The multiset semantics of SPARQL patterns

Renzo Angles[1,3] and Claudio Gutierrez[2,3]

[1] Dept. of Computer Science, Universidad de Talca, Chile
[2] Dept. of Computer Science, Universidad de Chile, Chile
[3] Center for Semantic Web Research

Abstract. The paper determines the algebraic and logic structure of the multiset semantics of the core patterns of SPARQL. We prove that the fragment formed by AND, UNION, OPTIONAL, FILTER, MINUS and SELECT corresponds precisely to both, the intuitive multiset relational algebra (projection, selection, natural join, arithmetic union and except), and the multiset non-recursive Datalog with safe negation.

1 Introduction

The incorporation of multisets (also called "duplicates" or "bags")[4] into the semantics of query languages like SQL or SPARQL is essentially due to practical concerns: duplicate elimination is expensive and duplicates might be required for some applications, e.g. for aggregation. Although this design decision in SQL may be debatable (e.g. see [6]), today multisets are an established fact in database systems [8, 14].

The theory behind these query languages is relational algebra or equivalently, relational calculus, formalisms that for sets have a clean and intuitive semantics for users, developers and theoreticians [1]. The same cannot be said of their extensions to multisets, whose theory is complex (particular containment of queries) and their practical use not always clear for users and developers [8]. Worst, there exist several possible ways of extending set relational operators to multisets and one can find them in practice. As illustration, let us remind the behaviour of SQL relational operators. Consider as example the multisets $A = \{a, a, a, b, b, d\}$ and $B = \{a, b, b, c\}$. Then A `UNION ALL` $B = \{a, a, a, a, b, b, b, b, c, d\}$, that is, the "sum" of all the elements in both multisets (`UNION DISTINCT` is classical set union). A `INTERSECT ALL` B is $\{a, b, b\}$, i.e., the common elements in A and B, each with the minimum of the multiplicities in A and B. Regarding negation or difference, there are at least two: A `EXCEPT ALL` B is $\{a, a, d\}$, i.e. the arithmetical difference of the copies, and A `EXCEPT` B is $\{d\}$, the elements in A (with their multiplicity) after filtering out all elements occurring in B. The reader can imagine that the "rules" for combining these operators are not simple nor intuitive as they do not follow the rules of classical set operations.

[4] There is no agreement on terminology ([18], p. 27). In this paper we will use the word "multiset".

Is there a rationale behind the possible extensions? Not easy to tell. Early on Dayal et al. [7] observed that there are two conceptual approaches to extend the set operators of union, intersection and negation, corresponding to the two possible interpretations of multiple copies of a tuple. The first approach treats all copies of a given tuple as being identical or indistinguishable. The second one treats all copies of a tuple as being distinct, e.g., as having an underlying identity. Each of these interpretations gives rise to a different semantics for multisets. The first one permits to extend the lattice algebra structure of sets induced by the $\subseteq$-order by defining a multiset order $\subseteq_m$ defined as $A \subseteq_m B$ if each element in A is contained in B and its multiplicity in B is bigger than in A. This order gives a lattice meet (multiset intersection) defined as the elements c present in both multisets, and with multiplicity $\min(c_A, c_B)$, where c_A, c_B are the number of copies of c in A and B respectively. This is the `INTERSECT ALL` operator of SQL. The lattice join of two multisets gives a union defined as the elements c present in both multisets with multiplicity $\max(c_A, c_B)$. This operator is not present in SQL. As was shown by Albert [2], there is no natural negation to add to this lattice to get a Boolean algebra structure like in sets. The second interpretation (all copies of an element are distinct) gives a poor algebraic structure. The union gives in this case an arithmetic version, where the elements in the union of the multisets A and B are the elements c present in both multisets with $c_A + c_B$ copies. This is the `UNION ALL` operator in SQL. Under this interpretation, the intersection loses its meaning (always gives the empty set) and the difference becomes trivial ($A - B = A$).

In order to illustrate the difficulties of having a "coherent" group of operators for multisets, let us summarize the case of SQL, that does not have a clear rationale on this point.[5] We classified the operators under those that: keep the set semantics; preserve the lattice structure of multiset order; do arithmetic with multiplicities. Let A, B be multisets, and for each element c, let c_A and c_B be their respective multiplicities in A and B.

$$\text{union} : \begin{cases} \textit{set} & \texttt{UNION DISTINCT} \ \text{(multiplicity: 1)} \\ \textit{lattice} & \text{not present in SQL(*)} \ \text{(multiplicity: } \max(c_A, c_B)\text{)} \\ \textit{arithmetic} & \texttt{UNION ALL} \ \text{(multiplicity: } c_A + c_B\text{)} \end{cases}$$

$$\text{intersection} : \begin{cases} \textit{set} & \texttt{INTERSECT DISTINCT} \ \text{(multiplicity: 1)} \\ \textit{lattice} & \texttt{INTERSECT ALL} \ \text{(multiplicity: } \min(c_A, c_B)\text{)} \\ \textit{arithmetic} & \text{does not make sense} \end{cases}$$

$$\text{difference} : \begin{cases} \textit{set} & \text{not present in SQL}(**) \ \text{(multiplicity: 1)} \\ \textit{lattice} & \text{does not exists} \\ \textit{arithmetic} & \texttt{EXCEPT ALL} \ \text{(multiplicity: } \max(0, c_A - c_B)\text{)} \\ \textit{filter} & \texttt{EXCEPT} \ \text{(mult: if } (c_B = 0) \text{ then } c_A \text{ else } 0 \end{cases}$$

[5] We follow the semantics of ANSI and ISO SQL:1999 Database Language Standard.

(*) Simulated as `(A UNION ALL B) EXCEPT ALL (A INTERSECT ALL B)`.
(**) Simulated as `SELECT DISTINCT * FROM (A EXCEPT B)`.

At this point, a question arises: Are there "reasonable", "well behaved", "harmonic", groups of these operations for multisets? The answer is positive. Albert [2] proved that lattice union and lattice intersection plus a filter difference work well in certain domains. On the other hand, Dayal et al. [7] introduced the multiset versions for projection (π_X), selection (σ_C), join ($\bowtie$) and distinct (δ) and studied their interaction with Boolean operators. They showed that the lattice versions above combine well with selection ($\sigma_{P \vee Q}(r) = \sigma_P(r) \cup \sigma_Q(r)$ and similarly for intersection); that the arithmetic versions combine well with projection ($\pi_X(r \cup s) = \pi_X(r) \cup \pi_X(s)$). An important facet is the complexity introduced by the different operators. Libkin and Wong [16, 17] and Grumbach et al. [9] studied the expressive power and complexity of the operations of the fragment including lattice union and intersection; arithmetic difference; and distinct.

