11}(\overline{\text{var}}(P_1)) \wedge p_{12}(\overline{\text{var}}(P_1))$
$p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1))$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))$
where $P_{11} = (P_1 \,\text{FILTER}\, C_1)$ and $P_{12} = (P_1 \,\text{FILTER}\, C_2)$

(6.4) C is $(C_1 \vee C_2)$, then $\delta(P, g)_D$ is:
$\delta(P_{11}, g)_D$
$\delta(P_{12}, g)_D$
$p''(\overline{\text{var}}(P_1)) \leftarrow p_{11}(\overline{\text{var}}(P_1))$
$p''(\overline{\text{var}}(P_1)) \leftarrow p_{12}(\overline{\text{var}}(P_1))$
$p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1))$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))$
where $P_{11} = (P_1 \,\text{FILTER}\, C_1)$ and $P_{12} = (P_1 \,\text{FILTER}\, C_2)$

SPARQL$_C$ queries as Datalog queries. Let $Q_s = (\text{SELECT}\, W, F, P)$ be a SPARQL$_C$ query and D be the dataset obtained from F. The function $\mathcal{T}_Q(Q_s)$ returns the Datalog query $(\Pi, q(\overline{W}))$ where $\overline{W}$ is a tuple of variables obtained from a lexicographically ordering of the variables in W, and Π is the Datalog program

$\mathcal{T}_D(D)$
$\delta(P, g_0)_D$
$q(\overline{W}) \leftarrow p(\overline{\text{var}}(P))$

where q and p are predicate names for Q_s and P respectively, and g_0 identifies the default graph in D.

The following theorem states that the transformation works well.

Theorem 4. *SPARQL$_C$ is contained in non-recursive safe Datalog with negation.*

Proof. Consider transformations $\mathcal{T}_Q$, $\mathcal{T}_D$, and $\mathcal{T}_S$ defined above. We have to prove that for every query $Q_s = (\text{SELECT}\, W, F, P)$ and dataset $D = \text{dataset}(F)$, it satisfies that $\mathcal{T}_S(\text{ans}_s(Q_s, D)) = \text{ans}_d(\mathcal{T}_Q(Q_s), \mathcal{T}_D(D))$.

We have that $\mathcal{T}_Q(Q_s)$ is the Datalog query $(\Pi, q(\overline{W}))$ where Π is given by rules $\mathcal{T}_D(D)$, $\delta(P, g_0)$, and $q(\overline{\text{var}}(W)) \leftarrow p(\overline{\text{var}}(P))$. A substitution θ is in $\text{ans}_d(\mathcal{T}_Q(Q_s), \mathcal{T}_D(D))$ if it satisfies that $\theta(q(\overline{\text{var}}(W))) \in facts^*(\Pi)$. The latter applies if there exists $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g_0))$.

On the other hand, a mapping μ is in $\text{ans}_s(Q_s, D)$ if and only if $\mu = \mu'_{|\,\text{var}(W)}$ and $\mu' \in [\![P]\!]^D_{\text{dg}(D)}$. Then, for the graph pattern P, we need to prove that μ' is in $[\![P]\!]^D_{\text{dg}(D)}$ if and only if substitution $\theta = \mathcal{T}_S(\mu')$ satisfies $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g_0))$.

Let P be a graph pattern, D be a dataset, and G be the active graph of D identified by IRI g. We will show that for each mapping $\mu \in [\![P]\!]^D_G$ there exists a substitution $\theta = \mathcal{T}_S(\mu)$ such that $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$. The proof is by induction on the structure of P.

(1) Base case: P is a triple pattern (x_1, x_2, x_3).
We have that $\delta(P, g)$ is the rule $p(\overline{\text{var}}(P)) \leftarrow triple(g, x_1, x_2, x_3)$.
Given a substitution θ, it satisfies that $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if there is fact $\theta(triple(g, x_1, x_2, x_3)) \in \mathcal{T}_D(D)$.
On the other hand, a mapping μ is in $[\![P]\!]^D_G$ if and only if $\text{dom}(\mu) = \text{var}(P)$ and $\mu((x_1, x_2, x_3)) = (v_1, v_2, v_3) \in G$. Then $\mu(x_i) = v_i$ when $x_i \in \text{var}(P)$. If we transform μ into a substitution, that is $\mathcal{T}_S(\mu) = \{x_i/v_i \mid x_i \in \text{var}(P)\}$. Then $\theta = \mathcal{T}_S(\mu)$ and we are done.

Inductive case: Let P_1 and P_2 be patterns. We need to consider several cases:

(2) P is $(P_1 \,\text{AND}\, P_2)$.
We have that $\delta(P, g)$ is the set of rules

{ $\delta(P_1, g)$, $\delta(P_2, g)$,
$p(\overline{\text{var}}(P)) \leftarrow \nu_1(p_1(\overline{\text{var}}(P_1))) \wedge \nu_2(p_2(\overline{\text{var}}(P_2)))$,
$comp(\nu_1(x_1), \nu_2(x_1), x_1)$, ..., $comp(\nu_1(x_n), \nu_2(x_n), x_n)$ }
where $x_i \in \text{var}(P_1) \cap \text{var}(P_2)$.
Given a substitution θ, a fact $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if $\theta(\nu_1(p_1(\overline{\text{var}}(P_1)))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$, $\theta(\nu_2(p_2(\overline{\text{var}}(P_2)))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_2, g))$, and for each $x_i \in \text{var}(p_1) \cap \text{var}(P_2)$ we have that $\theta(comp(\nu_1(x_i), \nu_2(x_i), x_i)) \in facts^*(\mathcal{T}_D(D))$ i.e., $\theta(x_i) = \theta(\nu_1(x_i)) = \theta(\nu_2(x_i))$, or $\theta(\nu_1(x_i)) = null$ and $\theta(x_i) = \theta(\nu_1(x_i))$, or $\theta(\nu_2(x_i)) = null$ and $\theta(x_i) = \theta(\nu_1(x_i))$, or $\theta(x_i) = \theta(\nu_1(x_i)) = \theta(\nu_2(x_i)) = null$.
On the other hand, a mapping μ is in $[\![(P_1 \text{ AND } P_2)]\!]^D_G$ if and only if $\mu = \mu_1 \cup \mu_2$ such that $\mu_1 \in [\![P_1]\!]^D_G$, $\mu_2 \in [\![P_2]\!]^D_G$, and μ_1 is compatible with μ_2 i.e, for each $x \in \text{var}(P_1) \cap \text{var}(P_2)$ it applies that either $\mu_1(x) = \mu_2(x)$ or $\mu_1(x)$ is unbounded or $\mu_2(x)$ is unbounded. For induction hypothesis, we have substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$, $\theta_2 = \mathcal{T}_S(\mu_2)$ such that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$, $\theta_2(p_2(\overline{\text{var}}(P_2))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_2, g))$, and for each $x \in \text{var}(P_1) \cap \text{var}(P_2)$ we have that either $\theta_1(x) = \theta_2(x)$, or $\theta_1(x)$ is null, or $\theta_2(x)$ is null. Considering that $\mathcal{T}_S(\mu) = \theta_1 \cup \theta_2$ we have that $\theta = \mathcal{T}_S(\mu)$ and we are done.

