Query Nesting, Assignment, and Aggregation in SPARQL 1.1

MARK KAMINSKI, EGOR V. KOSTYLEV, and BERNARDO CUENCA GRAU,
University of Oxford

Answering aggregate queries is a key requirement of emerging applications of Semantic Technologies, such as data warehousing, business intelligence, and sensor networks. To fulfil the requirements of such applications, the standardization of SPARQL 1.1 led to the introduction of a wide range of constructs that enable value computation, aggregation, and query nesting. In this article, we provide an in-depth formal analysis of the semantics and expressive power of these new constructs as defined in the SPARQL 1.1 specification, and hence lay the necessary foundations for the development of robust, scalable, and extensible query engines supporting complex numerical and analytics tasks.

CCS Concepts: • **Information systems → Query languages for non-relational engines**; **Resource Description Framework (RDF)**;

Additional Key Words and Phrases: SPARQL

ACM Reference format:
Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau. 2017. Query Nesting, Assignment, and Aggregation in SPARQL 1.1. *ACM Trans. Database Syst.* 42, 3, Article 17 (August 2017), 46 pages.
https://doi.org/10.1145/3083898

1 INTRODUCTION

The Resource Description Framework (RDF) is the World Wide Web Consortium (W3C) standard language for representing information in the Semantic Web [49]. RDF has been adopted in many applications as a convenient format for storing semistructured data. Moreover, the advent of major initiatives such as Linked Open Data [12] have led to a constant growth in the availability of RDF data online. RDF is based on a directed graph data model, where both nodes and edges are labelled. An RDF graph is a set of triples of the form (s, p, o), which can be interpreted as edges labelled by p (the *predicate*) from nodes labelled by s (the *subject*) to nodes labelled by o (the *object*). The elements of a triple are typically *Internationalised Resource Identifiers (IRIs)*—global names that uniquely identify resources on the Web.

SPARQL, which became a W3C recommendation in 2008, is the standard query language for RDF [52]. Since its standardisation, the study of SPARQL as a database query language has attracted significant attention [43, 50]. A key distinguishing feature of SPARQL over previous RDF query languages is that it comes with a well-defined algebraic semantics, which has been the subject of

This work was supported by the Royal Society and the EPSRC projects MaSI^3, Score!, DBOnto, and ED^3.
Authors' address: Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, OXFORD, OX1 3QD, UK; emails: {mark.kaminski, egor.kostylev, bernardo.cuenca.grau}@cs.ox.ac.uk.

https://doi.org/10.1145/3083898

intensive research and has laid the foundations for subsequent implementations [3, 41, 43, 50, 51, 53, 56]. The basic element of a SPARQL algebra query is a *basic graph pattern*, which is essentially an RDF graph where triples may also contain variables; the evaluation of a graph pattern over an RDF graph then consists of the mappings that homomorphically embed the pattern into the graph. Graph patterns can be combined with each other as well as with expressions using a rich set of algebraic operators.

Despite being a powerful query language, the first version of the SPARQL standard provided no support for queries that, rather than simply retrieving data, involve some form of computation or summarisation. This significantly limited the applicability of the language. Indeed, answering such queries is a key requirement in data warehousing and business intelligence, where data is aggregated across dimensions looking for patterns [1, 10, 22–24, 32, 58], as well as in emerging applications involving sensor networks and streaming RDF data [6, 11, 15].

The standardisation of a new version of the SPARQL query language, called SPARQL 1.1 [29], addressed these limitations by introducing a wide range of constructs supporting aggregation and value computation in line with those available in SQL:

- a collection of *aggregate functions* for value computation, such as Min, Max, Avg, Sum, and Count;
- the *grouping* constructs `GROUP BY` and `HAVING`, which restrict the application of aggregate functions to groups of solutions satisfying certain conditions;
- the *variable assignment* constructs `BIND`, `VALUES`, and `AS`, which are used to assign the value of a complex (e.g., arithmetic) expression to a variable; and
- a *query nesting* mechanism for embedding queries within graph patterns as well as within expressions.

Similarly to its predecessor, SPARQL 1.0, the semantics of SPARQL 1.1 is specified by means of an (extended) normative algebra. Many of the new features introduced in SPARQL 1.1, such as property paths [7, 40, 48, 57], query federation [13, 14], or entailment regimes [2, 8, 38, 39], have already received significant attention in the literature. In contrast, the theoretical properties of the algebraic operators that enable value computation, aggregation, and query nesting remain largely unexplored [4]. This is in stark contrast to the case of relational databases, where the formal properties of aggregation have been studied in depth [17–21, 30, 36, 44, 45, 54].

Our aim in this article is to provide a systematic study of the semantics and expressive power of the SPARQL 1.1 algebra with variable assignment, aggregates, and query nesting. Understanding the capabilities of the new constructs and their inter-dependencies is a key requirement for the development of query engines supporting complex numerical and analytics tasks, while providing correctness, robustness, scalability, and extensibility guarantees.

We take the well-known SPARQL algebra [50] as a starting point. Most existing works on SPARQL, including [50], assume that graph patterns are interpreted as sets of solution mappings rather than multisets (or bags) as in the normative specification. In this article, we follow the normative specification and adopt multiset semantics from the outset, which we formalise in Section 2.

In Sections 3 and 4, we study the query nesting mechanisms in SPARQL 1.1. We first consider, in Section 3, the nesting of `SELECT` and `SELECT DISTINCT` query blocks. In algebraic terms, this amounts to allowing the unrestricted use of operators *Project* and *Distinct* rather than limiting them to the outermost query level as in SPARQL 1.0. We show that there is no gain of expressive power by allowing unrestricted use of just one of these operators. In contrast, if *both* operators are allowed without restrictions, we show how to construct queries that cannot be equivalently expressed without nesting. The additional expressive power is due to the interplay between the set

semantics enforced by the *Distinct* operator and the bag semantics of *Project*—a phenomenon that was first observed in the relational case by Cohen [18], and was later conjectured by Angles and Gutierrez to yield additional expressive power in SPARQL [4]. As argued in Section 3.2, however, the evidence given in [4] in support of their conjecture is unsatisfactory. Our results settle this question and provide a detailed account of which combinations of constructs lead to expressivity gains and which ones are redundant.

In addition to subqueries as patterns, SPARQL 1.1 also provides a mechanism for embedding graph patterns within expressions in filter conditions using the exists construct. We investigate this form of query nesting in Section 4. We first argue that the normative semantics of queries with exists expressions is not always well-defined. Thus, we propose a new semantics, which is defined for all queries. Our proposed semantics is a conservative extension of the normative one, in the sense that they coincide whenever the normative semantics is well-defined. We finally show that the exists construct can be simulated, thus not resulting in any additional expressive power.