For our purposes here, namely the study of the semantics of multisets in SPARQL, none of the above fragments help. It turns out that is a formalism coming from a logical field, the well behaved fragment of *non-recursive Datalog with safe negation* (nr-Datalog$^\neg$), the one that matches the semantics of multisets in SPARQL. More precisely, the natural extension of the usual (set) semantics of Datalog to multisets developed by Mumick et al. [19]. In this paper we work out the relational counterpart of this fragment, using the framework defined by Dayal et al. [7], and come up with a *Multiset Relational Algebra* (MRA) that captures precisely the multiset semantics of the core relational patterns of SPARQL. MRA is based on the operators projection (π), selection (σ), natural join ($\bowtie$), union ($\cup$) and filter difference ($\setminus$). The identification of this algebra and the proof of the correspondence with the relational core of SPARQL are the main contributions of this paper. Not less important, as a side effect, this approach gives a new relational view of SPARQL (closer to classical relational algebra and hence more intuitive for people trained in SQL); allows to make a clean translation to a logical framework (Datalog); and matches precisely the fragment of SQL corresponding to it. Table 1 shows a glimpse of these correspondences, whose details are worked in this paper.

Contributions. Summarizing, this paper advances the current understanding of the SPARQL language by determining the precise algebraic (Multiset Relational Algebra) and logical (nr-Datalog$^\neg$) structure of the multiset semantics of the core pattern operators in the language. This contribution is relevant for users, developers and theoreticians. For *users*, it gives an intuitive and classic view of the relational core patterns of SPARQL, allowing a good understanding of how to use and combine the basic operators of the SPARQL language when dealing with multisets. For *developers*, helps to perform optimization, design extensions of the language, and understanding the semantics of multisets allowing for example translations from SPARQL operators to the right multiset operators of SQL and vice versa. For *theoreticians*, introduces a clean framework (Multiset Datalog as

Table 1. Schema of correspondences of multiset SPARQL patterns: with SPARQL algebra operators; Relational Multiset Algebra operators; Datalog rules; and SQL expressions. The operator `EXCEPT` in SPARQL is new (although expressible). The operator diff is a typed version of the diff SPARQL algebra operator, and $\setminus$ in MRA is the multiset filter difference.

SPARQL	Multiset Relational Algebra	nr-Datalog$^\neg$	SQL
`SELECT X ...`	$\pi_W(...)$	$q(X) \leftarrow L_1, \ldots, L_n$	`SELECT X ...`
`P FILTER C`	$\sigma_C(r)$	$L \leftarrow L_P, C$	`FROM r WHERE C`
`P1 . P2`	$r_1 \bowtie r_2$	$L \leftarrow L_1, L_2$	`r1 NATURAL JOIN r2`
`P1 UNION P2`	$r_1 \cup r_2$	$L \leftarrow L_1$ $L \leftarrow L_2$	`r1 UNION ALL r2`
`P1 EXCEPT P2`	$r_1 \setminus r_2$	$L \leftarrow L_1, \neg L_2$	`r1 EXCEPT r2`

defined by Mumick et al. [19]) to study from a formal point of view the multiset semantics of SPARQL patterns.

The paper is organized as follows. Section 2 presents the basic notions and notations used in the paper. Section 3 identifies a classical relational algebra view of SPARQL patterns, introducing the fragment SPARQL$^{\text{R}}$. Section 4 presents the equivalence between SPARQL$^{\text{R}}$ and multiset non-recursive Datalog with safe negation, and provides explicit transformations in both directions. Section 5 introduces the Multiset Relational Algebra, a simple and intuitive fragment of relational algebra with multiset semantics, and proves that it is exactly equivalent to multiset non-recursive Datalog with safe negation. Section 6 analyzes related work and presents brief conclusions.

2 SPARQL graph patterns

The definition of SPARQL graph patterns will be presented by using the formalism presented in [22], but in agreement with the W3C specifications of SPARQL 1.0 [25] and SPARQL 1.1 [10].

RDF graphs. Assume two disjoint infinite sets I and L, called IRIs and literals respectively.[6] An *RDF term* is an element in the set $T = I \cup L$. An *RDF triple* is a tuple $(v_1, v_2, v_3) \in I \times I \times T$ where v_1 is the *subject*, v_2 the *predicate* and v_3 the *object*. An *RDF Graph* (just graph from now on) is a set of RDF triples. The *union* of graphs, $G_1 \cup G_2$, is the set theoretical union of their sets of triples. Additionally, assume the existence of an infinite set V of variables disjoint from T. We will use $\text{var}(\alpha)$ to denote the set of variables occurring in the structure α.

[6] In addition to I and L, RDF and SPARQL consider a domain of anonymous resources called blank nodes. Their occurrence introduces issues that are not discussed in this paper. Based on the results in [11], we avoided blank nodes assuming that their absence does not affect the results presented in this paper.

A *solution mapping* (or just *mapping* from now on) is a partial function $\mu : V \to T$ where the domain of μ, $\text{dom}(\mu)$, is the subset of V where μ is defined. The *empty mapping*, denoted μ_0, is the mapping satisfying that $\text{dom}(\mu_0) = \emptyset$. Given $?X \in V$ and $c \in T$, we use $\mu(?X) = c$ to denote the solution mapping variable $?X$ to term c. Similarly, $\mu_{?X \to c}$ denotes a mapping μ satisfying that $\text{dom}(\mu) = \{?X\}$ and $\mu(?X) = c$. Given a finite set of variables $W \subset V$, the restriction of a mapping μ to W, denoted $\mu_{|W}$, is a mapping μ' satisfying that $\text{dom}(\mu') = \text{dom}(\mu) \cap W$ and $\mu'(?X) = \mu(?X)$ for every $?X \in \text{dom}(\mu) \cap W$. 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. Note that two mappings with disjoint domains are always compatible, and that the empty mapping μ_0 is compatible with any other mapping.

A *selection formula* is defined recursively as follows: (i) If $?X, ?Y \in V$ and $c \in I \cup L$ then $(?X = c)$, $(?X = ?Y)$ and $\text{bound}(?X)$ are atomic selection formulas; (ii) If F and F' are selection formulas then $(F \wedge F')$, $(F \vee F')$ and $\neg(F)$ are boolean selection formulas. The evaluation of a selection formula F under a mapping μ, denoted $\mu(F)$, is defined in a three-valued logic with values *true*, *false* and *error*. We say that μ satisfies F when $\mu(F) = \textit{true}$. The semantics of $\mu(F)$ is defined as follows:

- If F is $?X = c$ and $?X \in \text{dom}(\mu)$, then $\mu(F) = \textit{true}$ when $\mu(?X) = c$ and $\mu(F) = \textit{false}$ otherwise. If $?X \notin \text{dom}(\mu)$ then $\mu(F) = \textit{error}$.
- If F is $?X = ?Y$ and $?X, ?Y \in \text{dom}(\mu)$, then $\mu(F) = \textit{true}$ when $\mu(?X) = \mu(?Y)$ and $\mu(F) = \textit{false}$ otherwise. If either $?X \notin \text{dom}(\mu)$ or $?Y \notin \text{dom}(\mu)$ then $\mu(F) = \textit{error}$.
- If F is $\text{bound}(?X)$ and $?X \in \text{dom}(\mu)$ then $\mu(F) = \textit{true}$ else $\mu(F) = \textit{false}$.
- If F is a Boolean combination of the previous atomic cases, then it is evaluated following a three value logic table (see [25], 17.2).