(3) If P is $(P_1 \text{ UNION } P_2)$.
We have that $\delta(P, g)$ is the set of rules
{ $\delta(P_1, g)$, $\delta(P_2, g)$,
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x)$,
$p(\overline{\text{var}}(P)) \leftarrow p_2(\overline{\text{var}}(P_2)) \bigwedge_{x \in \text{var}(P_1) \wedge x \notin \text{var}(P_2)} Null(x)$ }
Given a substitution θ, it satisfies that $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if either (a) $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and x is null for each $x \in \text{var}(P_1) \backslash \text{var}(P_2)$; or (b) $\theta(p_2(\overline{\text{var}}(P_2))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_2, g))$ and x is null for each $x \in \text{var}(P_2) \setminus \text{var}(P_1)$. In case (a), we have that $\theta = \{x/v \mid x \in \text{var}(P_1)\} \cup \{x/null \mid x \notin \text{var}(P_1)\}$. In case (b), we have that $\theta = \{x/v \mid x \in \text{var}(P_2)\} \cup \{x/null \mid x \notin \text{var}(P_2)\}$.
On the other hand, a mapping μ is in $[\![(P_1 \text{ UNION } P_2)]\!]^D_G$ if and only if either (a) $\mu = \mu_1 \in [\![P_1]\!]^D_G$ or (b) $\mu = \mu_2 \in [\![P_2]\!]^D_G$. For induction hypothesis, we have that there exist substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$, $\theta_2 = \mathcal{T}_S(\mu_2)$ satisfying that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and $\theta_2(p_2(\overline{\text{var}}(P_2))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_2, g))$. Assuming that $\theta_1 = \{x/v \mid x \in \text{var}(P_1)\}$ and $\theta_2 = \{x/v \mid x \in \text{var}(P_2)\}$. In case (a), $\mu(x)$ is unbounded for each $x \in \text{var}(P_2) \setminus \text{var}(P_1)$, that is $\{x/null \mid x \notin \text{var}(P_1)\}$, then $\mathcal{T}_S(\mu) = \theta_1 \cup \{x/null \mid x \notin \text{var}(P_1)\}$. In case (b), $\mu(x)$ is unbounded for each $x \in \text{var}(P_1) \setminus \text{var}(P_2)$, that is $\{x/null \mid x \notin \text{var}(P_2)\}$, then $\mathcal{T}_S(\mu) = \theta_2 \cup \{x/null \mid x \notin \text{var}(P_2)\}$. We have that $\theta = \mathcal{T}_S(\mu)$ and we are done.

(4) P is $(P_1 \text{ OPT } P_2)$.
Considering that $P_3 = (P_1 \text{ AND } P_2)$, we have that $\delta(P, g)$ is the set of rules
{ $\delta(P_1, g)$, $\delta(P_2, g)$, $\delta(P_3, g)$,
$p'_1(\overline{\text{var}}(P_1)) \leftarrow p_3(\overline{\text{var}}(P_3))$,
$p(\overline{\text{var}}(P)) \leftarrow p_3(\overline{\text{var}}(P_3))$,
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'_1(\overline{\text{var}}(P_1)) \bigwedge_{x \in \text{var}(P_2) \wedge x \notin \text{var}(P_1)} Null(x)$}.

Given a substitution θ, we have that $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if one of two cases applies:
(i) $\theta(p_3(\overline{\text{var}}(P_3))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_3, g))$; that is, it satisfies that $\theta(p_3(\overline{\text{var}}((P_1 \text{ AND } P_2)))) \in facts^*(\mathcal{T}_D(D) \cup \delta((P_1 \text{ AND } P_2), g))$.
(ii) $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$; that is, it is false that $\theta(p'_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P'_1, g))$ and x is null for each variable $x \in \text{var}(P_2) \setminus \text{var}(P_1)$.
Additionally, $\theta(p'_1(\overline{\text{var}}(P_1))) \notin facts^*(\mathcal{T}_D(D) \cup \delta(P'_1, g))$ iff case (i) is false. Then, θ satisfies that: $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$; for each $\theta_2(p_2(\overline{\text{var}}(P_2))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_2, g))$, $\theta(x) \neq \theta_2(x)$ for all variable $x \in \text{var}(P_1) \cap \text{var}(P_2)$; and $x/null \in \theta$ for each variable $x \in \text{var}(P_2) \setminus \text{var}(P_1)$.
On the other hand, a mapping μ is in $[\![(P_1 \text{ OPT } P_2)]\!]^D_G$ if and only if one of two cases applies:
(i) μ is in $[\![P_3]\!]^D_G$ where $P_3 = (P_1 \text{ AND } P_2)$. From (2) we have a substitution $\theta' = \mathcal{T}_S(\mu)$ satisfying that $\theta'(p_3(\overline{\text{var}}(P_3))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_3, g))$.
(ii) $\mu = \mu_1 \in [\![P_1]\!]^D_G$ such that for all $\mu_2 \in [\![P_2]\!]^D_G$ it satisfies that μ_1 and μ_2 are not compatible. Additionally, $\mu(x)$ is unbounded for each $x \in \text{var}(P_2) \setminus \text{var}(P_1)$. For induction hypothesis, we have substitutions $\theta_1 = \mathcal{T}_S(\mu_1)$ and $\theta_2 = \mathcal{T}_S(\mu_2)$ satisfying that $\theta_1(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and $\theta_2(p_2(\overline{\text{var}}(P_2))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_2, g))$. Suppose that $\theta' = \mathcal{T}_S(\mu)$. Following definition of μ, we have that: $\theta' = \theta_1$; for each $\theta_2(p_2(\overline{\text{var}}(P_2))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_2, g))$, $\theta'(x) \neq \theta_2(x)$ for all variable $x \in \text{var}(P_1) \cap \text{var}(P_2)$; and $x/null \in \theta'$ for each variable $x \in \text{var}(P_2) \setminus \text{var}(P_1)$. Then $\theta = \theta' = \mathcal{T}_S(\mu)$ for cases (i) and (ii), and we are done.

(5) P is $(n \text{ GRAPH } P_1)$.
We consider two cases:

(5.1) When $n \in I$.
We have that $\delta(P, g)$ is the set of rules
$\{\delta(P_1, n),\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1))\}$
Given a substitution θ, we have that $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, n))$. On the other hand, a mapping μ is in $[\![P]\!]^D_G$ if and only if $\mu \in [\![P_1]\!]^D_{G'}$ such that $G' = \text{graph}(n)$. In both cases, the active graph identified g has been changed by the graph identified n. Then by induction hypothesis we have that $\theta = \mathcal{T}_S(\mu)$.

(5.2) When $n \in V$.
For each named graph identified u_i in dataset D, we have the set of rules
$\{\delta(P_{1i}, u_i),\ p(\overline{\text{var}}(P)) \leftarrow p_{1i}(\overline{\text{var}}(P_{1i})) \wedge graph(n) \wedge n = u_i\}$.
Considering that P_{1i} is a copy of P_1 and using result (5.1) we can prove that $p(\overline{\text{var}}(P)) \leftarrow p_{1i}(\overline{\text{var}}(P_{1i}))$ is correct. Additionally, given that $\text{var}(P)$ is $n \cup \text{var}(P_{1i})$, we use predicate $graph(n)$ and condition $n = u_i$ to assign the respective IRI u_i to variable n, then we are changing the active graph to the graph identified by u_i. As conclusion, a substitution θ is in $\delta(P, g)$ if θ is a substitution for a some $\delta(P_{1i}, u_i)$ where $u_i \in \text{names}(D)$.

(6) If P is $(P_1 \text{ FILTER } C)$. By induction on the structure of C.

(6.1) Base case: C is an atomic filter constraint.
We have that $\delta(P, g)$ is the set of rules
$\{\ \delta(P_1, g),\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge cond\ \}$.
Given a substitution θ, we have that $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and $\theta(cond)$ is true.
On the other hand, a mapping μ is in $[\![P]\!]^D_G$ if and only if $\mu \in [\![P_1]\!]^D_G$ and μ satisfies C. By induction hypothesis (1) and considering that *cond* is a Datalog literal equivalent to filter constraint C; it applies that there exists substitution $\theta = \mathcal{T}_S(\mu)$ satisfying that $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and $\theta(cond)$ is true. We have proved the base case.

Inductive case:

(6.2) C is $(\neg C_1)$.
We have that $\delta(P, g)$ is the set of rules
$\{\ \delta(P_1, g),\ \delta(P'_1, g),\ p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'_1(\overline{\text{var}}(P_1))\ \}$
where $P'_1 = (P_1 \text{ FILTER } C_1)$.
Given a substitution θ, it satisfies that $\theta(p(\overline{\text{var}}(P))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P, g))$ if and only if $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and $\theta(p'_1(\overline{\text{var}}(P'_1))) \notin facts^*(\mathcal{T}_D(D) \cup \delta(P'_1, g))$. If $cond_1$ is the Datalog literal equivalent to C_1, we have that θ satisfy that $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and $\theta(cond_1)$ is not true;
On the other hand, we have that a mapping μ is in $[\![P]\!]^D_G$ if and only if $\mu \in [\![(P_1 \text{ FILTER } C_1)]\!]^D_G$ and it is not true that μ satisfies C_1. By induction hypothesis $\theta = \mathcal{T}_S(\mu)$ is a substitution which satisfies that $\theta(p_1(\overline{\text{var}}(P_1))) \in facts^*(\mathcal{T}_D(D) \cup \delta(P_1, g))$ and $\theta(cond_1)$ is not true; Then we have proved the case.