In Section 5, we turn our attention to variable assignment. It is enabled in the normative algebra by the *Extend* operator, which extends solution mappings with a fresh variable assigned to the value of an expression. We show that *Extend* increases expressive power, as it introduces arithmetic into the language [54].

In Section 6, we analyse the SPARQL 1.1 aggregate algebra. It turns out that this algebra is rather non-standard when compared to its relational counterpart. It provides a great deal of power and flexibility, and we show that it can express all forms of query nesting and variable assignment previously described. We then define a normal form, which we exploit to define a substantial simplification of the normative aggregate algebra that eliminates most of its unconventional aspects. The resulting algebra is much closer to its relational counterpart and can be exploited to provide a more transparent algebraic translation of the SPARQL 1.1 syntax.

In Section 7, we study property paths, which bring a limited form of recursion into the language, and discuss their interplay with aggregation and arithmetic. We first demonstrate that the combination of aggregation and property paths brings additional expressive power into the language, as it allows us to construct queries that cannot be captured if either of these features is disallowed. We then show that the normative multiset semantics of property paths can be equivalently recast in terms of sets.

In Section 8, we provide a semantics for online analytical processing (OLAP) queries (such as cube and window-based queries) in terms of our simplified aggregate algebra. We also discuss OLAP queries in the specific case when RDF data conforms to the W3C Data Cube specification [23].

Finally, in Section 9, we revisit the simplifying assumptions adopted in our formal presentation with respect to the standard and discuss their implications.

The article is accompanied with an online Appendix, accessible in the ACM Digital Library, which contains the more technical aspects of certain proofs.

2 THE SPARQL ALGEBRA

We assume familiarity with the normative SPARQL syntax, which we will use for writing example queries. In this section, we recapitulate the core fragment of the SPARQL 1.1 algebra capturing SPARQL 1.0, which we refer to as Sparql; in subsequent sections we gradually introduce and study other algebraic constructs. We also recapitulate here basic notions on RDF, query equivalence, and expressive power. In contrast to most articles in the literature, we follow the W3C standard by using the ternary left join (`OPTIONAL`) operator and by defining the semantics of the algebra in terms of bags rather than sets.

2.1 RDF Graphs

Let **I**, **L**, and **B** be countably infinite pairwise disjoint sets of *IRIs*, *literals*, and *blank nodes*, respectively, where literals can be numbers, strings, or Boolean values *true* and *false*. The set **T** of *(RDF) terms* is $\mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$. An *(RDF) triple* is an element (s, p, o) of $(\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times \mathbf{T}$, with s called the *subject*, p the *predicate*, and o the *object*. An *(RDF) graph* is a finite set of RDF triples. In this article, we assume that all graphs are non-empty. Although several of our results hold only under that assumption, we do not see this as a limitation in practice: graphs can be trivially tested for emptiness and querying empty graphs is of no use in most applications.

2.2 SPARQL Algebra Syntax

We adopt the basic algebra from the normative specification [52], but omit certain features that are immaterial to our results, such as ternary expression operator if and auxiliary difference pattern operator *Diff*; these omissions are discussed in Section 9.

We distinguish three types of syntactic building blocks—expressions, patterns, and queries, which we define next for our basic algebra Sparql. They are built over terms **T** and an infinite set $\mathbf{X} = \{?x, ?y, \ldots\}$ of *variables*, disjoint from **T**. In this article, we will often use the following notion: a *renaming* of a set of variables X is an injective substitution from X into fresh variables; then, given a syntactic object s, such as an expression, pattern, query, and a renaming θ of X, $s\theta$ is the object obtained from s by a simultaneous replacement of all $?x \in X$ by $\theta(?x)$.

Expressions in Sparql are inductively defined as follows:

- —all variables in **X** and all terms in $\mathbf{I} \cup \mathbf{L}$ are expressions;
- —if $?x \in \mathbf{X}$, then $\mathsf{bound}(?x)$ is an expression;
- —if $t \in \mathbf{T} \cup \mathbf{X}$, then $\mathsf{isIRI}(t)$, $\mathsf{isLiteral}(t)$, and $\mathsf{isBlank}(t)$ are expressions;
- —if E_1 and E_2 are expressions, then so are
  - $(E_1 + E_2)$, $(E_1 - E_2)$, $(E_1 * E_2)$, (E_1/E_2);
  - $(E_1 \doteq E_2)$, $(E_1 < E_2)$, $(E_1 > E_2)$;
  - $(\neg E)$, $(E_1 \wedge E_2)$, and $(E_1 \vee E_2)$.

We use $E_1 \rightarrow E_2$ and $E_1 \leftrightarrow E_2$ as the usual abbreviations for $\neg E_1 \vee E_2$ and $(E_1 \wedge E_2) \vee (\neg E_1 \wedge \neg E_2)$, respectively. Furthermore, given a set of variables X and a renaming θ of X, we denote with $\mathsf{eq}(X, \theta)$ the expression

$$\bigwedge\nolimits_{?x \in X} \Big((\mathsf{bound}(?x) \leftrightarrow \mathsf{bound}(?x\theta)) \wedge (\mathsf{bound}(?x) \rightarrow ?x \doteq ?x\theta) \Big).$$

This expression checks that values of variables $X\theta$ duplicate the values of X.

The set $\mathsf{var}(E)$ of variables *in the scope of* an expression E is defined as follows:

- —if E is one of $?x$, $\mathsf{bound}(?x)$, $\mathsf{isIRI}(?x)$, $\mathsf{isLiteral}(?x)$, and $\mathsf{isBlank}(?x)$, then $\mathsf{var}(E) = \{?x\}$;
- —if E is a term in $\mathbf{I} \cup \mathbf{L}$ or one of $\mathsf{isIRI}(t)$, $\mathsf{isLiteral}(t)$, and $\mathsf{isBlank}(t)$ for $t \in \mathbf{T}$, then $\mathsf{var}(E) = \emptyset$;
- —if $E = E_1 \circ E_2$, for $\circ \in \{+, -, *, /, \doteq, <, >, \wedge, \vee\}$, then $\mathsf{var}(E) = \mathsf{var}(E_1) \cup \mathsf{var}(E_2)$;
- —if $E = \neg E'$, then $\mathsf{var}(E) = \mathsf{var}(E')$.

Note that for a Sparql expression E the set $\mathsf{var}(E)$ just consists of all the variables occurring in E. We have given a more explicit definition, because this will be different for some extensions of Sparql studied later on.

Patterns in Sparql are inductively defined as follows:

— a *basic graph pattern* (*BGP*) is a possibly empty set of *triple patterns*, that is, elements of the set

$$(\mathbf{I} \cup \mathbf{L} \cup \mathbf{X}) \times (\mathbf{I} \cup \mathbf{X}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{X});$$

— $\mathit{Join}(P_1, P_2)$ and $\mathit{Union}(P_1, P_2)$ are patterns if P_1 and P_2 are patterns;
— $\mathit{Filter}(E, P)$ is a pattern if P is a pattern and E is an expression; and
— $\mathit{LeftJoin}(E, P_1, P_2)$ is a pattern if P_1 and P_2 are patterns and E is an expression.