Multisets. A *multiset* is an unordered collection in which each element may occur more than once. A multiset M will be represented as a set of pairs (t, j), each pair denoting an element t and the number j of times it occurs in the multiset (called multiplicity or cardinality). When $(t, j) \in M$ we will say that t j-belongs to M (intuitively "t has j copies in M"). To uniformize the notation and capture the corner cases, we will write $(t, *) \in M$ or simply say $t \in M$ when there are ≥ 1 copies of t in M. Similarly, when there is no occurrence of t in M, we will simply say "t does not belong to M", and abusing notation write $(t, 0) \in M$, or $(t, *) \notin M$. All of them indicate that t does not occur in M.

For *multisets* of solution mappings, following the notation of SPARQL, we will also use the symbol Ω to denote a multiset and $\text{card}(\mu, \Omega)$ to denote the cardinality of the mapping μ in the multiset Ω. In this sense, we use $(\mu, n) \in \Omega$ to denote that $\text{card}(\mu, \Omega) = n$, or simply $\mu \in \Omega$ when $\text{card}(\mu, \Omega) > 0$. Similarly, $\text{card}(\mu, \Omega) = 0$ when $\mu \notin \Omega$. The domain of a multiset Ω is defined as $\text{dom}(\Omega) = \bigcup_{\mu \in \Omega} \text{dom}(\mu)$.

SPARQL algebra. Let Ω_1, Ω_2 be multisets of mappings, W be a set of variables and F be a selection formula. The *SPARQL algebra for multisets of mappings* is

composed of the operations of projection, selection, join, union, minus, difference and left-join, defined respectively as follows:

- $\pi_W(\Omega_1) = \{\mu' \mid \mu \in \Omega_1, \mu' = \mu_{|W}\}$
  where $\text{card}(\mu', \pi_W(\Omega_1)) = \sum_{\mu'=\mu_{|W}} \text{card}(\mu, \Omega_1)$
- $\sigma_F(\Omega_1) = \{\mu \in \Omega_1 \mid \mu(F) = \mathit{true}\}$
  where $\text{card}(\mu, \sigma_F(\Omega_1)) = \text{card}(\mu, \Omega_1)$
- $\Omega_1 \bowtie \Omega_2 = \{\mu = (\mu_1 \cup \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\}$
  where $\text{card}(\mu, \Omega_1 \bowtie \Omega_2) = \sum_{\mu=(\mu_1 \cup \mu_2)} \text{card}(\mu_1, \Omega_1) \ \times \ \text{card}(\mu_2, \Omega_2)$
- $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
  where $\text{card}(\mu, \Omega_1 \cup \Omega_2) = \text{card}(\mu, \Omega_1) + \text{card}(\mu, \Omega_2)$
- $\Omega_1 - \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \nsim \mu_2 \vee \text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset\}$
  where $\text{card}(\mu_1, \Omega_1 - \Omega_2) = \text{card}(\mu_1, \Omega_1)$
- $\Omega_1 \setminus_F \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, (\mu_1 \nsim \mu_2) \vee (\mu_1 \sim \mu_2 \wedge (\mu_1 \cup \mu_2)(F) \neq \mathit{true})\}$
  where $\text{card}(\mu_1, \Omega_1 \setminus_F \Omega_2) = \text{card}(\mu_1, \Omega_1)$
- $\Omega_1 \text{⟕}_F \Omega_2 = \sigma_F(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus_F \Omega_2)$
  where $\text{card}(\mu, \Omega_1 \text{⟕}_F \Omega_2) = \text{card}(\mu, \sigma_F(\Omega_1 \bowtie \Omega_2)) + \text{card}(\mu, \Omega_1 \setminus_F \Omega_2)$

Syntax of graph patterns. A SPARQL *graph pattern* is defined recursively as follows: A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern called a *triple pattern*. [7] If P_1 and P_2 are graph patterns then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ UNION } P_2)$, $(P_1 \text{ OPT } P_2)$ and $(P_1 \text{ MINUS } P_2)$ are graph patterns. Also if C is a filter constraint (as defined below) then $(P_1 \text{ FILTER } C)$ is a graph pattern. And if W is a set of variables, $(\text{SELECT } W\, P_1)$ is a graph pattern.

A *filter constraint* is defined recursively as follows: (i) If $?X, ?Y \in V$ and $c \in I \cup L$ then $(?X = c)$, $(?X = ?Y)$ and $\text{bound}(?X)$ are *atomic filter constraints*; (ii) If C_1 and C_2 are filter constraints then $(!C_1)$, $(C_1 \mid\mid C_2)$ and $(C_1 \,\&\&\, C_2)$ are *complex filter constraints*. Given a filter constraint C, we denote by $f(C)$ the selection formula obtained from C. Note that there exists a simple and direct translation from filter constraints to selection formulas and vice versa.

Semantics of SPARQL graph patterns. The evaluation of a SPARQL graph pattern P over an RDF graph G is defined as a function $[\![P]\!]_G$ (or $[\![P]\!]$ where G is clear from the context) which returns a multiset of solution mappings. Let P_1, P_2, P_3 be graph patterns and C be a filter constraint. The evaluation of a graph pattern P over a graph G is defined recursively as follows:

1. If P is a triple pattern t, then $[\![P]\!]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \wedge \mu(t) \in G\}$ where $\mu(t)$ is the triple obtained by replacing the variables in t according to μ, and each mapping μ has cardinality 1.
2. $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
3. If P is $(P_1 \text{ OPT } P_2)$ then
   (a) if P_2 is $(P_3 \text{ FILTER } C)$ then $[\![P]\!]_G = [\![P_1]\!]_G \text{⟕}_C [\![P_3]\!]_G$
   (b) else $[\![P]\!]_G = [\![P_1]\!]_G \text{⟕}_{(\mathit{true})} [\![P_2]\!]_G$
4. $[\![(P_1 \text{ MINUS } P_2)]\!]_G = [\![P_1]\!]_G - [\![P_2]\!]_G$

[7] We assume that any triple pattern contains at least one variable.

5. $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$
6. $[\![(P_1 \text{ FILTER } C)]\!]_G = \sigma_{f(C)}([\![P_1]\!]_G)$
7. $[\![(\text{SELECT } W\, P_1)]\!]_G = \pi_W([\![P_1]\!]_G)$.

For the rest of the paper, we will call SPARQL$^{\text{W3C}}$ the fragment of graph patterns including the operators AND, UNION, OPT, FILTER, MINUS and SELECT, as defined above.

3 The relational fragment of SPARQL

In this section we will introduce a fragment of SPARQL which follows standard intuitions of the operators from relational algebra and SQL. We will prove that this fragment is equivalent to SPARQL$^{\text{W3C}}$. First, let us introduce the DIFF operator as an explicit way of expressing negation-by-failure[8] in SPARQL.