(6.3) C is $(C_1 \wedge C_2)$. We have that $\delta(P, g)$ is the set of rules
$\{\ \delta(P_{11}, g)_D,\ \delta(P_{12}, g)_D,$
$p''(\overline{\text{var}}(P_1)) \leftarrow p_{11}(\overline{\text{var}}(P_1)) \wedge p_{12}(\overline{\text{var}}(P_1)),$
$p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1)),$
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))\ \}$
where $P_{11} = (P_1 \text{ FILTER } C_1)$ and $P_{12} = (P_1 \text{ FILTER } C_2)$.
Note that the graph pattern $(P_1 \text{ FILTER}(C_1 \wedge C_2))$ can be rewritten as $((P_1 \text{ FILTER } C_2) \text{ AND}(P_1 \text{ FILTER } C_2))$ (it is showed in the rule for predicate p'' and by the patterns P_{11} and P_{12}). Note that this transformation is true under set-semantics but it fails when we consider bag-semantics, because it duplicates the bag of solutions. To solve this problem, we consider a double negation of the filter condition, that is $(\neg(\neg(C_1 \wedge C_2)))$ (as is showed by the rules for predicates p and p'). Given that negated literals does not increase solutions, in the rule for predicate p we will have only solutions from predicate p_1, which satisfies the bag semantics.

(6.4) C is $(C_1 \vee C_2)$. We have that $\delta(P, g)$ is the set of rules
$\{\ \delta(P_{11}, g)_D,\ \delta(P_{12}, g)_D,$
$p''(\overline{\text{var}}(P_1)) \leftarrow p_{11}(\overline{\text{var}}(P_1)),$

$p''(\overline{\text{var}}(P_1)) \leftarrow p_{12}(\overline{\text{var}}(P_1))$,
$p'(\overline{\text{var}}(P_1)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p''(\overline{\text{var}}(P_1))$,
$p(\overline{\text{var}}(P)) \leftarrow p_1(\overline{\text{var}}(P_1)) \wedge \neg p'(\overline{\text{var}}(P_1))$ }

where $P_{11} = (P_1 \text{ FILTER } C_1)$ and $P_{12} = (P_1 \text{ FILTER } C_2)$.
Note that the graph pattern $(P_1 \text{ FILTER}(C_1 \vee C_2))$ can be rewritten as $((P_1 \text{ FILTER } C_1) \text{ UNION}(P_1 \text{ FILTER } C_2))$ (it is showed by the rules for predicate p'' and by the patterns P_{11} and P_{12}). As was explained in (6.3), this transformation fails when we consider bag-semantics, and we also consider a double negation of the filter condition to solve such problem.

Note that, for each graph pattern P the transformation $\delta(P, g)$ preserves the bag semantics of the SPARQL WG specification. Consider the cardinality m of a solution s for P (and for the equivalent solution for $\delta(P, g)$), then it can be checked that: in case (1), the value of m is 1 because each triple occurs once in the active graph; in case (2), m is the product of the cardinalities for s in two bags of solutions. in case (3), m is the sum of the cardinalities for s in two bags of solutions. in case (4), m is given by the product of cardinalities for s in two bags of solutions, plus the cardinalities for s in a bag of solutions. In case (5.1), m is given by the cardinality of s in the bag of solutions for named graph n. In case (5.2), m is given by the sum of cardinalities for s in the bag of solutions for each named graph in the dataset. In cases (6.1),(6.2),(6.3), and (6.4), n is given by the cardinality of s in the bag of solutions for P_1.

6.2 From Datalog to SPARQL$_C$

Let $L_d = (\mathcal{Q}_d, \mathcal{D}_d, \mathcal{S}_d, \text{ans}_d)$ be the Datalog language and $L_s = (\mathcal{Q}_s, \mathcal{D}_s, \mathcal{S}_s, \text{ans}_s)$ be the SPARQL$_C$ language. To prove that L_d is contained in L_s, we define transformations $\mathcal{T}'_Q : \mathcal{Q}_d \to \mathcal{Q}_s$, $\mathcal{T}'_D : \mathcal{D}_d \to \mathcal{D}_s$, and $\mathcal{T}'_S : \mathcal{S}_d \to \mathcal{S}_s$. That is, $\mathcal{T}'_Q$ transforms a Datalog Query into an SPARQL$_C$ query, $\mathcal{T}'_D$ transforms a set of Datalog facts into an SPARQL$_C$ dataset, and $\mathcal{T}'_S$ transforms a set of Datalog substitutions into a set of SPARQL$_C$ solution mappings.

Datalog facts as a Dataset. A Datalog fact $p(c_1, ..., c_n)$ can be described by a set of RDF triples as follows

$$\text{desc}(p(c_1, \ldots, c_n)) = \{(\_\text{:b,predicate,p}), (\_\text{:b,rdf:\_1}, c_1), \ldots, (\_\text{:b,rdf:\_n}, c_n)\}, \quad (2)$$

where _:b is a fresh blank node. Given a set of Datalog facts F, we define

$$\mathcal{T}'_D(F) = \{\text{desc}(f) \mid f \in F\},$$

where $\text{blank}(\text{desc}(f_i)) \cap \text{blank}(\text{desc}(f_j)) = \emptyset$ for each $f_i, f_j \in F$ where $i \neq j$. Then, $\mathcal{T}'_D(F)$ returns an RDF graph which describes the set of Datalog facts F.

Datalog substitutions as solution mappings. Given a set of substitutions Θ, the set of mappings obtained from Θ, denoted $\mathcal{T}'_S(\Theta)$, is defined as follows: for each substitution $\theta \in \Theta$ there exists a mapping $\mu \in \mathcal{T}'_S(\Theta)$ satisfying that, if $x/t \in \theta$ then $x \in \mathrm{dom}(\mu)$ and $\mu(x) = t$.

Due to the similarity of the objects and to avoid complicating the notation, we will not distinguish between substitutions and maps (that is, will consider $\mathcal{T}'_S$ as the identity).

Datalog rules as SPARQL$_C$ graph patterns. Let Π be a Datalog program, and $L = p(x_1, \ldots, x_n)$ be a literal where p is a predicate in Π and each x_i is a variable. We will define recursively a function $\mathrm{gp}(L)_\Pi$ which returns an SPARQL$_C$ graph pattern encoding the program (Π, L), that is, the fragment of the Datalog program Π used for evaluating literal L.

The translation works intuitively as follows. For facts, same as (2) above. Let L_k^{eq} literals of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$. Consider the following rule of a program Π:

$$L \leftarrow p_1 \wedge \cdots \wedge p_s \wedge \neg q_1 \wedge \cdots \wedge \neg q_t \wedge L_1^{eq} \wedge \cdots \wedge L_u^{eq}. \tag{3}$$

In this case $\mathrm{gp}(L)_\Pi$ returns a graph pattern with the structure

$$\begin{aligned}(((\cdots((\mathrm{gp}(p_1)_\Pi \,\mathrm{AND} \cdots \mathrm{AND}\, \mathrm{gp}(p_s)_\Pi) \\ \mathrm{MINUS}\, \mathrm{gp}(q_1)) \cdots) \,\mathrm{MINUS}\, \mathrm{gp}(q_t)) \\ \mathrm{FILTER}(L_1^{eq} \wedge \cdots \wedge L_u^{eq})).\end{aligned} \tag{4}$$

The formal definition is Algorithm 2.

Datalog queries as SPARQL$_C$ queries. Now we can define the transformation $\mathcal{T}'_Q$ as follows: given a Datalog query $Q_d = (\Pi, L)$ where $L = p(x_1, \ldots, x_n)$, the SPARQL$_C$ query defined by $\mathcal{T}'_Q(Q_d)$ is

$$(\mathrm{SELECT}\, x_1 \cdots x_n, \mathrm{FROM}\, g, \mathrm{gp}(L)_\Pi),$$

where $\mathrm{graph}(g) = \mathcal{T}'_D(\mathrm{facts}(\Pi))$.