For brevity, we will often omit braces when writing BGPs consisting of single triple patterns (e.g., we write $(?x, ?y, ?z)$ instead of $\{(?x, ?y, ?z)\}$).

The set $\mathsf{var}(P)$ of variables *in the scope of* a pattern P is as follows:

— $\mathsf{var}(B)$ for a BGP B is the set of all its variables;
— $\mathsf{var}(\mathit{Op}(S_1, S_2)) = \mathsf{var}(S_1) \cup \mathsf{var}(S_2)$ if Op is one of *Join*, *Union* and *Filter*;
— $\mathsf{var}(\mathit{LeftJoin}(E, P_1, P_2)) = \mathsf{var}(E) \cup \mathsf{var}(P_1) \cup \mathsf{var}(P_2)$.

Similar to expressions, $\mathsf{var}(P)$ is just the set of all variables in P for Sparql, which will be different for fragments defined later on.

Queries in Sparql are expressions of the form $\mathit{Project}(X, P)$ or $\mathit{Distinct}(\mathit{Project}(X, P))$, for P a pattern and X a set of variables (called *free variables*).

The variables $\mathsf{var}(Q)$ *in the scope of* a query Q are its free variables.

2.3 SPARQL Algebra Semantics

The semantics of Sparql is defined in terms of *(solution) mappings*, that is, partial functions μ from variables $\mathbf{X}$ to terms $\mathbf{T}$. The domain of μ, denoted $\mathsf{dom}(\mu)$, is the set of variables over which μ is defined. Mappings μ_1 and μ_2 are *compatible*, written $\mu_1 \sim \mu_2$, if $\mu_1(?x) = \mu_2(?x)$ for each $?x$ in $\mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2)$. If $\mu_1 \sim \mu_2$, then $\mu_1 \cup \mu_2$ is the mapping obtained by extending μ_1 according to μ_2 on all the variables in $\mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1)$. We also write $\mu_1 \subseteq \mu_2$ if $\mu_1 \sim \mu_2$ and $\mathsf{dom}(\mu_1) \subseteq \mathsf{dom}(\mu_2)$. The *restriction* $\mu|_X$ of a mapping μ to a set of variables X is the mapping that coincides with μ on X and is undefined on all other variables.

Expressions. The *evaluation* $[\![E]\!]^{\mu}_{G}$ of an expression E over a graph G with respect to a mapping μ is a *value* in $\mathbf{T} \cup \{\mathit{error}\}$, as defined next (graph G does not affect the semantics of expressions in Sparql, but it will do so in relevant extensions of Sparql discussed later on; hence, we include here the subscript G for completeness):

— $[\![?x]\!]^{\mu}_{G}$ is $\mu(?x)$ if $?x \in \mathsf{dom}(\mu)$ and *error* otherwise;
— $[\![\ell]\!]^{\mu}_{G}$ is ℓ for $\ell \in \mathbf{I} \cup \mathbf{L}$;
— $[\![\mathsf{bound}(?x)]\!]^{\mu}_{G}$ is *true* if $?x \in \mathsf{dom}(\mu)$, and it is *false* otherwise;
— $[\![\mathsf{isIRI}(t)]\!]^{\mu}_{G}$, $[\![\mathsf{isLiteral}(t)]\!]^{\mu}_{G}$, or $[\![\mathsf{isBlank}(t)]\!]^{\mu}_{G}$ is *true* if t is an IRI, literal, or blank node, respectively, and *false* otherwise;
— $[\![E_1 \circ E_2]\!]^{\mu}_{G}$, for an arithmetic or comparison operator $\circ \in \{+, -, *, /, \doteq, <, >\}$, is either $[\![E_1]\!]^{\mu}_{G} \circ [\![E_2]\!]^{\mu}_{G}$ if $[\![E_1]\!]^{\mu}_{G}$ and $[\![E_2]\!]^{\mu}_{G}$ are both different from *error* and of suitable types, or *error* otherwise;
— $[\![\neg E']\!]^{\mu}_{G}$ is *true* if $[\![E']\!]^{\mu}_{G}$ is *false*, it is *false* if $[\![E']\!]^{\mu}_{G}$ is *true*, and it is *error* otherwise;
— $[\![E_1 \wedge E_2]\!]^{\mu}_{G}$ is *true* if both $[\![E_1]\!]^{\mu}_{G}$ and $[\![E_2]\!]^{\mu}_{G}$ are *true*, it is *false* if $[\![E_1]\!]^{\mu}_{G}$ or $[\![E_2]\!]^{\mu}_{G}$ is *false*, and it is *error* otherwise;
— $[\![E_1 \vee E_2]\!]^{\mu}_{G}$ is equal to $[\![\neg(\neg E_1 \wedge \neg E_2)]\!]^{\mu}_{G}$.

Patterns. The semantics of patterns is based on *multisets* $\Omega = (S_\Omega, \mathrm{card}_\Omega)$, where S_Ω is the *base set* of mappings, and the *multiplicity* function card_Ω assigns a positive number to each element of S_Ω. Abusing notation, we often write $\mu \in \Omega$ instead of $\mu \in S_\Omega$, as well as $\Omega = \emptyset$ instead of $S_\Omega = \emptyset$. We use the following operations on multisets.

- The *union* $\Omega_1 \uplus \Omega_2$ of multisets Ω_1 and Ω_2 of mappings is the multiset with base set $S_{\Omega_1} \cup S_{\Omega_2}$ such that the multiplicity of each mapping μ is $\mathrm{card}_{\Omega_1}(\mu) + \mathrm{card}_{\Omega_2}(\mu)$ if μ is in both S_{Ω_1} and S_{Ω_2}, and $\mathrm{card}_{\Omega_i}(\mu)$ if μ is only in one S_{Ω_i}.
- The *restriction* $\{\!\!\{ \mu \mid \mu \in \Omega, \mathit{Cond}(\mu) \}\!\!\}$ of a multiset Ω of mappings given a Boolean condition *Cond* on a mapping μ is the multiset with the base set consisting of all $\mu \in \Omega$ with $\mathit{Cond}(\mu)$ true such that the multiplicity of each such μ coincides with the multiplicity of μ in Ω.