Definition 1 (The DIFF **operator).** *The weak difference of two graph patterns, P_1 and P_2, is defined as*

$$[\![(P_1 \text{ DIFF } P_2)]\!] = \{\mu_1 \in [\![P_1]\!] \mid \forall \mu_2 \in [\![P_2]\!], \mu_1 \nsim \mu_2\}$$

where $\text{card}(\mu_1, [\![(P_1 \text{ DIFF } P_2)]\!]) = \text{card}(\mu_1, [\![P_1]\!])$.

It is important to note that the DIFF operator is not defined in SPARQL 1.0 nor in SPARQL 1.1 at the syntax level. However, it can be implemented in current SPARQL engines by using the difference operator of the SPARQL$^{\text{W3C}}$ algebra. It was showed [4, 13] that the operators OPT and MINUS can be simulated with the operator DIFF in combination with AND, UNION and FILTER.

In order to facilitate, and make more natural the translation from SPARQL to Relational Algebra (and Datalog), we will introduce a more intuitive notion of difference between two graph patterns. We define the domain of a pattern P, denoted $\text{dom}(P)$, as the set of variables that occur (defining the output "schema") in the multiset of solution mappings for any evaluation of P.

Definition 2 (The EXCEPT **operator).** *Let P_1, P_2 be graph patterns satisfying* $\text{dom}(P_1) = \text{dom}(P_2)$. *The except difference of P_1 and P_2 is defined as*

$$[\![(P_1 \text{ EXCEPT } P_2)]\!] = \{\mu \in [\![P_1]\!] \mid \mu \notin [\![P_2]\!]\}$$

where $\text{card}(\mu, [\![(P_1 \text{ EXCEPT } P_2)]\!]) = \text{card}(\mu, [\![P_1]\!])$.

We will denote by EXCEPT* *(or outer EXCEPT) the version of this operation when the restriction on domains is not considered.*[9]

Note that the restriction on the domains of P_1 and P_2 follows the philosophy of classical relational algebra. But it can be proved that EXCEPT and its outer version are simulable each other:

[8] Recall that negation-by-failure can be expressed in SPARQL 1.0 as the combination of an optional graph pattern and a filter constraint containing the bound operator.

[9] This operation is called SetMinus in [12].

Lemma 1. *For each pair of graph patterns P_1, P_2 in SPARQL^{W3C}, and any RDF graph G, the operator* EXCEPT *can be simulated by* EXCEPT* *and vice versa.*

Proof. Clearly EXCEPT can be simulated by EXCEPT*. On the other direction, let us assume that $\mathrm{dom}(P_1) \neq \mathrm{dom}(P_2)$. Then $(P_1 \,\mathrm{EXCEPT}^*\, P_2)$ can be expressed by the pattern $(P_1' \,\mathrm{EXCEPT}\, P_2')$ where:
– $P_1' = (P_1 \,\mathrm{FILTER}(\neg\mathrm{bound}(?x_1) \;\&\&\; \ldots \;\&\&\; \neg\mathrm{bound}(?x_n)))$ when $\mathrm{dom}(P_1) \setminus \mathrm{dom}(P_2) = \{x_1, \ldots, x_n\}$ and $P_1' = P_1$ when $\mathrm{dom}(P_1) \setminus \mathrm{dom}(P_2) = \emptyset$; and
– $P_2' = (P_2 \,\mathrm{FILTER}(\neg\mathrm{bound}(?y_1) \;\&\&\; \ldots \;\&\&\; \neg\mathrm{bound}(?y_m)))$ when $\mathrm{dom}(P_2) \setminus \mathrm{dom}(P_1) = \{y_1, \ldots, y_m\}$ and $P_2' = P_2$ when $\mathrm{dom}(P_2) \setminus \mathrm{dom}(P_1) = \emptyset$.

Note that cardinalities of selected mappings are not touched.

The next lemma establishes the relationship between EXCEPT and DIFF, showing that EXCEPT can be simulated in SPARQL$^{\mathrm{W3C}}$.

Lemma 2. *For every pair of graph patterns P_1, P_2 in SPARQL^{W3C}, and any RDF graph G, the operator* EXCEPT *can be simulated by* DIFF *and vice versa.*

Proof. The high level proof goes as follows. EXCEPT, as we saw, is equivalent to EXCEPT*. And EXCEPT* differs from DIFF only in checking compatibility of mappings (i.e. $\sim$). $[\![P_1 \,\mathrm{EXCEPT}^*\, P_2]\!]$ eliminates from $[\![P_1]\!]$ those mappings in $[\![P_2]\!]$ that are equal to one in $[\![P_1]\!]$; while DIFF eliminates those that are compatible with one in $[\![P_1]\!]$. That is, the difference is between the multisets $\{(\mu_1, n_1) \in \Omega_1 \mid \neg\exists \mu_2 \in \Omega_2 \;\wedge\; \mu_1 = \mu_2\}$ versus $\{(\mu_1, n_1) \in \Omega_1 \mid \neg\exists \mu_2 \in \Omega_2 \;\wedge\; \mu_1 \sim \mu_2\}$. Now, for two mappings μ_1, μ_2, equality and compatibility ($\mu_1 = \mu_2$ versus $\mu_1 \sim \mu_2$) differ only in those variables that are bound in μ_1 and unbound in μ_2 or vice versa. Thus, to simulate $=$ with $\sim$ and vice versa, it is enough to have an operator that replaces all unbound entries in mappings of Ω_1 and Ω_2 by a fresh new constant, e.g. c, call the new sets Ω_1' and Ω_2', and we will have that $\{(\mu_1, n_1) \in \Omega_1 \mid \neg\exists \mu_2 \in \Omega_2 \;\wedge\; \mu_1 \sim \mu_2\}$ is equivalent to $\{(\mu_1, n_1) \in \Omega_1' \mid \neg\exists \mu_2 \in \Omega_2' \;\wedge\; \mu_1 = \mu_2\}$. Note that cardinalities are preserved because the change between "unbound" and "c" does not change them. The rest is to express the two operations on multisets of solution mappings: the one that fills in unbound entries with a fresh constant c; and the one that changes back the values c to unbound.

Now we are ready to state the main theorem. Define SPARQL$^{\mathrm{R}}$ as the fragment of SPARQL$^{\mathrm{W3C}}$ graph pattern expressions defined recursively by triple patterns plus the operators AND, UNION, FILTER and EXCEPT. Considering that DIFF is able to express OPT and MINUS (cf. [4, 13]), and that the DIFF operator is expressible in SPARQL$^{\mathrm{R}}$ (Lemma 2), we have the following result:

Theorem 1. *SPARQL^{R} is equivalent to SPARQL^{W3C}.*

For the rest of the paper, we will concentrate our interest on SPARQL$^{\mathrm{R}}$.

Note 1. An alternative proof of Theorem 1 is given as follows. (Compare [13], Lemma 12). Let θ be a function that renames variables by fresh ones.