Theorem 5. *nr-Datalog$^\neg$ is contained in SPARQL$_C$.*

Proof. Consider transformations $\mathcal{T}'_Q$, $\mathcal{T}'_D$, and $\mathcal{T}'_S$ as defined above. We need to prove that for every Datalog query Q_d with database D_d, it satisfies that $\mathcal{T}'_S(\mathrm{ans}_d(Q_d, D_d)) = \mathrm{ans}_k(\mathcal{T}'_Q(Q_d), \mathcal{T}'_D(D_d))$,

Let $Q_d = (\Pi, L)$ be a Datalog query where $L = p(x_1, \ldots, x_n)$, and D_d be the Datalog database facts(Π). Then, we have to show that

$$\mathcal{T}'_S(\mathrm{ans}_d((\Pi, L), D_d)) = [\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)}.$$

The proof is by induction on the level l of the program (Π, L). The level l of a program (Π, L) is the number defined recursively as follows: $l(\Pi, \neg p) = l(\Pi, p)$

Algorithm 2 Transformation of Datalog predicates into SPARQL$_C$ patterns

if predicate p is extensional in Π **then**
  $\mathrm{gp}(p(x_1, \ldots, x_n))_\Pi \leftarrow ((z, \mathrm{predicate}, p)\,\mathrm{AND}(z, \mathrm{rdf{:}\_1}, x_1)\,\mathrm{AND} \cdots \mathrm{AND}(z, \mathrm{rdf\_n}, x_n))$, where z is a fresh variable.
else if predicate p is intensional in Π **then**
  $P \leftarrow \emptyset$
  for each rule $r \in \Pi$ with head $p(x'_1, \ldots, x'_n)$ **do**
    $P' \leftarrow \emptyset$
    $C \leftarrow \emptyset$
    Let $r' = \nu(r)$ where ν is a substitution such that $\nu(x'_i) = x_i$
    for each positive literal $q(y_1, \ldots, y_m)$ in the body of r' **do**
      if $P' = \emptyset$ **then**
        $P' \leftarrow \mathrm{gp}(q)_\Pi$
      else
        $P' \leftarrow (P'\,\mathrm{AND}\,\mathrm{gp}(q)_\Pi)$
      end if
    end for
    for each negative literal $\neg q(y_1, \ldots, y_m)$ in the body of r' **do**
      $P' \leftarrow (P'\,\mathrm{MINUS}\,\mathrm{gp}(q))$
    end for
    for each equality formula $t_1 = t_2$ in r' **do**
      if $C = \emptyset$ **then**
        $C \leftarrow (t_1 = t_2)$
      else
        $C \leftarrow C \wedge (t_1 = t_2)$
      end if
    end for
    for each negative literal $\neg(t_1 = t_2)$ in r' **do**
      if $C = \emptyset$ **then**
        $C \leftarrow \neg(t_1 = t_2)$
      else
        $C \leftarrow C \wedge \neg(t_1 = t_2)$
      end if
    end for
    if $C \neq \emptyset$ **then**
      $P' \leftarrow (P'\,\mathrm{FILTER}\,C)$
    end if
    if $P = \emptyset$ **then**
      $P \leftarrow P'$
    else
      $P \leftarrow (P\,\mathrm{UNION}\,P')$
    end if
  end for
  $\mathrm{gp}(p)_\Pi \leftarrow P$
end if

for a predicate p; $l(\Pi, p) = 0$ if p is an extensional predicate; $l(\Pi, p) = 1 + \max_i(l(\Pi, L_i))$ if p is intensional and L_i are all literals which occur in any rule

with head p in Π. (The function is well defined because the programs considered are not recursive).

Base case: $l(\Pi, L) = 0$. Note that in this case L matches line 1, hence $\mathrm{gp}(L)_\Pi$ returns the graph pattern

$$((z, \text{predicate}, p)\,\text{AND}(z, \text{rdf:\_1}, x_1)\,\text{AND}\cdots\text{AND}(z, \text{rdf:\_n}, x_n)).$$

Now, a mapping μ is in $[\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)}$ if and only if $\mathrm{dom}(\mu) = \{x_1, \ldots, x_n\}$ and for every triple pattern t in $\mathrm{gp}(L)_\Pi$ it satisfies that $\mu(t) \in \mathrm{dg}(D)$. Considering that $\mathrm{dg}(D) = \mathcal{T}'_D(\mathrm{facts}(\Pi))$, we have that

$$\{(\mu(z), \text{predicate}, p), (\mu(z), \text{rdf:\_1}, \mu(x_1)), \ldots, (\mu(z), \text{rdf\_n}, \mu(x_n))\} \subseteq \mathcal{T}'_D(\mathrm{facts}(\Pi)). \quad (5)$$

Now, a substitution θ is in $\mathrm{ans}_d((\Pi, L), D_d)$ if and only if $\theta(L) \in \mathrm{facts}(\Pi)$, that is $p(\theta(x_1), \ldots, \theta(x_n)) \in \mathrm{facts}(\Pi)$. Note that, we only consider the initial database $\mathrm{facts}(\Pi)$ because predicate p is extensional. So, if we transform $\theta(L) = p(\theta(x_1), \ldots, \theta(x_n))$ in a set of RDF triples we get:

$$\mathrm{desc}(\theta(L)) = \{(\theta(z), \text{predicate}, p), (\theta(z), \text{rdf:\_1}, \theta(x_1)), \ldots, (\theta(z), \text{rdf\_n}, \theta(x_n))\}.$$

Hence it is easy to see that $\mathcal{T}'_S$ maps bijectively substitutions of $\mathrm{ans}_d((\Pi, L), D_d)$ to mappings in $[\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)}$.

Inductive step: $l(\Pi, L) = n > 0$. Recall that $Q_d = (\Pi, L)$ and $L = p(x_1, \ldots, x_n)$. Additionally, we assume that $r_1, \ldots, r_m$ are all the rules of Π with head $p(\ldots)$. A substitution θ is in $\mathrm{ans}_d(Q_d, D_d)$ if and only if there is a rule r_i in the set $\{r_1, \ldots, r_m\}$, such that $\theta'(r_i)$ is true in Π.

On the other hand, in this case L matches line 3 of the algorithm, hence it returns the graph pattern

$$(\ \mathrm{gp}(L_1)_\Pi\,\text{UNION}\cdots\text{UNION}\,\mathrm{gp}(L_m)_\Pi\ ),$$

where $\mathrm{gp}(L_i)_\Pi$ is the graph pattern obtained for rule $r_i \in \{r_1, \ldots, r_m\}$. In this case it clearly holds that $[\![\mathrm{gp}(L)_\Pi]\!]^D_{\mathrm{dg}(D)} = \bigcup_i [\![\mathrm{gp}(L_i)_\Pi]\!]^D_{\mathrm{dg}(D)}$. Using the fact $\mathrm{ans}((\Pi, L), D_d) = \bigcup_{L_i} \mathrm{ans}(\Pi_{L_i}, D_d)$ and the fact that the programs are not recursive, it is enough to prove that for each particular rule j:

$$\mathcal{T}'_S(\mathrm{ans}((\Pi, L_j), D_d)) = [\![\mathrm{gp}(L_j)]\!]^D_{\mathrm{dg}(D)}. \quad (6)$$

To prove this, assume that this rule has the following general structure (we are not writing explicitly the variables):

$$p \leftarrow p_1 \wedge \cdots \wedge p_s \wedge \neg q_1 \wedge \cdots \wedge \neg q_t \wedge L_1^{eq} \wedge \cdots \wedge L_u^{eq}, \quad (7)$$

where p_i, q_j are predicates and and each L_k^{eq} is a literal of the form $t_1 = t_2$ or $\neg(t_1 = t_2)$. (We are including here literals of the form $t = c$ for c constant.)

Let us compute the SPARQL evaluation first. We have that $\mathrm{gp}(p)_{\Pi}$ returns a graph pattern with the structure

$$\begin{aligned}(((\cdots((\mathrm{gp}(p_1)_{\Pi}\,\mathrm{AND}\cdots\mathrm{AND}\,\mathrm{gp}(p_s)_{\Pi})\\ \mathrm{MINUS}\,\mathrm{gp}(q_1))\cdots)\,\mathrm{MINUS}\,\mathrm{gp}(q_t))\\ \mathrm{FILTER}(L_1^{eq}\wedge\cdots\wedge L_u^{eq})),\end{aligned}\quad(8)$$

Observe that a mapping μ is in $[\![\mathrm{gp}(p)_{\Pi}]\!]^D_{\mathrm{dg}(D)}$ if and only if

(i) for each p_i, there exists a mapping $\mu_i' \in [\![\mathrm{gp}(p_i)_{\Pi}]\!]^D_{\mathrm{dg}(D)}$ satisfying that μ and μ_i' are compatible;
(ii) for each q_j, there exists no mapping $\mu_j'' \in [\![\mathrm{gp}(q_j)_{\Pi}]\!]^D_{\mathrm{dg}(D)}$ satisfying that μ and μ_j'' are compatible; and
(iii) for each L_k^{eq}, it satisfies that $\mu(t_1)=\mu(t_2)$ when L_k^{eq} is $t_1=t_2$, and $\mu(t_1)\neq\mu(t_2)$ when L_k^{eq} is $\neg(t_1=t_2)$ (suppose that $\mu(t)=t$ where t is a constant).