We consider two generalisations of multiset restriction:

- $\{\!\!\{ \mu' \mid \mu \in \Omega, \mathit{Cond}(\mu', \mu) \}\!\!\}$ for a multiset Ω of mappings given a Boolean condition *Cond* on two mappings μ' and μ is the multiset with the base set consisting of all μ' for which there exists $\mu \in \Omega$ with $\mathit{Cond}(\mu', \mu)$ true, such that the multiplicity of each such μ' is the sum of the multiplicities of all the contributing μ in Ω (i.e., all μ for which $\mathit{Cond}(\mu', \mu)$ is true);
- $\{\!\!\{ \mu' \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mathit{Cond}(\mu', \mu_1, \mu_2) \}\!\!\}$ for multisets Ω_1 and Ω_2 of mappings given a Boolean condition *Cond* on three mappings μ', μ_1 and μ_2 is the multiset with the base set consisting of all μ' for which there exist μ_1 in Ω_1 and μ_2 in Ω_2 with *Cond* true, such that the multiplicity of each such μ' is the following sum ranging over the pairs of contributing μ_1 and μ_2:

$$\sum \mathrm{card}_{\Omega_1}(\mu_1) \times \mathrm{card}_{\Omega_2}(\mu_2).$$

The *evaluation* $[\![P]\!]_G$ of a pattern P over a graph G is defined as follows:

- $[\![B]\!]_G$ for a BGP B is the multiset with $S_{[\![B]\!]_G}$ consisting of all μ with $\mathrm{dom}(\mu) = \mathrm{var}(B)$ and $\mu(B) \subseteq G$, and with $\mathrm{card}_{[\![B]\!]_G}(\mu) = 1$ for each such μ, where $\mu(B)$ is the BGP obtained from B by replacing its variables according to μ;
- $[\![\mathit{Join}(P_1, P_2)]\!]_G = \{\!\!\{ \mu \mid \mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_G, \mu = \mu_1 \cup \mu_2 \}\!\!\}$;
- $[\![\mathit{Union}(P_1, P_2)]\!]_G = [\![P_1]\!]_G \uplus [\![P_2]\!]_G$;
- $[\![\mathit{Filter}(E, P')]\!]_G = \{\!\!\{ \mu \mid \mu \in [\![P']\!]_G, [\![E]\!]^\mu_G = \mathit{true} \}\!\!\}$; and
- $[\![\mathit{LeftJoin}(E, P_1, P_2)]\!]_G =$
  $\{\!\!\{ \mu \mid \mu \in [\![\mathit{Join}(P_1, P_2)]\!]_G, [\![E]\!]^\mu_G = \mathit{true} \}\!\!\} \uplus$
  $\{\!\!\{ \mu \mid \mu \in [\![P_1]\!]_G, \forall \mu_2 \in [\![P_2]\!]_G . (\mu \nsim \mu_2 \text{ or } [\![E]\!]^{\mu \cup \mu_2}_G \neq \mathit{true}) \}\!\!\}$.[1]

Note that the condition $\mu = \mu_1 \cup \mu_2$ in the case of *Join* implies that $\mu_1 \sim \mu_2$; the latter is not explicitly written here and in similar places throughout the article for brevity. Note also that the semantics of a non-empty BGP $B = \{T_1, \ldots, T_n\}$ can be alternatively defined as $[\![\mathit{Join}(\ldots \mathit{Join}(T_1, T_2), \ldots T_n)]\!]_G$.

Queries. We conclude with the *evaluation* of queries, which are also evaluated as multisets of mappings in our formalisation:

- $[\![\mathit{Project}(X, P)]\!]_G$ is a multiset whose base set consists of the restrictions $\mu|_X$ of all μ in $[\![P]\!]_G$, and the multiplicity of each such $\mu|_X$ is the sum of multiplicities of all corresponding μ in $[\![P]\!]_G$;

[1] The SPARQL 1.1 specification requires $[\![E]\!]^{\mu \cup \mu_2}_G = \mathit{false}$ instead of $[\![E]\!]^{\mu \cup \mu_2}_G \neq \mathit{true}$ in the second argument, but we follow the errata document [31] here. This choice does not affect any results of this article; see a discussion about this issue in [38].

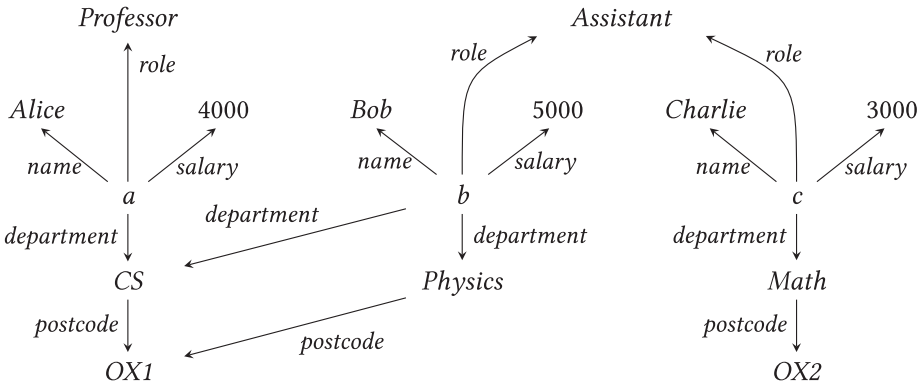

Fig. 1. Example RDF graph G_{ex}.

— $[\![Distinct(Q)]\!]_G$ is the multiset with the same base set as $[\![Q]\!]_G$, but with multiplicity 1 for all mappings.

Example 2.1. Consider RDF graph G_{ex} depicted in Figure 1, which we will use as a running example. Let us evaluate the following SPARQL query over G_{ex}.

(Q0) *Find the departments having at least one employee with salary over 3500.*

It can be written in SPARQL 1.1 as follows:

```
SELECT ?d
  WHERE { ?x department ?d . { ?x salary ?y . FILTER (?y > 3500) } }.
```

Query (Q0) is translated to the Sparql algebra as follows:

$$Project(\{?d\}, Join((?x, department, ?d), Filter((?y > 3500), (?x, salary, ?y)))).$$

The *Filter* subpattern evaluates to mappings

$$\mu_1 = \{?x \mapsto a, ?y \mapsto 4000\}, \qquad \mu_2 = \{?x \mapsto b, ?y \mapsto 5000\},$$

both with multiplicity 1. Alice works only for the *CS* department, whereas *Bob* works for both *CS* and *Physics*; thus, (Q0) evaluates to the multiset of mappings consisting of $\mu_3 = \{?d \mapsto CS\}$ with multiplicity 2 and $\mu_4 = \{?d \mapsto Physics\}$ with multiplicity 1.