SPARQL$^{\text{W3C}}$ contains SPARQL$^{\text{R}}$: The graph pattern $(P_1 \,\text{EXCEPT}\, P_2)$ can be rewritten into an equivalent pattern $(((P_1 \,\text{OPT}(\theta P_2)) \,\text{FILTER}\, C) \,\text{FILTER}\, C')$ where $\text{dom}(P_1) = \{?x_1, \ldots, ?x_n\}$, C is $(?x_1 = \theta?x_1 \;\&\&\ldots\&\&\; ?x_n = \theta?x_n)$ and C' is $(!\,\text{bound}(\theta?x_1))$.

SPARQL$^{\text{R}}$ contains SPARQL$^{\text{W3C}}$: The graph pattern $(P_1 \,\text{DIFF}\, P_2)$ can be rewritten into an equivalent graph pattern

$$(P_1 \,\text{EXCEPT}(\text{SELECT}\, W((P_1 \,\text{AND}\, P_1') \,\text{FILTER}\, C) \,\text{AND}\, P_2'))$$

where $W = \text{dom}(P_1) = \{?x_1, \ldots, ?x_n\}$, $P_1' = \theta(P_1)$, $P_2' = \theta(P_2)$ and C is $(?x_1 = \theta?x_1 \;\&\&\ldots\; ?x_n = \theta?x_n)$.

4 SPARQL$^{\text{R}}$ ≡ Multiset Datalog

In this section we prove that SPARQL$^{\text{R}}$ have the same expressive power of Multiset Datalog. Although the ideas of the proof are similar to those in [3] (now for SPARQL$^{\text{R}}$), we will sketch the main transformations to make the paper as self contained as possible. For notions of Datalog see Levene and Loizou [15], for the semantics of Multiset Datalog, Mumick et al. [19].

4.1 Multiset Datalog

A *term* is either a variable or a constant. A positive literal L is either a *predicate formula* $p(t_1, ..., t_n)$ where p is a predicate name and $t_1, ..., t_n$ are terms, or an *equality formula* $t_1 = t_2$ where t_1 and t_2 are terms. A negative literal $\neg L$ is the negation of a literal L. A *rule* is an expression of the form $L \leftarrow L_1 \wedge \cdots \wedge L_k \wedge \neg L_{k+1} \wedge \cdots \wedge \neg L_n$ where L is a positive literal called the *head* of the rule and the rest of literals (positive and negative) are called the *body*. A *fact* is a rule with empty body and no variables. A *Datalog program* Π is a finite set of rules and its set of facts is denoted facts(Π).

A variable x is *safe* in a rule r if it occurs in a positive predicate or in $x = c$ (c constant) or in $x = y$ where y is safe. A rule is safe it all its variables are safe. A program is *safe* if all its rules are safe. A program is non-recursive if its dependency graph is acyclic. In what follows, we only consider non-recursive and safe Datalog programs, denoted by nr-Datalog$^{\neg}$.

To incorporate multisets to the classical Datalog framework we will follow the approach introduced by Mumick and Shmueli [20]. The idea is rather intuitive: Each derivation tree gives rise to a substitution θ. In the standard (set) semantics, what matters is the set of the different substitutions that instantiates the distinguished literal. On the contrary, in multiset semantics the number of such instantiations also becomes relevant. As Mumick and Shmueli state [20, 19], "duplicate semantics of a program is obtained by counting the number of derivation trees". Thus now we have pairs (θ, n) of substitutions θ plus the number n of derivation trees that produce θ.

A Datalog query is a pair (Π, L) where Π is a program and L is a distinguished predicate (the goal) occurring as the head of a rule. The answer to (Π, L) is the multiset of substitutions θ such that makes $\theta(L)$ true.

Normalized Datalog. Let L, L_1, L_2 be literals. We assume, without loss of generality, that any safe non-recursive Datalog program can be normalized such that it just contains rules of the following types:

- (Projection rule) $L \leftarrow L_1$ where $\text{var}(L) \subset \text{var}(L_1)$;
- (Selection rule) $L \leftarrow L_1, EQ$ where EQ is a set of equalities of the form $x_i = x_j$ such that x_i, x_j are variables or constants.
- (Join rule) $L \leftarrow L_1, L_2$ where $\text{var}(L) \subseteq \text{var}(L_1) \cup \text{var}(L_2)$; and
- (Negation rule) $L \leftarrow L_1, \neg L_2$ where $\text{var}(L_2) \subseteq \text{var}(L_1)$ and $\text{var}(L) = \text{var}(L_1)$.

4.2 From SPARQL to Datalog

The algorithm that transforms SPARQL into Datalog includes transformations of RDF graphs to Datalog facts, SPARQL queries into a Datalog queries, and SPARQL mappings into Datalog substitutions.

RDF graphs to Datalog facts. Let G be an RDF graph: each term t in G is encoded by a fact $iri(t)$ or $literal(t)$ when t is an IRI or a literal respectively; the set of terms in G is defined by the rules $term(X) \leftarrow iri(X)$ and $term(X) \leftarrow literal(X)$; the fact $Null(null)$ encodes the *null* value (unbounded value); each RDF triple (v_1, v_2, v_3) in G is encoded by a fact $triple(v_1, v_2, v_3)$. Recall that we are assuming that an RDF graph is a "set" of triples.

SPARQL patterns into Datalog rules: The transformation follows essentially the idea presented by Polleres [23]. Let P be a graph pattern and G an RDF graph. Denote by $\delta(P)_G$ the function which transforms P into a set of Datalog rules. Table 2 shows the transformation rules defined by the function $\delta(P)_G$, where the notion of compatible mappings is implemented by the rules:

$comp(X, X, X) \leftarrow term(X)$, $comp(X, Y, X) \leftarrow term(X) \wedge Null(Y)$,
$comp(Y, X, X) \leftarrow Null(Y) \wedge term(X)$, $comp(X, X, X) \leftarrow Null(X)$.

Also, an atomic filter condition C is encoded by a literal L as follows (where $?X, ?Y \in V$ and $u \in I \cup L$): if C is either $(?X = u)$ or $(?X = ?Y)$ then L is C; if C is bound$(?X)$ then L is $\neg Null(?X)$.

SPARQL mappings to Datalog substitutions: Let P be a graph pattern, G an RDF graph and μ a solution mapping of P in G. Then μ gets transformed into a substitution θ satisfying that for each $x \in \text{var}(P)$ there exists $x/t \in \theta$ such that $t = \mu(x)$ when $\mu(x)$ is bounded and $t = null$ otherwise.

Now, the correspondence between the multiplicities of mappings and substitutions works as follows: Each SPARQL mapping comes from an evaluation tree. A *set* of evaluation trees becomes a *multiset* of mappings. Similarly, a *set* of Datalog derivation trees becomes a *multiset* of substitutions. Thus, each occurrence of a mapping μ comes from a SPARQL evaluation tree. This tree is translated by Table 2 to a Datalog derivation tree, giving rise to an occurrence of a substitution in Datalog. Each recursive step in Table 2 carries out bottom up the correspondence between cardinalities of mappings and substitutions.