Now, let us compute the Datalog evaluation. A substitution θ is in the result of $\mathrm{ans}_d((\Pi,p),D_d)$ if and only if $\theta(p)\in\mathrm{facts}^*(\Pi)$. This means exactly:

(a) for each p_i, there exists a substitution $\theta_i' \in \mathrm{ans}_d((\Pi,p_i),D_d)$ satisfying that for each variable $x\in\mathrm{var}(\theta')\cap\mathrm{var}(\theta_i')$, $\theta(x)=\theta_i'(x)$.
(b) for each q_j, there exists no substitution $\theta_j'' \in \mathrm{ans}_d((\Pi,q_j),D_d)$ satisfying that for each variable $x\in\mathrm{var}(\theta)\cap\mathrm{var}(\theta_j'')$, $\theta(x)=\theta_i''(x)$
(c) for each literal L_k^{eq}, it satisfies that $\theta'(t_1)=\theta'(t_2)$ when L_k^{eq} is $t_1=t_2$, and $\theta'(t_1)\neq\theta'(t_2)$ when L_k^{eq} is $\neg(t_1=t_2)$ (assume that $\theta'(t)=t$ where t is a constant).

Note that (because Π is not recursive), for each pair of literal p_i, q_j, it holds that $l(\Pi,p_i)<n$ and $l(\Pi,q_j)<n$. Hence, by induction hypothesis we have that $\mathcal{T}_S'(\mathrm{ans}_d((\Pi,p_i),D_d)) = [\![\mathrm{gp}(p_i)_{\Pi}]\!]^D_{\mathrm{dg}(D)}$ and $\mathcal{T}_S'(\mathrm{ans}_d((\Pi,q_j),D_d)) = [\![\mathrm{gp}(q_j)_{\Pi}]\!]^D_{\mathrm{dg}(D)}$. These identities plus the conditions (i), (ii), (iii) and (a), (b), (c) above show the bijections between maps $\mu\in[\![\mathrm{gp}(p)]\!]^D_{\mathrm{dg}(D)}$ and substitutions $\theta\in\mathrm{ans}((\Pi,p),D_d)$, that is:

$$\mathcal{T}_S'(\mathrm{ans}((\Pi,p),D_d)) = [\![\mathrm{gp}(p)]\!]^D_{\mathrm{dg}(D)}.$$

Note 5. It is possible to enhance the transformation $\mathcal{T}'$ from Datalog to SPARQL_C in order for it to be the inverse of the transformation $\mathcal{T}$ from SPARQL_C to Datalog. Define $\mathcal{T}''$ from SPARQL_C to SPARQL_C as follows:

Let P be the pattern of query $\mathcal{T}'(\mathcal{T}(Q))$. For each sub-pattern P' occurring in P with the structure

$$((?\mathrm{Z},\mathrm{predicate},\mathrm{triple})\,\mathrm{AND}(?Z,\mathrm{rdf}{:}\_1,x_1)\,\mathrm{AND}(?Z,\mathrm{rdf}{:}\_2,x_2)\,\mathrm{AND}(?Z,\mathrm{rdf}{:}\_3,x_3))$$

where $?Z\in V$ and $x_1,x_2,x_3\in T$, replace P' by the graph pattern (x_1,x_2,x_3).

Then it is not difficult to check that $[\![Q]\!] = [\![\mathcal{T}''(\mathcal{T}'(\mathcal{T}(Q)))]\!]$.

7 Conclusions

We have studied the expressive power of SPARQL. Among the most important findings are the definition of negation, the proof that non-safe filter expressions are superfluous, the proof of the equivalence of the WG semantics and the compositional one presented in [6].

From these results we can state the most relevant result of the paper:

Theorem 6 (main). *$SPARQL_{WG}$ has the same expressive power as relational algebra under bag semantics.*

This result follows from the well known fact (for example, see [1] and [5]) that relational algebra and non-recursive safe Datalog with negation have the same expressive power, and from theorems 2, 3, 4 and 5.

We will not develop here the consequences of such equivalence. Surely it will help the design of its extensions and suggest some syntactic restriction which are today not present. Future work includes the development of the manifold consequences implied by the Main Theorem.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. R. Cyganiak. A relational algebra for sparql. Technical Report HPL-2005-170, HP Labs, 2005.
3. T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Reasoning Web*, number 4126 in LNCS, pages 1–52, 2006.
4. G. Klyne and J. Carroll. Resource Description Framework (RDF) Concepts and Abstract Syntax. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/, Feb 2004.
5. M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springe-Verlag, 1999.
6. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, number 4273 in LNCS, pages 30–43. Springer-Verlag, 2006.
7. J. Pérez, M. Arenas, and C. Gutierrez. Semantics of SPARQL. Technical Report TR/DCC-2006-17, Department of Computer Science, Universidad de Chile, 2006.
8. A. Polleres. From sparql to rules (and back). In *Proceedings of the 16th International World Wide Web Conference (WWW)*, pages 787–796. ACM, 2007.
9. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, January 2008.
10. S. Schenk. A sparql semantics based on datalog. In *30th Annual German Conference on Advances in Artificial Intelligence (KI)*, volume 4667 of *LNCS*, pages 160–174. Springer, 2007.

8 Appendix I: SPARQL Syntax and Semantics (W3C)

8.1 SPARQL Syntax

A *query variable* is a member of the set V where V is infinite and disjoint from RDF-T. A *triple pattern* is member of the set (RDF-T$\cup V$) $\times$ ($I \cup V$) $\times$ (RDF-T$\cup V$) A *Basic Graph Pattern* is a set of triple patterns.

A *SPARQL query string* is syntactically represented by a query block which consists of a *query form*, zero o more *dataset clauses*, a *where clause*, and possibly *solution modifiers*.

SPARQL graph patterns. A *where clause* provides a *graph pattern* which is defined using the following forms: GroupGraphPattern, TriplesBlock, Filter, OptionalGraphPattern, GroupOrUnionGraphPattern, GraphGraphPattern. The form TriplesBlock denote an expression (t_1 . $\cdots$. t_n) where each t_i is a triple pattern.

The syntax for graph patterns is defined by the following grammar (we use GP to abbreviate GraphPattern):

[13] WhereClause ::= 'WHERE'? GroupGP
[20] GroupGP ::= '{' TriplesBlock? ((GPNotTriples | Filter) '.'? TriplesBlock?)* '}'
[22] GPNotTriples ::= OptionalGP | GroupOrUnionGP | GraphGP
[23] OptionalGP ::= 'OPTIONAL' GroupGP
[24] GraphGP ::= 'GRAPH' VarOrIRIref GroupGP
[25] GroupOrUnionGP ::= GroupGP ('UNION' GroupGP)*
[26] Filter ::= 'FILTER' Constraint

SPARQL Algebra. A graph pattern in the SPARQL algebra is defined recursively as follows:

- If p is a basic graph pattern, then BGP(p) is a graph pattern.
- If P_1, P_2 are graph patterns and C is a filter constraint, then Join(P_1, P_2), LeftJoin(P_1, P_2, C), Diff(P_1, P_2, C), Filter(C, P_1), Union(P_1, P_2) are graph patterns.
- If P is a graph pattern and $n \in I \cup V$ then Graph(n, P) is a graph pattern.