2.4 Extensions of SPARQL

In this article, we compare Sparql with several richer languages. Formally, a Sparql *extension* is a language that allows, besides all the constructs of Sparql, for some other forms of expressions, patterns, or queries; additionally, we impose the following natural requirements on the evaluation of any pattern or query S over a graph G:

— $\mathsf{dom}(\mu) \subseteq \mathsf{var}(S)$ for any $\mu \in [\![S]\!]_G$,
— for any renaming θ of $\mathsf{var}(S)$ to variables not mentioned in S, $\mu\theta \in [\![S\theta]\!]_G$ if and only if $\mu \in [\![S]\!]_G$, and the multiplicity of $\mu\theta$ in $[\![S\theta]\!]_G$ equals that of μ in $[\![S]\!]_G$.

Note that, under this definition, not every language that includes all of the Sparql queries can be seen as its extension; however, all the languages considered in this article satisfy these requirements. These restrictions guarantee that $S\theta$ for a renaming θ of $\mathsf{var}(S)$ always give a copy of the answers of S over fresh variables. We silently use this guarantee throughout many proofs in this article.

2.5 Expressive Power of Query Languages

Extending Sparql can lead to an increase in expressive power; that is, some queries in the extended language may not be equivalently expressible in the original language. We next make this notion precise.

A *query language for RDF* is a pair $(\mathbf{Q}, [\![.]\!]_.)$ where $\mathbf{Q}$ is a class of *queries* and $[\![.]\!]_.$ is an *evaluation function* that maps queries and RDF graphs to multisets of solution mappings (e.g., Sparql is a query language for RDF).

A query Q in a language $\mathcal{L} = (\mathbf{Q}, [\![.]\!]_.)$ is *equivalent* to a query Q' in a language $\mathcal{L}' = (\mathbf{Q}', [\![.]\!]'_.)$ if $[\![Q]\!]_G = [\![Q']\!]'_G$ for every graph G. If such Q' exists, then Q is $\mathcal{L}'$*-expressible*. Language $\mathcal{L}_1$ is *more expressive* than $\mathcal{L}_2$ if every $\mathcal{L}_1$ query is $\mathcal{L}_2$-expressible. For example, any Sparql extension is trivially more expressive than Sparql. Languages $\mathcal{L}_1$ and $\mathcal{L}_2$ have the *same expressive power* if each of them is more expressive than the other. Finally, $\mathcal{L}_1$ is *strictly more expressive* than $\mathcal{L}_2$ if it is more expressive than $\mathcal{L}_2$, but does not have the same expressive power as $\mathcal{L}_2$.

3 SUBQUERIES AS PATTERNS

In this and the next section, we investigate the expressive power provided by nested queries, that is, those having other queries (called *subqueries*) embedded within. Subqueries can itself be nested queries; thus, queries can have a deeply nested structure.

The SPARQL 1.1 specification provides two mechanisms for query nesting. First, subqueries can play the role of patterns within the `WHERE` clause of another query. This is tantamount to allowing the arbitrary use of the operators *Project* and *Distinct* within patterns (in which case there is no real distinction between queries and patterns anymore), rather than allowing them only on the outermost level of queries. This basic form of nested queries is the topic of this section; in particular, after introducing a useful tool in Section 3.1, we define subqueries as patterns in Section 3.2 and study their expressive power in Section 3.3. In Section 4, we investigate the second mechanism for query nesting, where patterns can be embedded within expressions.

3.1 Set Difference Operator

Before moving into further particulars, we first show that Sparql can express a "set difference" operator, which we will exploit to encode other constructs (note that this operator is different from the *Minus* and *Diff* operators discussed in Section 9 and in [3, 9, 37]).

Definition 3.1. For P_1 and P_2 patterns, $SetMinus(P_1, P_2)$ is a pattern, whose semantics for a graph G is as follows:

$$[\![SetMinus(P_1, P_2)]\!]_G = \{\!\!\{ \mu \mid \mu \in [\![P_1]\!]_G, \mu \notin [\![P_2]\!]_G \}\!\!\}.$$

In contrast to the relational multiset difference operator, where the occurrences of an element in the right operand are subtracted from those in the left one, this operator yields each μ in $[\![P_1]\!]_G$ but not in $[\![P_2]\!]_G$ with the same multiplicity as in $[\![P_1]\!]_G$. Thus, mapping μ is not returned whenever $\mu \in [\![P_2]\!]_G$, regardless of its multiplicity in $[\![P_1]\!]_G$.

Example 3.2. Consider the evaluation of the following pattern over graph G_{ex}:

$$SetMinus(Union((?x, department, CS), (?x, salary, 5000)), (?x, department, Physics)).$$

The inner *Union* pattern yields the multiset of mappings $\{?x \mapsto a\}$ with multiplicity 1 and $\{?x \mapsto b\}$ with multiplicity 2. Then, the evaluation of the overall *SetMinus* pattern yields $\{?x \mapsto a\}$ with multiplicity 1. Note that the relational multiset difference operator would have additionally produced the mapping $\{?x \mapsto b\}$ with multiplicity 1.

This pattern can be expressed in Sparql using *Filter*, *LeftJoin*, and bound, analogously to the well-known encoding of *Diff* (see, e.g., [9]):

$$Filter(\neg\text{bound}(?u), LeftJoin(true, Union((?x, department, CS), (?x, salary, 5000)), \{(?x, department, Physics), (?u, ?v, ?w)\})).$$

For more complex patterns, however, this encoding needs to be generalised to account for the possibility of some variables in the arguments of *LeftJoin* being unbound.

Lemma 3.3. *For any extension* Sparql$_\star$ *of* Sparql*, the pattern* $SetMinus(P_1, P_2)$ *with* Sparql$_\star$ *patterns* P_1 *and* P_2 *has an equivalent pattern in* Sparql$_\star$.

Proof. We need to establish a mechanism that distinguishes, for any graph G, between the mappings in $[\![P_1]\!]_G$ that occur in $[\![P_2]\!]_G$ and those mappings that do not.

As the first step, we construct a pattern P' whose evaluation contains each mapping μ in $[\![P_1]\!]_G$; if μ is also in $[\![P_2]\!]_G$, however, the corresponding mapping in $[\![P']\!]_G$ is extended with a copy of μ (on fresh variables) as well as a binding for the fresh variables $?x$, $?y$, and $?z$. Then, we show that $SetMinus(P_1, P_2)$ is equivalent to

$$P = Filter(\neg\text{bound}(?x), P').$$

Intuitively, the filter condition eliminates all mappings with fresh variables, thus producing the intended result. Before moving to the details, note that in this construction the multiplicities are important only for those mappings in $[\![P']\!]_G$ that, restricted to $\text{var}(P_1)$, do not appear in $[\![P_2]\!]_G$.

We construct P' as follows, where $X = \text{var}(P_1) \cup \text{var}(P_2)$ and θ is a renaming of X:

$$P'_2 = Join(P_2, (?x, ?y, ?z)), \qquad P' = LeftJoin(\text{eq}(X, \theta), P_1, P'_2\theta).$$

The following claim completes the proof of the lemma.