Table 2. Transforming SPARQL$^{\mathrm{R}}$ graph patterns into Datalog Rules. The function $\delta(P)_G$ takes a graph pattern P and an RDF graph G, and returns a set of Datalog rules with main predicate $p(\overline{\mathrm{var}}(P))$, where $\overline{\mathrm{var}}(P)$ denotes the tuple of variables obtained from a lexicographical ordering of the variables in P. If L is a Datalog literal, then $\nu_j(L)$ denotes a copy of L with its variables renamed according to a variable renaming function $\nu_j : V \to V$. *comp* is a literal encoding the notion of compatible mappings. *cond* is a literal encoding a filter condition C. $\overline{W}$ is a subset of $\overline{\mathrm{var}}(P_1)$.

Pattern P	$\delta(P)_G$
(x_1, x_2, x_3)	$p(\overline{\mathrm{var}}(P)) \leftarrow \mathit{triple}(x_1, x_2, x_3)$
$(P_1 \text{ AND } P_2)$	$p(\overline{\mathrm{var}}(P)) \leftarrow \nu_1(p_1(\overline{\mathrm{var}}(P_1))) \ \wedge \ \nu_2(p_2(\overline{\mathrm{var}}(P_2)))$ $\bigwedge_{x \in \mathrm{var}(P_1) \cap \mathrm{var}(P_2)} \mathit{comp}(\nu_1(x), \nu_2(x), x),$ $\delta(P_1)_G \ , \ \delta(P_2)_G$
	$\mathrm{dom}(\nu_1) = \mathrm{dom}(\nu_2) = \mathrm{var}(P_1) \cap \mathrm{var}(P_2),\ \mathrm{range}(\nu_1) \cap \mathrm{range}(\nu_2) = \emptyset.$
$(P_1 \text{ UNION } P_2)$	$p(\overline{\mathrm{var}}(P)) \leftarrow p_1(\overline{\mathrm{var}}(P_1)) \bigwedge_{x \in \mathrm{var}(P_2) \setminus \mathrm{var}(P_1)} \mathit{Null}(x),$ $p(\overline{\mathrm{var}}(P)) \leftarrow p_2(\overline{\mathrm{var}}(P_2)) \bigwedge_{x \in \mathrm{var}(P_1) \setminus \mathrm{var}(P_2)} \mathit{Null}(x),$ $\delta(P_1)_G \ , \ \delta(P_2)_G$
$(P_1 \text{ EXCEPT } P_2)$	$p(\overline{\mathrm{var}}(P_1)) \leftarrow p_1(\overline{\mathrm{var}}(P_1)) \wedge \neg p_2(\overline{\mathrm{var}}(P_2)),$ $\delta(P_1)_G \ , \ \delta(P_2)_G$
$(\text{SELECT } W P_1)$	$p(\overline{W}) \leftarrow p_1(\overline{\mathrm{var}}(P_1)),$ $\delta(P_1)_G$
$(P_1 \text{ FILTER } C)$ and C is atomic	$p(\overline{\mathrm{var}}(P)) \leftarrow p_1(\overline{\mathrm{var}}(P_1)) \wedge \mathit{cond}$ $\delta(P_1)_G$

Thus we have that a SPARQL query $Q = (P, G)$ where P is a graph pattern and G is an RDF graph gets transformed into the Datalog query $(\Pi, p(\overline{\mathrm{var}}(P)))$ where Π is the Datalog program $\delta(P)_G$ plus the facts got from the transformation of the graph G, and p is the goal literal related to P.

4.3 From Datalog to SPARQL

Now we need to transform Datalog facts into RDF data, Datalog substitutions into SPARQL mappings, and Datalog queries into SPARQL queries.

Datalog facts as an RDF Graph: Given a Datalog fact $f = p(c_1, ..., c_n)$, consider the function $\mathrm{desc}(f)$ which returns the set of triples

$$\{(u, \mathrm{predicate}, p), (u, \mathrm{rdf{:}\_1}, c_1), \ldots, (u, \mathrm{rdf{:}\_n}, c_n)\},$$

where u is a fresh IRI. Given a set of Datalog facts F, the RDF description of F will be the graph $G = \bigcup_{f \in F} \mathrm{desc}(f)$.

Datalog rules as SPARQL graph patterns: Let Π be a (normalized) Datalog program and L be a literal $p(x_1, \ldots, x_n)$ where p is a predicate in Π and each x_i is a variable. We define the function $\mathrm{gp}(L)_\Pi$ which returns a graph pattern encoding of the program (Π, L). The translation works intuitively as follows:

(a) If predicate p is extensional, then $\mathrm{gp}(L)_{\Pi}$ returns the graph pattern $((?Y, \text{predicate}, p)\ \mathrm{AND}(?Y, \text{rdf:\_1}, x_1)\ \mathrm{AND} \cdots \mathrm{AND}(?Y, \text{rdf\_n}, x_n))$, where $?Y$ is a fresh variable.

(b) If predicate p is intensional and $\{r_1, \ldots, r_n\}$ is the set of all the rules in Π where p occurs in the head, then $\mathrm{gp}(L)_{\Pi}$ returns the graph pattern $(\ldots(T(r_1)\,\mathrm{UNION}\,T(r_2))\ldots\mathrm{UNION}\,T(r_n))$ where $T(r_i)$ is defined as follows (when $n = 1$ the resulting graph pattern is reduced to $T(r_1)$):

- If r_i is $L \leftarrow L_1$ then $T(r_i)$ returns $\mathrm{SELECT}\,x_1, \ldots, x_n\,\mathrm{WHERE}\,\mathrm{gp}(L_1)_{\Pi}$.
- If r_i is $L \leftarrow L_1 \wedge EQ$, where EQ is a set of equalities of the form $x_i = x_j$ such that x_i, x_j are variables or constants, then $T(r_i)$ returns $(\mathrm{gp}(L_1)_{\Pi}\,\mathrm{FILTER}\,C)$ where C is a filter condition equivalent to EQ.
- If r_i is $L \leftarrow L_1 \wedge L_2$ then $T(r_i)$ returns $(\mathrm{gp}(L_1)_{\Pi}\,\mathrm{AND}\,\mathrm{gp}(L_2)_{\Pi})$.
- If r_i is $L \leftarrow L_1 \wedge \neg L_2$ then $T(r_i)$ returns $(\mathrm{gp}(L_1)_{\Pi}\,\mathrm{EXCEPT}^*\,\mathrm{gp}(L_2)_{\Pi})$.

Datalog substitutions as SPARQL mappings: For each substitution θ satisfying (Π, L) build a mapping μ satisfying that, if $x/t \in \theta$ then $x \in \mathrm{dom}(\mu)$ and $\mu(x) = t$. The correspondence of multiplicities work in a similar way (via derivation tree to evaluation tree) as in the case of mappings to substitutions.

Putting together the transformation in Table 2 and the pattern obtained by using $\mathrm{gp}(L)_{\Pi}$, we get the following theorem, whose proof is a long but straightforward induction on the structure of the patterns in one direction, and on the level of Datalog in the other.

Theorem 2. *Multiset nr-Datalog$^{\neg}$ has the same expressive power as SPARQL^{R}.*

5 The relational version of Multiset Datalog: MRA

In this section we introduce a multiset relational algebra (called MRA), counterpart of Multiset Datalog, and prove its equivalence with the fragment of non-recursive Datalog with safe negation.