Transforming graph patterns into algebra expressions. Consider function Transform which receives a syntax form as input and returns a graph pattern in the SPARQL algebra. Given a syntax form f, the function Transform(f) is defined recursively as follows:

- If f is TripleBlock then
  Transform(f) = BGP(list of triple patterns)

– If f is GroupGraphPattern then Transform(f) = G where G is defined as follows:

Let FS := the empty set
Let G := the empty pattern, Z, a basic graph pattern which is the empty set.
for each element E in the GroupGraphPattern **do**
  if E is of the form FILTER(expr) **then**
    FS := FS set-union {expr}
  else if E is of the form OPTIONAL{P} **then**
    Let A := Transform(P)
    if A is of the form Filter($F, A2$) **then**
      G := LeftJoin($G, A2, F$)
    else
      G := LeftJoin($G, A, true$)
    end if
  else if E is any other form **then**
    Let A := Transform(E)
    G := Join(G, A)
  end if
end for
if FS is not empty **then**
  Let X := Conjunction of expressions in FS
  G := Filter(X, G)
end if
return G

– If f is GroupOrUnionGraphPattern of the form G_1 UNION $\cdots$ UNION G_n, then Transform(f) = A where A is defined as follows:

Let A := undefined
for each element G in the GroupOrUnionGraphPattern **do**
  if A is undefined **then**
    A := Transform(G)
  else
    A := Union(A, Transform(G))
  end if
end for
return A

– If f is GraphGraphPattern then

if the form is GRAPH IRI GroupGraphPattern **then**
  Transform(f) = Graph(IRI, Transform(GroupGraphPattern))
else if the form is GRAPH Var GroupGraphPattern **then**
  Transform(f) = Graph(Var, Transform(GroupGraphPattern))
end if

8.2 Evaluation semantics

A *solution mapping*, μ, is a partial function $\mu : V \rightarrow T$. The domain of μ, $\text{dom}(\mu)$, is the subset of V where μ is defined. Two solution mappings μ_1 and μ_2 are *compatible*, if for every variable v in $\text{dom}(\mu_1)$ and in $\text{dom}(\mu_2)$, $\mu_1(v) = \mu_2(v)$. Write μ_0 for the mapping such that $\text{dom}(\mu_0)$ is the empty set. If μ_1 and μ are compatible then μ_1 set-union μ_2 is also a mapping. Write $\text{merge}(\mu_1, \mu_2)$ for μ_1 set-union μ_2.

A *solution sequence* is a multiset of solution mappings, possibly unordered. A multiset (also known as a bag) is an unordered collection of elements in which each element may appear more than once. It is described by a set of elements and a cardinality function giving the number of occurrences of each element from the set in the multiset. Write $\text{card}[\Omega](\mu)$ for the cardinality of solution mapping μ in a multiset of mappings Ω.

A Pattern Instance Mapping, ρ, is the combination of an RDF instance mapping, σ, and a solution mapping, μ. That is $\rho(.) = \mu(\sigma(.))$

Basic Graph Pattern Matching. Let BGP be a basic graph pattern and let G be an RDF graph. A mapping μ is a solution for BGP from G when there is a pattern instance mapping ρ such that $\rho(BGP)$ is a subgraph of G and μ is the restriction of ρ to the query variables in BGP.

We have that $\text{card}[\Omega](\mu)$ is given by the number of distinct RDF instance mappings, Ω, such that $\rho(.) = \mu(\sigma(.))$ is a pattern instance mapping and $\rho(BGP)$ is a subgraph of G).

Evaluation of SPARQL Algebra Expressions. Let p be a basic graph pattern, P_1, P_2 be graph patterns and expr be a filter constraint. The evaluation of a graph pattern P with respect to a dataset D having active graph G (the active graph is initially the default graph), denoted $\text{eval}(D(G), P)$, is defined recursively as follows:

- $\text{eval}(D(G), \text{BGP}(p)) = \{\mu \mid \mu \text{ is a solution for } p \text{ from } G\ \}$
- $\text{eval}(D(G), \text{Join}(P_1, P_2)) = \text{Join}(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2))$
- $\text{eval}(D(G), \text{LeftJoin}(P_1, P_2, \text{expr})) = \text{LeftJoin}(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2), \text{expr})$
- $\text{eval}(D(G), \text{Union}(P_1, P_2)) = \text{Union}(\text{eval}(D(G), P_1), \text{eval}(D(G), P_2))$
- $\text{eval}(D(G), \text{Filter}(F, P_1)) = \text{Filter}(\text{expr}, \text{eval}(D(G), P_1))$
- $\text{eval}(D(G), Graph(IRI, P))$ then
  if IRI is a graph name in D **then**
  $\text{eval}(D(G), \text{Graph}(IRI, P)) = \text{eval}(D(D[IRI]), P)$
  else if IRI is not a graph name in D **then**
  $\text{eval}(D(G), \text{Graph}(IRI, P))$ = the empty multiset
  end if

– eval($D(G)$, Graph($?var, P$)) then
  Let R be the empty multiset
  for each IRI i in D **do**
    $R :=$ Union(R, Join(eval($D(D[i]), P$), $\Omega(?var \rightarrow i)$)
  end for
  return R

Semantics of SPARQL Algebra operators. Let Ω_1 and Ω_2 be multisets of solution mappings, and expr be a filter expression.

– Join$(\Omega_1, \Omega_2) = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1$ and μ_2 are compatible $\}$
  card[Join$(\Omega_1, \Omega_2)](\mu)$ is given by the following iteration:
  for each merge(μ_1, μ_2) such that $\mu =$ merge(μ_1, μ_2) **do**
    sum over (μ_1, μ_2), card$[\Omega_1](\mu_1) *$ card$[\Omega_2](\mu_2)$
  end for
– Diff$(\Omega_1, \Omega_2,$ expr$) = \{\mu \mid \mu \in \Omega_1$ such that for all $\mu' \in \Omega_2$, either μ and μ' are not compatible or μ and μ' are compatible and expr$(\mu \cup \mu')$ has an effective boolean value of false $\}$
    card[Diff$(\Omega_1, \Omega_2,$ expr$)](\mu) =$ card$[\Omega](\mu_1)$
– LeftJoin$(\Omega_1, \Omega_2,$ expr$) =$ Filter(expr, Join$(\Omega_1, \Omega_2)) \cup$ Diff$(\Omega_1, \Omega_2,$ expr$)$
    card[LeftJoin$(\Omega_1, \Omega_2,$ expr$)](\mu) =$ card[Filter(expr, Join$(\Omega_1, \Omega_2))](\mu)$
      $+$ card[Diff$(\Omega_1, \Omega_2,$ expr$)](\mu)$
– Union$(\Omega_1, \Omega_2) = \{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$
    card[Union$(\Omega_1, \Omega_2)](\mu) =$ card$[\Omega_1](\mu) +$ card$[\Omega_2](\mu)$
– Filter$($expr$, \Omega) = \{\mu \mid \mu \in \Omega$ and expr(μ) is an expression that has an effective boolean value of true $\}$
    card[Filter(expr, Ω)]$(\mu) =$ card$[\Omega](\mu)$

9 Appendix II: Semantics of SPARQL$_C$.

9.1 RDF and Datasets

Assume there are pairwise disjoint infinite sets I, B, L (IRIs, Blank nodes, and RDF literals respectively). We denote by T the union $I \cup B \cup L$ (RDF *terms*).

A tuple $(v_1, v_2, v_3) \in (I \cup B) \times I \times T$ is called an *RDF triple*, where v_1 is the *subject*, v_2 the *predicate*, and v_3 the *object*. An *RDF Graph* [4] (just graph from now on) is a set of RDF triples. Given a graph G, we denote by term(G) the set of elements of T appearing in the triples of G and blank(G) denotes the set of blank nodes in G, i.e. blank$(G) =$ term$(G) \cap B$. If G is referred to by an IRI u, then graph(u) returns the graph available in u, i.e, $G =$ graph(u).

We define two operations on two graphs G_1 and G_2. The *union* of graphs, denoted $G_1 \cup G_2$, is the set theoretical union of their sets of triples. The *merge* of graphs, denoted $G_1 + G_2$, is the graph $G_1 \cup G_2'$ where G2' is the graph obtained from G_2 by renaming its blank nodes to avoid clashes with those in G_1.

An *RDF dataset* is a set $\{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ where each G_i is a graph and each u_j is an IRI. G_0 is called the *default graph* and each pair $\langle u_i, G_i\rangle$ is called a *named graph*. Every dataset satisfies that: (i) it always contains one default graph, (ii) there may be no named graphs, (iii) each u_j is distinct, and (iv) $\text{blank}(G_i) \cap \text{blank}(G_j) = \emptyset$ for $i \neq j$.