Claim 3.4. *Patterns* $SetMinus(P_1, P_2)$ *and* P *are equivalent.*

The proof of this claim can be found in the Appendix. □

To check if a mapping constitutes a match for a subpattern, the proof of Lemma 3.3 uses a join with a triple pattern $(?x, ?y, ?z)$, which matches any triple in a graph, coupled with a check of $?x$ for boundedness. This idea is used in several proofs throughout the article. As mentioned above, it is commonly used to express the auxiliary *Diff* operator via the core Sparql operators (see Section 9 for more details). Note, however, that the idea works only if empty graphs are not allowed [37].

3.2 Syntax and Semantics of Subqueries as Patterns

We start by discussing the basic nesting mechanism in SPARQL 1.1, which allows queries to be subpatterns of other patterns as illustrated by the following example.

Example 3.5. Consider the query (Q1) given next.
(Q1) *Find the names of people and the postcodes of the departments where they work.*
It can be written in SPARQL 1.1 as follows:

```
SELECT ?n ?p WHERE { ?x name ?n .
  { SELECT DISTINCT ?x ?p WHERE { ?x department ?d . ?d postcode ?p} } }.
```

Let us evaluate (Q1) over RDF graph G_{ex}. Variable $?d$ is only visible within the subquery, whereas $?x$ and $?p$ are projected and hence visible in the outer query. Since a person can work in two departments with the same postcode (e.g., *Bob* works in *CS* and *Physics*, both located in *OX1*), the subquery uses `DISTINCT` to ensure that the result of the subquery does not contain duplicates. The evaluation of (Q1) over G_{ex} yields the multiset of mappings $\mu_a = \{?n \mapsto Alice, ?p \mapsto OX1\}$,

$\mu_b = \{?n \mapsto Bob, ?p \mapsto OX1\}$, and $\mu_c = \{?n \mapsto Charlie, ?p \mapsto OX2\}$, all with multiplicity 1. Omitting `DISTINCT` in the subquery would yield two copies of μ_b in the evaluation.

To support queries such as (Q1), we extend our basic algebra Sparql by allowing *Project* and *Distinct* in patterns. After this extension, there is no longer a meaningful distinction between patterns and queries.

Definition 3.6. Language Sparql_{PD} extends Sparql by allowing the query constructs $Project(X, P)$ and $Distinct(P)$ as patterns, called *subquery patterns*. The semantics of such patterns is the same as in the case of queries. The intermediate languages Sparql_P and Sparql_D allow only for *Project* and only for *Distinct* as patterns, respectively.

Language Sparql_{PD} captures the query nesting functionality in SPARQL 1.1.

Example 3.7. Query (Q1) from Example 3.5 translates to Sparql_{PD} as follows:

$$Project(\{?n, ?p\}, Join((?x, name, ?n), \\ Distinct(Project(\{?x, ?p\}, \{(?x, department, ?d), (?d, postcode, ?p)\})))).$$

3.3 Expressive Power of Subqueries as Patterns

At first sight, query nesting provides a great deal of power and flexibility to the language. It can lead to sophisticated interactions between set and bag semantics, which may be difficult (or, as we will soon see, impossible) to simulate within plain Sparql. Furthermore, subquery nesting can be arbitrarily deep, and it is reasonable to expect that each additional level of nesting may increase the expressive power of the language.

We next argue that Sparql_{PD} queries can be brought into a normal form where the nesting depth is bounded by two; thus, there is a natural bound on the level of nesting after which no further increase in expressive power can be achieved. This normal form is defined next, and one can check that query (Q1) satisfies its requirements.

Definition 3.8. A Sparql_{PD} query is in *s-normal form* if it has the form $Distinct\ (Project(X, P))$ with P subquery-free, or the form $Project(X, P)$, where all subquery patterns in P are of the form $Distinct(Project(X', P'))$ with P' subquery-free.

This normal form not only limits the nesting depth, but also restricts the ways in which *Project* and *Distinct* can be combined. If a query Q is in Sparql_P (or in Sparql_D) and hence *Distinct* (respectively, *Project*) only occurs in the outermost level, then Definition 3.8 requires that pattern P is subquery-free and hence Q is a Sparql query.

We next show that each Sparql_{PD} query can be brought into s-normal form. The normalisation is based on two ideas that are illustrated in the following example.

Example 3.9. For the first idea, consider the Sparql_D query

$$Distinct(Project(\{?n\}, Distinct((?x, name, ?n)))).$$

Clearly, the inner application of *Distinct* is redundant as the outermost application of *Distinct* already ensures that the answer will contain no duplicates. Hence, the query can be simplified to $Distinct(Project(\{?n\}, (?x, name, ?n)))$.

For the second idea, consider the following modification of (Q1):

$$Project(\{?n, ?p\}, Join((?x, name, ?n), \\ Project(\{?x, ?p\}, \{(?x, department, ?n),\ (?n, postcode, ?p)\}))).$$

To remove the inner projection, it suffices to rename the variable $?n$ in the *Project* subpattern so it no longer clashes with any variable outside of the subpattern:

$$Project(\{?n, ?p\}, \{(?x, name, ?n),\ (?x, department, ?d),\ (?d, postcode, ?p)\}).$$

THEOREM 3.10. *Let* $\text{Sparql}_\star$ *be one of* Sparql_P, Sparql_D, *and* Sparql_{PD}. *Every query in* $\text{Sparql}_\star$ *has an equivalent* $\text{Sparql}_\star$ *query in s-normal form.*

PROOF. The first relevant observation is that all occurrences of *Distinct* in a $\text{Sparql}_\star$ query Q that are in the scope of other *Distinct* subpatterns can be removed upfront without affecting the semantics. In the Appendix, we show the following claim.

CLAIM 3.11. *Any* Sparql_{PD} *query* Q *that has a subpattern* $Distinct(P)$ *such that* P *has in turn a subpattern* $Distinct(P')$ *is equivalent to the query obtained from* Q *by replacing* $Distinct(P')$ *with* P'.

Second, *Project* can be seamlessly "pushed" upwards through all operators except *Distinct*. For instance, $Join(Project(X, P_1), P_2)$ can be equivalently rewritten as $Project(X \cup \text{var}(P_2), Join(P_1\theta, P_2))$ with θ a renaming of $\text{var}(P_1) \setminus X$. Moreover, subsequent occurrences of *Project* can be merged, since the patterns $Project(X_1 \cap X_2, P)$ and $Project(X_1, Project(X_2, P))$ are equivalent.

To complete the proof, we note that every subpattern of Q of the form $Distinct(P)$ can be extended with a "dummy" projection to $Distinct(Project(\text{var}(P), P))$ without changing the semantics. □

From Theorem 3.10 and the definition of s-normal form it immediately follows that extending Sparql by allowing only *Project* or only *Distinct* as pattern constructs does not increase the expressive power of the language.