5.1 Multiset Relational Algebra (MRA)

Multiset relational algebra is an extension of classical relation algebra having multisets of relations instead of sets of relations. As indicated in the introduction, there are manifold approaches and operators to extend set relational algebra with multisets. We use the semantics of multiset operators defined by Dayal et al. [7] for the operations of selection, projection, natural join and arithmetic union; and add filter difference (not present there) represented by the operator "except".

Let us formalize these notions. In classical *(Set) relational algebra*, a database schema is a set of relational schemas. A relational schema is defined as a set of attributes. Each attribute A has a domain, denoted $\mathrm{dom}(A)$. A *relation* R over the relational schema $S = \{A_1, \ldots, A_n\}$ is a finite *set* of tuples. An instance r of a schema S is a relation over S. Given an instance r of a relation R with schema S, $A_j \in S$ and $t = (a_1, \ldots, a_n) \in r$, we denote by $t[A_j]$ the tuple (a_j). Similarly with $t[X]$ when $X \subseteq S$ and we will define $t[\emptyset] = \emptyset$.

In the *Multiset relational algebra* setting, an instance of a schema is a *multiset relation*, that is, a set of pairs (t, i), where t is a tuple over the schema S, and $i \geq 1$ is a positive integer. (For notions and notations on multisets recall section 2, *Multisets*).

Definition 3 (Multiset Relational Algebra (MRA)). *Let r and r' be multiset relations over the schemas S and S' respectively. Let $A \in S$ be an attribute, $a \in \mathrm{dom}(A)$ and $I = S \cap S'$. MRA consists of the following operations:*

1. *Selection.* $\sigma_{A=a}(r) = \{(t, i) : (t, i) \in r \wedge t[A] = a\}$.
2. *Natural Join.* $r \bowtie r'$ *is a multiset relation over* $S \cup S'$ *defined as follows. Let* $S'' = S' - S$. *Let* $t^\frown t'$ *denotes concatenation of tuples. Then*

$$r \bowtie r' = \{(t^\frown(t'[S'']), i \times j) : (t, i) \in r \wedge (t', j) \in r' \wedge t[I] = t'[I]\}.$$

3. *Projection. Let* $X \subseteq S$. *Then:*

$$\pi_X(r) = \{(t[X], \textstyle\sum_{(t_j, n_j) \in r \ s.t. \ t_j[X]=t} n_j) : (t, *) \in r\}.$$

4. *Union. Assume* $S = S'$.

$$\begin{aligned} r \cup r' = &\{(t, i) : \ t \ i\text{-belongs to } r \text{ and } t \notin r' \ \} \\ &\cup \{(t', j) : \ t' \notin r \text{ and } t' \ j\text{-belongs to } r' \ \} \\ &\cup \{(t, i + j) : \ t \ i\text{-belongs to } r \text{ and } t \ j\text{-belongs to } r' \ \}. \end{aligned}$$

5. *Except. Assume* $S = S'$.

$$r \setminus r' = \{(t, i) \in r : (t, *) \notin r'\}.$$

As usual, we will define a query *in this multiset relational algebra as an expression over an extended domain which includes, besides the original domains of the schemas, a set of variables V.*

5.2 MRA $\equiv$ Multiset nr-Datalog$^\neg$

This subsection is devoted to prove the following result.

Theorem 3. *Multiset relational algebra (***MRA***) has the same expressive power as Multiset Non-recursive Datalog with safe negation.*

From this theorem and Theorem 2 it follows:

Corollary 1. *SPARQL^R is equivalent to MRA.*

Proof. The proof is based on the ideas of the proof of Theorem 3.18 in [15], extended to multisets. Let E be a relational algebra query expression over the schema R and D a database. Then it will be translated by a function $(\cdot)^{\Pi}$ to the Datalog program $\mathrm{facts}(\Pi) \cup E^{\Pi}$, where $\mathrm{facts}(\Pi)$ is the multiset of facts (over fresh predicates r^{Π} for each relation r, and having the same arity as the original schema of r):

$\mathrm{facts}(\Pi) = \{(r^{\Pi}(t), n) : \ t$ is a tuple with multiplicity n in schema r in $D \ \}$, and E^{Π} is the translation of the expression E given by the recursive specification below. For the expression E_j, the set V_j will denote its list of attributes.

1. Base case. No operator involved. Thus the query is a member of the schema R, namely $r(x_1, \ldots, x_n)$. The corresponding Multiset Datalog query is: $out_r(x_1, \ldots, x_n) \leftarrow r^{\Pi}(x_1, \ldots, x_n)$
2. $E = \sigma_C(E_1)$, where C is a set of equalities of the form $x_i = x_j$ where x_i, x_j are variables or constants. The translation E^{Π} is the program: $out_E(x_1, \ldots, x_k) \leftarrow E_1^{\Pi}(x_1, \ldots, x_k) \wedge C$
3. $E = E_1 \bowtie E_2$. Let $V = V_2 \setminus V_1$. The translation is: $out_E(V_1, V) \leftarrow E_1^{\Pi}(V_1) \wedge E_2^{\Pi}(V_2)$
4. $E = \pi_A(E_1)$, where A is a sublist of the attributes in E_1. The translation is: $out_E(A) \leftarrow E_1^{\Pi}(V_1)$.
5. $E = E_1 \cup E_2$, where E_1 and E_2 have the same schema. The translation is: $out_E(x_1, \ldots, x_k) \leftarrow E_1^{\Pi}(x_1, \ldots, x_k)$ $out_E(x_1, \ldots, x_k) \leftarrow E_2^{\Pi}(x_1, \ldots, x_k)$
6. $E = E_1 \setminus E_2$, where E_1 and E_2 have the same schema. The translation is: $out_E(x_1, \ldots, x_k) \leftarrow E_1^{\Pi}(x_1, \ldots, x_k) \wedge \neg E_2^{\Pi}(x_1, \ldots, x_k)$

It is important to check that the resulting program is non-recursive (this is because the structure of the algebraic relational expression from where it comes is a tree). Also it is safe because in rule (6) both expressions have the same schema). Now, it needs to be shown that for each relational expression (query) E in R, $[E]_D$ and $[E^{\Pi}]$ return the same "tuples" with the same multiplicity. This is done by induction on the structure of E.

Now, let us present the transformation from Multiset Datalog to Multiset Relational Algebra. Note that we may assume a normal form for the Datalog programs as presented in Section 4.1. Then the recursive translation $(\cdot)^R$ from Datalog programs to MRA expressions goes as follows.