Given a dataset $D = \{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$, we denote by $\text{term}(D)$ the set of terms occurring in the graphs of D. The default graph of D is denoted $\text{dg}(D)$. For a named graph $\langle u_i, G_i\rangle$ we have that $\text{name}(G_i)_D = u_i$ and $\text{graph}(u_i)_D = G_i$; otherwise $\text{name}(G_i)_D = \emptyset$ and $\text{graph}(u_i)_D = \emptyset$. We denote by $\text{names}(D)$ the set of IRIs $\{u_1, \ldots, u_n\}$. Although $\text{name}(G_0) = \varnothing$, we sometimes will use g_0 when referring to G_0. Finally, the *active graph* of D is the graph G_i used for querying the dataset.

9.2 Semantics of SPARQL$_C$

A *mapping* μ is a partial function $\mu : V \to T$. The domain of μ, $\text{dom}(\mu)$, is the subset of V where μ is defined. The *empty mapping* μ_0 is a mapping such that $\text{dom}(\mu_0) = \emptyset$. Two mappings μ_1, μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, when for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it satisfies that $\mu_1(?X) = \mu_2(?X)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping. We define the operations of join, union, difference, and left outer-join between two sets of mappings Ω_1, Ω_2 as:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$$
$$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$$
$$\Omega_1 \text{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$$

Now, we will define the semantics for the components of a SPARQL query:

(1) *[Semantics for result query forms]* Given a mapping μ and a result query form R, the result generated by R given μ, denoted $\text{result}(R, \mu)$, is defined as follows:
   - If R is SELECT W, then $\text{result}(R, \mu)$ is the restriction of μ to W, that is the mapping denoted $\mu_{|W}$ such that, $\text{dom}(\mu_{|W}) = \text{dom}(\mu) \cap W$ and $\mu_{|W}(?X) = \mu(?X)$ for every $?X \in \text{dom}(\mu) \cap W$.
   - If R is CONSTRUCT H, then $\text{result}(R, \mu)$ is the graph $\{\mu(t) \mid t \in H\}$, where $\mu(t)$ is the triple obtained by replacing the variables in t according to μ and satisfying that $\mu(t) \subset (I \cup B) \times I \times T$.
   - If R is ASK, then $\text{result}(R, \mu)$ is *false* if $\mu = \emptyset$ and $\text{result}(R, \mu)$ is *true* otherwise.

(2) *[Semantics for dataset clauses]* Given a set of dataset clauses F, the RDF dataset resulting from F, denoted $\text{dataset}(F)$, contains:
   (i) A default graph consisting of the merge of the graphs referred to in the FROM clauses of F. If there is no FROM clause, then the dataset includes an empty graph $G_0 = \emptyset$ as the default graph.
   (ii) A named graph $\langle u, \text{graph}(u)\rangle$ for each clause "FROM NAMED u" in F.

(3) *[Semantics for filter constraints]* Given a mapping μ and a filter constraint C, we say that μ satisfies C, denoted by $\mu \models C$, if:
- C is bound($?X$) and $?X \in \text{dom}(\mu)$;
- C is $?X = u$, $?X \in \text{dom}(\mu)$ and $\mu(?X) = u$.
- C is $?X = ?Y$, $?X \in \text{dom}(\mu)$, $?Y \in \text{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$.
- C is $(\neg C_1)$, C_1 is a filter constraint, and it is not the case that $\mu \models C_1$.
- C is $(C_1 \vee C_2)$, C_1 and C_2 are filter constraints, and $\mu \models C_1$ or $\mu \models C_2$.
- C is $(C_1 \wedge C_2)$, C_1 and C_2 are filter constraints, $\mu \models C_1$ and $\mu \models C_2$.

(4) *[Semantics for graph patterns]* Let P_1, P_2 be graph patterns and C be a filter constraint. The *evaluation* of a graph pattern P over a dataset D with active graph G, denoted $[\![P]\!]^D_G$, is defined recursively as follows:
- If t is a triple pattern, $[\![t]\!]^D_G = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in G\}$ where $\mu(t)$ is the triple obtained by replacing the variables in t according to μ.
- $[\![(P_1 \text{ AND } P_2)]\!]^D_G = [\![P_1]\!]^D_G \bowtie [\![P_2]\!]^D_G$
- $[\![(P_1 \text{ OPT } P_2)]\!]^D_G = [\![P_1]\!]^D_G \text{⟕} [\![P_2]\!]^D_G$
- $[\![(P_1 \text{ UNION } P_2)]\!]^D_G = [\![P_1]\!]^D_G \cup [\![P_2]\!]^D_G$
- $[\![(P_1 \text{ FILTER } C)]\!]^D_G = \{\mu \mid \mu \in [\![P_1]\!]^D_G \text{ and } \mu \models C\}$ $(\text{var}(C) \subseteq \text{var}(P).)$
- If $u \in I$, then $[\![u \text{ GRAPH } P]\!]^D_G = [\![P]\!]^D_{\text{graph}(u)_D}$.
  If $?X \in V$, then
  $$[\![(?X \text{ GRAPH } P)]\!]^D_G = \bigcup_{v \,\in\, \text{names}(D)} ([\![P]\!]^D_{\text{graph}(v)_D} \bowtie \{\mu_{?X \to v}\}),$$
  where $\mu_{?X \to v}$ is a mapping such that $\text{dom}(\mu) = \{?X\}$ and $\mu(?X) = v$.

Given a SPARQL query $Q = (R, F, P)$ and the RDF dataset $D = \text{dataset}(F)$. The *answer* to Q over database D, denoted $\text{ans}_s(Q, D)$, is defined as follows:
- if R is SELECT W, then $\text{ans}_s(Q, D) = \{\text{result}(R, \mu) \mid \mu \in [\![P]\!]^D_{\text{dg}(D)}\}$
- if R is CONSTRUCT H and blank(H) is the set of blank nodes appearing in H, then $\text{ans}_s(Q, D) = \{\beta_i(\text{result}(R, \mu_i)) \mid \mu_i \in [\![P]\!]^D_{\text{dg}(D)}\}$ where each $\beta_i : \text{blank}(H) \to (B \setminus H)$ is a *blank renaming function* such that for each pair of mappings $\mu_1, \mu_2 \in [\![P]\!]^D_{\text{dg}(D)}$, $\text{range}(\beta_1) \cap \text{range}(\beta_2) = \emptyset$.
- if R is ASK, then $\text{ans}_s(Q, D) = \textit{false}$ when $[\![P]\!]^D_{\text{dg}(D)} = \emptyset$ (i.e., there exists no mapping $\mu \in [\![P]\!]^D_{G_0}$) and $\text{ans}_s(Q, D) = \textit{true}$ otherwise.

9.3 Bag Semantics in SPARQL

The original compositional semantics presented in [6] considered *sets* of mappings. Later, it was extended to include *bags* [7]. In what follows, we present a summary of those definitions.

A *bag* (or *multiset*) is a set of annotated elements; the annotation of an element, also called the cardinality (or multiplicity) of the element, is a positive integer. Intuitively, a bag may contain duplicate occurrences of an element; the cardinality of an element indicates the number of duplicates for the element in the set.

The cardinality of a mapping μ in a *bag of mappings* Ω will be denoted by $\text{card}_s(\mu)_{[\Omega]}$ (or simply $\text{card}_s(\mu)$ when is clear from the context). If $\mu \notin \Omega$ then $\text{card}_s(\mu)_{[\Omega]} = 0$.

Definition 2 (Cardinality of Basic Graph Pattern Solutions). *Consider a basic graph pattern P (possibly with blank nodes) and a dataset D with active graph G. The cardinality of a mapping $\mu \in [\![P]\!]_G^D$ is defined as the number of distinct RDF instance mappings $\sigma : \text{blank}(P) \to \text{term}(G)$ such that $\mu(\sigma(P)) \subseteq G$, i.e.*

$$\text{card}_s(\mu)_{[[\![P]\!]_G^D]} = \mid \{\sigma : \text{blank}(P) \to \text{term}(G) \mid \mu(\sigma(P)) \subseteq G\} \mid$$

Note 6. For a basic graph pattern P without blank nodes, every solution $\mu \in [\![P]\!]_G^D$ has cardinality 1, as in this case the only possible substitution is $\sigma : \emptyset \to \text{term}(G)$. Notice that it is consistent with the fact that an RDF graph is a set (without duplicates).