COROLLARY 3.12. *Languages* Sparql_P *and* Sparql_D *have the same expressive power as* Sparql.

Note that under set semantics, *Distinct* is redundant and hence subqueries as patterns do not add expressivity to Sparql. An obvious question is whether or not this is still the case under bag semantics.

Cohen [18] was the first to observe that query nesting in SQL can cause a complex interplay between bag and set semantics. In her article, Cohen proposes a logical semantics for SQL queries were the aforementioned interplay between bags and sets occurs, and studies the query equivalence problem under such semantics. Cohen's observations were used by Angles and Gutierrez [4] to conjecture that query nesting adds expressive power to SPARQL. Cohen's article, however, does not study questions related to expressive power and her results do not imply the conjecture by Angles and Gutierrez. In particular, Angles and Gutierrez claim that the query given in Example 3.13 below has no equivalent in Sparql. As we will see in Example 4.22, however, this query can be equivalently expressed in Sparql.

Example 3.13 (Angles and Gutierrez [4]). Consider the following query.
(Q2) *Find the names of people working for either the CS or the Math department.*
It can be written in SPARQL 1.1 as follows:

```
SELECT ?n WHERE { ?x name ?n .
  { SELECT DISTINCT *
        WHERE {?x department CS} UNION {?x department Math} } }.
```

Query (Q2) is translated to Sparql_{PD} algebra as follows:

$$Project(\{?n\}, Join((?x, name, ?n), Distinct(Project(\{?x\}, P_u)))),$$

where P_u is $Union((?x, department, CS), (?x, department, Math))$.

We now settle the question whether query nesting in Sparql_{PD} can be fully eliminated by showing that there exist Sparql_{PD} queries that cannot be expressed in Sparql (and hence, by Corollary 3.12, in Sparql_P or Sparql_D).

THEOREM 3.14. *Language* Sparql_{PD} *is strictly more expressive than* Sparql.

PROOF. We claim that the following Sparql_{PD} query Q in s-normal form is not expressible in Sparql:

$$\mathit{Project}(\{?x, ?y\}, \mathit{Join}((?x, p, ?z), \mathit{Distinct}(\mathit{Project}(\{?y\}, (?y, q, ?u))))).$$

Assume for the sake of contradiction that there is a query Q' that is equivalent to Q. Consider graphs $G_{m,n}$, for $m, n \geq 1$, of the following form, where IRIs a, b_i, c, and d_j are chosen such that they do not occur in Q':

$$\{ \quad (a, p, b_1), \ldots, (a, p, b_m), \quad (c, q, d_1), \ldots, (c, q, d_n) \quad \}.$$

Query Q evaluates on every $G_{m,n}$ to the multiset consisting of m copies of the mapping $\mu = \{?x \mapsto a, ?y \mapsto c\}$.

We claim that, in contrast, every Sparql query equivalent to Q modulo multiplicities evaluates on $G_{m,n}$ to a multiset Ω' such that either $\text{card}_{\Omega'}(\mu) = 1$ or $\text{card}_{\Omega'}(\mu) = m \cdot n$. This contradicts our assumption that Q' is equivalent to Q. Intuitively, the contradiction arises because a Sparql query cannot distinguish between the different b_i and d_j, and hence, if a query of the form $\mathit{Project}(X, P)$ returns μ once, it must return its all $m \cdot n$ copies. Of course, we can use *Distinct* in the query's outermost level, but then we obtain a single copy of μ instead of the required m copies.

In what follows, we make this intuition precise. First, note that Q' cannot be of the form $\mathit{Distinct}(\mathit{Project}(X, P))$, since such queries cannot yield mappings with multiplicity greater than one. Therefore, $Q' = \mathit{Project}(X, P)$ for a set of variables X and pattern P. Let us consider the graph $G_{1,2}$ and recall that Q evaluates to a single copy of μ on $G_{1,2}$. Consequently, by assumption of equivalence between Q and Q', we have that $[\![Q']\!]_{G_{1,2}}$ also consists of a single copy of μ. Thus, by the structure of Q', there exists $\mu' \in [\![P]\!]_{G_{1,2}}$ such that $\mu = \mu'|_X$. We argue that there is a triple pattern T in P such that $\mu'(T)$ is a triple in $G_{1,2}$ mentioning IRI c; this can be proved by an easy induction on the structure of P using the facts that $\mu' \in [\![P]\!]_{G_{1,2}}$ and that μ' mentions c (as it extends μ). Recall that, by construction of $G_{m,n}$, query Q' does not mention d_1; hence, T has a variable $?u$ at the object position and μ' sends this variable to either d_1 or d_2. The following claim shows the existence of a mapping in $[\![P]\!]_{G_{1,2}}$ different from μ'. Its proof proceeds by induction on the pattern structure and is given in the Appendix.

CLAIM 3.15. *For every* Sparql *pattern* P *not mentioning* d_1, d_2 *and every* $\mu' \in [\![P]\!]_{G_{1,2}}$, *the mapping* μ'' *such that* $\text{dom}(\mu'') = \text{dom}(\mu')$ *and, for each* $?u \in \text{dom}(\mu')$,

$$\mu''(?u) = \begin{cases} d_2 & \text{if } \mu'(?u) = d_1, \\ d_1 & \text{if } \mu'(?u) = d_2, \\ \mu'(?u) & \text{otherwise,} \end{cases}$$

also belongs to $[\![P]\!]_{G_{1,2}}$.

Claim 3.15 implies that $[\![Q']\!]_{G_{1,2}}$ contains at least two mappings, which contradicts our assumption that Q' is equivalent to Q. This concludes the proof of the theorem. □

4 SUBQUERIES WITHIN EXPRESSIONS

Expressions in Sparql can only be constructed inductively from other expressions; that is, it is not possible for other constructs such as patterns or queries to occur within expressions. In

SPARQL 1.1, however, the construct exists can be used to nest patterns within (possibly complex) expressions as illustrated next.

Example 4.1. Consider the following query (Q3).
(Q3) *Find the names of people not working in CS.*
It can be written in SPARQL 1.1 as follows:

```
SELECT ?n WHERE { ?x name ?n . FILTER NOT EXISTS {?x department CS} }.
```

Intuitively, query (Q3) should evaluate over our example graph G_{ex} to the single mapping $\{?n \mapsto Charlie\}$.

To support queries such as (Q3), the SPARQL 1.1 normative algebra introduces a dedicated construct exists. This construct can be syntactically incorporated into any extension of Sparql as given next.

Definition 4.2. For any Sparql extension $\text{Sparql}_\star$, the language $\text{Sparql}_\star^\exists$ further extends $\text{Sparql}_\star$ by permitting expressions of the form $\text{exists}(P)$ for a pattern P. Moreover, we define $\text{var}(\text{exists}(P)) = \text{var}(P)$.