1. First translate those head predicates q occurring in ≥ 2 rules as follows. Let q be the head of rules $r_1, \ldots, r_k$, $k \geq 2$. Rename each such head q with the same set of variables V. Then the translation is $(q)^R = (q_{r_1})^R \cup \cdots \cup (q_{r_k})^R$. From now on, we can assume that, not considering these q's, all other predicates occur as head in at most one rule. Hence we will not need the subindex indicating the rule to which they belong to.
2. (Base case.) Let r be a fact $q(V)$. Then translates it as $(q_r)^R = q^R(V)$, where q^R is a fresh new schema with the corresponding arity.
3. Let r be $q(A) \leftarrow p(V)$, where A is a sublist of V. The translation is $(q_r)^R = \pi_A((p)^R)$.
4. Let r be $q(V) \leftarrow p(V) \wedge C$, where C is a set of equalities $x_i = x_j$ such that x_i, x_j are variables or constants. The translation is $(q_r)^R = \sigma_C((p)^R)$.
5. Let r be $q(X, Y, Z) \leftarrow p_1(X, Y) \wedge p_2(Y, Z)$, where X, Y, Z are disjoint lists of variables. The translation is $(q_r)^R = (p_1)^R \bowtie (p_2)^R$.
6. Let r be $q(X, Y) \leftarrow p_1(X, Y) \wedge \neg p_2(Y)$, that is the rule is safe. The translation is $(q_r)^R = (p_1)^R \setminus ((p_1)^R \bowtie (p_2)^R)$.

The arguments about multiplicity are straightforward verifications. And because the program Π is non-recursive (i.e. its dependency graph is acyclic), the recursive translation to the relational expression gives a well formed algebraic expression.

6 Related Work and Conclusions

To the best of our knowledge, the multiset semantics of SPARQL has not been systematically addressed. There are works that, when studying the expressive power of SPARQL, touched some aspects of this topic. Cyganiak [5] was among the first who gave a translation of a core fragment of SPARQL into relational algebra. Polleres [23] proved the inclusion of the fragment of SPARQL patterns with safe filters into Datalog by giving a precise and correct set of rules. Schenk [26] proposed a formal semantics for SPARQL based on Datalog, but concentrated on complexity more than expressiveness issues. Both, Polleres and Schenk do not consider multiset semantics of SPARQL in their translations. Perez et al. [21] gave the first formal treatment of multiset semantics for SPARQL. Angles and Gutierrez [3], Polleres [24] and Schmidt et al. [27] extended the set semantics to multiset semantics using this idea. Kaminski et al [12] considered multisets in subqueries and aggregates in SPARQL. In none of these works was addressed the goal of characterizing the multiset algebraic and/or logical structure of the operators in SPARQL.

We studied the multiset semantics of the core SPARQL patterns, in order to shed light on the algebraic and logic structure of them. In this regard, the discovery that the core fragment of SPARQL patterns matches precisely the multiset semantics of Datalog as defined by Mumick et al. [19] and that this logical structure corresponds to a simple multiset algebra, namely the Multiset Relational Algebra (MRA), builds a nice parallel to that of classical set relational algebra and relational calculus. Contrary to the rather chaotic variety of multiset operators in SQL, it is interesting to observe that in SPARQL there is a coherent body of multiset operators. We think that this should be considered by designers in order to try to keep this clean design in future extensions of SPARQL.

Last, but not least, this study shows the complexities and challenges that the introduction of multisets brings to query languages, exemplified here in the case of SPARQL.

Acknowledgments. The authors have funding from Millennium Nucleus Center for Semantic Web Research under Grant NC120004. The authors thank useful feedback from O. Hartig and anonymous reviewers.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Albert, J.: Algebraic properties of bag data types. In: Proc. of the Int. Conference on Very Large Data Bases (VLDB). pp. 211–219 (1991)
3. Angles, R., Gutierrez, C.: The Expressive Power of SPARQL. In: Proc. of the Int. Semantic Web Conference (ISWC). pp. 114–129 (2008)
4. Angles, R., Gutierrez, C.: Negation in SPARQL. In: Alberto Mendelzon Int. Workshop on Foundations of Data Management (AMW) (2016)

5. Cyganiak, R.: A relational algebra for SPARQL. Tech. Rep. HPL-2005-170, HP Labs (2005)
6. Date, C.J.: Date on Database: Writings 2000-2006. APress (2006)
7. Dayal, U., Goodman, N., Katz, R.H.: An extended relational algebra with control over duplicate elimination. In: Proc. of the Symposium on Principles of Database Systems (PODS). pp. 117–123. ACM (1982)
8. Green, T.J.: Bag semantics. In: Encyclopedia of Database Systems, pp. 201–206 (2009)
9. Grumbach, S., Libkin, L., Milo, T., Wong, L.: Query languages for bags: expressive power and complexity. SIGACT News 27(2), 30–44 (1996)
10. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language - W3C Recommendation. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/ (March 21 2013)
11. Hogan, A., Arenas, M., Mallea, A., Polleres, A.: Everything you always wanted to know about blank nodes. Journal of Web Semantics 27(1) (2014)
12. Kaminski, M., Kostylev, E.V., Grau, B.C.: Semantics and Expressive Power of Subqueries and Aggregates in SPARQL 1.1. In: Proceedings of the Int. Conference on World Wide Web (WWW). pp. 227–238. ACM (2016)
13. Kontchakov, R., Kostylev, E.V.: On Expressibility of Non-Monotone Operators in SPARQL. In: Int. Conference on the Principles of Knowledge Representation and Reasoning (2016)
14. Lamperti, G., Melchiori, M., Zanella, M.: On multisets in database systems. In: Proceedings of the Workshop on Multiset Processing. pp. 147–216 (2001)
15. Levene, M., Loizou, G.: A Guided Tour of Relational Databases and Beyond. Springer-Verlag (1999)
16. Libkin, L., Wong, L.: Some properties of query languages for bags. In: Proc. of the Int. Workshop on Database Programming Languages (DBPL) - Object Models and Languages. pp. 97–114 (1994)
17. Libkin, L., Wong, L.: Query languages for bags and aggregate functions. Journal of Computer and System Sciences 55(2), 241–272 (1997)
18. Melton, J., Simon, A.R.: SQL:1999. Understanding Relational Language Components. Morgan Kaufmann Publ. (2002)
19. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In: Proc. of the Int. Conference on Very Large Data Bases (VLDB). pp. 264–277 (1990)
20. Mumick, I.S., Shmueli, O.: Finiteness Properties of Database Queries. In: Australian Database Conference. pp. 274–288 (1993)
21. Pérez, J., Arenas, M., Gutierrez, C.: Semantics of SPARQL. Tech. Rep. TR/DCC-2006-17, Department of Computer Science, University of Chile (2006)
22. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems (TODS) 34(3), 1–45 (2009)
23. Polleres, A.: From SPARQL to Rules (and back). In: Proceedings of the 16th Int. World Wide Web Conference (WWW). pp. 787–796. ACM (2007)
24. Polleres, A.: How (well) Do Datalog, SPARQL and RIF Interplay? In: Proc. of the Int. conference on Datalog in Academia and Industry. pp. 27–30 (2012)
25. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/2008/REC-115-sparql-query-20080115/ (January 15 2008)
26. Schenk, S.: A SPARQL Semantics Based on Datalog. In: Annual German Conference on Advances in Artificial Intelligence. vol. 4667, pp. 160–174 (2007)
27. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proc. of the Int. Conference on Database Theory. pp. 4–33. ACM (2010)