In section 9.2, we consider operations between set of mappings. Those operations can be extended to bags, roughly speaking, making the operations not to discard duplicates. Formally, if Ω_1, Ω_2 are bags of mappings, then

$$\text{for } \mu \in \Omega_1 \bowtie \Omega_2, \text{card}_s(\mu)_{[\Omega_1 \bowtie \Omega_2]} = \textstyle\sum_{\mu=\mu_1 \cup \mu_2} \text{card}_s(\mu_1)_{[\Omega_1]} \cdot \text{card}_s(\mu_2)_{[\Omega_2]}$$
$$\text{for } \mu \in \Omega_1 \cup \Omega_2, \text{card}_s(\mu)_{[\Omega_1 \cup \Omega_2]} = \text{card}_s(\mu_1)_{[\Omega_1]} + \text{card}_s(\mu_2)_{[\Omega_2]}$$
$$\text{for } \mu \in \Omega_1 \setminus \Omega_2, \text{card}_s(\mu)_{[\Omega_1 \setminus \Omega_2]} = \text{card}_s(\mu_1)_{[\Omega_1]}$$

Definition 3. *Given a dataset D and a general graph pattern P composed from basic graph patterns possibly with blank nodes, we define the evaluation of P in D using a bag/multiset semantics, simply as defined in Section 9.2 but applying bag operators and with the base case as in Definition 2.*

Proposition 1. *Let P be a graph pattern without blank nodes and composed only by* AND, FILTER *and* OPT *operators, and let D be an RDF dataset with active graph G. Then every solution $\mu \in [\![P]\!]_G^D$ has cardinality 1.*

Note 7. The above proposition implies that in absence of blank nodes in graph patterns, duplicated solutions could be generated only by the use of UNION and GRAPH operators.

Definition 4 (Cardinality in SELECT **Result Form).** *Informally, given a query* (SELECT W, F, P)*, for bag/multiset semantics we simply take the projection of the solutions for P over variables W without discarding duplicates. Formally, given a query $Q =$ (SELECT W, F, P) and a mapping μ in the answer of Q in dataset D obtained from F, we define the cardinality of μ as*

$$\sum_{\mu'=\mu} \text{card}_s(\mu)_{[[\![P]\!]_G^D]}$$

10 Appendix III: Datalog

To make the paper self-contained, we will briefly review notions of Datalog. The reader can consult the books [1,5] for further details and proofs.

10.1 Syntax of Datalog.

A *term* is either a variable or a constant. A *predicate formula* is an expression $p(x_i, ..., x_n)$ where p is a predicate name and each x_i is a variable [8]. An *equality formula* is an expression $t_1 = t_2$ where t_1 and t_2 are terms. An *atom* is either a predicate or an equality formula. A *literal* is either an atom (a *positive literal* L) or the negation of an atom (a *negative literal* $\neg L$). A *rule* is an expression of the form $L \leftarrow L_1, \ldots, L_n$ where L is a predicate formula called the *head* of the rule and the sequence of literals $L_1, \ldots, L_n$ is called the *body* of the clause. Note that may assume that all heads of rules have only variables by adding the respective equality formula to its body. A *Datalog program* is a finite set of Datalog rules.

Let Π be a Datalog program. A rule having no variables is called a *ground rule*. A ground rule with empty body is called a *fact*. The set of facts occurring in Π, denoted facts(Π), is called the *initial database* of Π. A predicate is *extensional* if it occurs only in facts; otherwise it is called *intensional*.

A variable x occurs positively in a rule r if and only if x occurs in a positive literal L in the body of r such that: (1) L is a predicate formula; (2) if L is $x = c$ then c is a constant; (3) if L is $x = y$ or $y = x$ then y is a variable occurring positively in r. A Datalog rule r is said to be *safe* if all the variables occurring in the literals of r (including the head of r) occur positively in r. A Datalog program Π is *safe* if all the rules of Π are safe.

The *dependency graph* of a Datalog program Π is a digraph (N, E) where the set of nodes N is the set of predicates that occur in the literals of Π, and there is an arc (p_1, p_2) in E if there is a rule in Π whose body contains predicate p_1 and whose head contains predicate p_2. A Datalog program is said to be *recursive* if its dependency graph is cyclic, otherwise it is said to be *non-recursive*.

A *Datalog query* is a pair (Π, L) where Π is a Datalog program and L is a predicate formula $p(x_1, \ldots, x_n)$ called the goal literal.

In what follows, we will only consider non-recursive and safe Datalog programs (*nr-Datalog*$^\neg$).

10.2 Semantics of Datalog

A *substitution* θ is a set of assignments $\{x_1/t_1, \ldots, x_n/t_n\}$ where each x_i is a variable and each t_i is a term. Given a rule r, we denote by $\theta(r)$ the rule resulting from applying the substitution θ to the literals in r, i.e., the result of substituting the variable x_i for the term t_i in each literal of r. A substitution is *ground* if every term t_i is a constant.

[8] In this paper we assume that a predicate formula only contains variables, but in general it is also possible to have constants.

A rule r in a Datalog program Π is true with respect to a ground substitution θ, if for each literal L in the body of r one of the following conditions is satisfied:
(1) $\theta(L) \in \text{facts}(\Pi)$;
(2) $\theta(L)$ is an equality, $c = c$ where c is a constant;
(3) $\theta(L)$ is a literal of the form $\neg p(c_1, ..., c_n)$ and $p(c_1, ..., c_n) \notin \text{facts}(\Pi)$;
(4) $\theta(L)$ is a literal of the form $\neg(c_1 = c_2)$ and c_1 and c_2 are distinct constants.

The *meaning* of a Datalog program Π, denoted $\text{facts}^*(\Pi)$, is the database resulting from adding to the initial database of Π as many new facts of the form $\theta(L)$ as possible, where θ is a substitution that makes a rule r in Π true and L is the head of r. Then the rules are applied repeatedly and new facts are added to the database until this iteration stabilizes, i.e., until a *fixpoint* is reached. Note that for nr-Datalog$^\neg$ programs this naive algorithm works well. In fact for nr-Datalog$^\neg$ programs, all semantics coincide (the fixpoint, the answer set, and the well founded) [5].

Given a Datalog query $Q = (\Pi, L)$ and the initial database $D = \text{facts}(\Pi)$. The *answer* to Q over database D, denoted $\text{ans}_d(Q, D)$, is a set of substitutions defined as $\text{ans}_d(Q, D) = \{\theta \mid \theta(L) \in \text{facts}^*(\Pi)\}$.

10.3 Bag Semantics in Datalog

The cardinality of a substitution θ in a *bag of substitutions* Θ will be denoted by $\text{card}_d(\theta)_{[\Theta]}$ (or simply $\text{card}_d(\theta)$ when is clear from the context). If $\theta \notin \Theta$ then $\text{card}_d(\theta)_{[\Theta]} = 0$.

Let Π be a Datalog program. The cardinality of a substitution θ in a bag of solutions Θ that makes a rule r true in Π, is defined as

$$\text{card}_d(\theta)_{[\Theta]}^{\Pi(r)} = \text{card}_d(\theta)_{[\Theta_1]} \times \cdots \times \text{card}_d(\theta)_{[\Theta_n]}$$

where $\Theta_i = \text{ans}((\Pi, L_i), D)$ for each positive literal L_i in r.

The cardinality of a substitution θ in a bag of solutions Ω for a literal L in Π, denoted $\text{card}_d(\theta)_{[\Theta]}^{\Pi_L}$, is defined as follows:

- If L is an extensional literal, then $\text{card}_d(\theta)_{[\Theta]}^{\Pi_L}$ is the cardinality of $\theta(L)$ in $\text{facts}(\Pi)$, and 0 otherwise.
- If L is an intensional predicate, then:

$$\text{card}_d(\theta)_{[\Theta]}^{\Pi_L} = \sum \text{card}_d(\theta)_{[\Theta]}^{\Pi(r)}$$

  where r is a rule having literal L in its head.

Let $Q = (\Pi, L)$ be Datalog query and $D = \text{facts}(\Pi)$. The cardinality of a substitution θ in the bag of solutions $\Theta = \text{ans}_s(Q, D)$ is given by $\text{card}_d(\theta)_{[\Theta]}^{\Pi_L}$.