Next, we illustrate the translation of subqueries in expressions to $\text{Sparql}^\exists$.

Example 4.3. Query (Q3) from Example 4.1 translates to $\text{Sparql}^\exists$ as follows:

$$Project(\{?n\}, Filter(\neg\text{exists}((?x, department, CS)), (?x, name, ?n))).$$

We discuss the formal semantics of exists expressions in the next two subsections.

4.1 Normative Semantics of exists

The normative semantics of SPARQL 1.1 treats exists as a Boolean expression. Formally, it works with patterns normalised as follows: a pattern P is *rectified* if for each of its subpatterns $Project(X, P')$, no variable in $\text{var}(P') \setminus X$ appears in P outside this subpattern. Intuitively, variables that occur in P' but do not belong to X are local for the pattern $Project(X, P')$. Every pattern P can be converted to an equivalent rectified pattern $\bar{P}$ by renaming all local variables to fresh variables.

Definition 4.4. The *(normative) evaluation* of exists expressions over a graph G with respect to a mapping μ is defined as follows, where $\mu(\bar{P})$ is the result of replacing all variables $?x \in \text{dom}(\mu)$ in $\bar{P}$ by $\mu(?x)$:

$$[\![\text{exists}(P)]\!]_G^\mu = \begin{cases} true & \text{if } [\![\mu(\bar{P})]\!]_G \neq \emptyset, \\ false & \text{otherwise.} \end{cases}$$

In other words, the evaluation of $\text{exists}(P)$ for μ and G is realised in four steps. First, the pattern P is rectified to $\bar{P}$.[2] Second, the mapping μ is applied to $\bar{P}$, resulting in a new pattern $\mu(\bar{P})$ where variables in $\bar{P}$ have been replaced according to μ. Third, the new pattern $\mu(\bar{P})$ is evaluated against graph G to yield a multiset of mappings $[\![\mu(\bar{P})]\!]_G$. Finally, $[\![\text{exists}(P)]\!]_G^\mu$ is deemed *true* (respectively *false*) if $[\![\mu(\bar{P})]\!]_G$ is non-empty (respectively, empty).

Example 4.5. Consider query (Q3) from Examples 4.1 and 4.3. The triple pattern $(?x, name, ?n)$ evaluates over G_{ex} to mappings

$$\mu_1 = \{?x \mapsto a, ?n \mapsto Alice\}, \mu_2 = \{?x \mapsto b, ?n \mapsto Bob\}, \text{ and } \mu_3 = \{?x \mapsto c, ?n \mapsto Charlie\},$$

[2]The rectification step is missing in the SPARQL 1.1 specification itself, but should be done according to the errata document [31, query 8]. This reflects the intuition that renaming of local variables should not affect the semantics.

all with multiplicity 1. The expression in the filter condition evaluates to *true* only for μ_3. Specifically, when applying μ_3 to $(?x, department, CS)$, we obtain the ground triple pattern $(c, department, CS)$, which evaluates to the empty multiset of mappings over G_{ex}. As a result, we have that $[\![\text{exists}((?x, department, CS))]\!]^{\mu_3}_{G_{ex}}$ is *false* and its negation is therefore *true*.

Now, suppose the subpattern $(?x, department, CS)$ in query (Q3) is replaced by a subquery $P = Project(\emptyset, (?x, department, CS))$, resulting in the following $\text{Sparql}^{\exists}_{P}$ query (Q3′):

$$Project(\{?n\}, Filter(\neg\text{exists}(Project(\emptyset, (?x, department, CS))), (?x, name, ?n))).$$

Since $\text{var}(P) = \emptyset$ but variable $?x$ occurs outside P in (Q3′), the rectified version of (Q3′) replaces $?x$ in P by a fresh variable $?y$:

$$Project(\{?n\}, Filter(\neg\text{exists}(Project(\emptyset, (?y, department, CS))), (?x, name, ?n))).$$

Consequently, since $[\![\text{exists}((?y, department, CS))]\!]^{\mu_i}_{G_{ex}}$ is *true* for all μ_i, query (Q3′) evaluated over G_{ex} returns the empty multiset.

There are at least two entries in the SPARQL 1.1 errata document that ultimately relate to the semantics of exists (see errata queries 8 and 10 in [31]). The main issue behind both reports boils down to the fact that SPARQL 1.1 patterns are not closed under substitution; that is, $\mu(\bar{P})$ may not be a syntactically valid pattern even if P is. In such cases, the semantics of exists is undefined.

Example 4.6. Consider the following SPARQL 1.1 query:

```
SELECT ?x
    WHERE { ?x ?x ?x . FILTER EXISTS { ?y ?y ?y . FILTER BOUND(?x) } }.
```

The corresponding algebraic translation is as follows:

$$Project(\{?x\}, Filter(\text{exists}(Filter(\text{bound}(?x), (?y, ?y, ?y))), (?x, ?x, ?x))).$$

Given the graph $\{(a, a, a)\}$, one would intuitively expect the query to return the multiset containing a single mapping $\mu = \{?x \mapsto a\}$ with multiplicity 1. The normative semantics is, however, ill-defined in this case: if we apply μ to the pattern inside the exists expression, we obtain $Filter(\text{bound}(a), (?y, ?y, ?y))$, which is not a valid pattern, since bound only accepts variables as arguments.

A similar problem can appear when the pattern inside an exists expression involves subqueries. Consider the SPARQL 1.1 query

```
SELECT ?x WHERE { ?x ?x ?x .
    FILTER EXISTS { ?y ?y ?y . { SELECT ?x WHERE { (?x a ?y) } } } }
```

and its algebraic translation to $\text{Sparql}^{\exists}_{P}$

$$Project(\{?x\}, Filter(\text{exists}(Join((?y, ?y, ?y), Project(\{?x\}, (?x, a, ?y)))), (?x, ?x, ?x))).$$

The normative evaluation of the query on the graph is also ill-defined as it requires evaluating $Project(\{a\}, (a, a, ?y))$, which is not a syntactically valid pattern.

Neither the SPARQL 1.1 specification nor the errata documents provide a fix for this issue and hence the normative semantics of exists is currently ill-defined.

In the following section, we propose a novel semantics for exists that provides a principled and elegant solution to the aforementioned issues.

4.2 Environment Semantics of exists

The semantics proposed in this section is a conservative extension of the normative one, in the sense that both semantics coincide whenever the normative semantics is well-defined. Unlike the normative specification, our semantics does not require any manipulation of syntactic objects. It is inspired by standard practices for defining denotational semantics of programming languages. The main idea is to make the semantics of patterns depend not only on the input graph, but also on a mapping, called *environment*. A pattern in an exists expression is then evaluated in the environment given by the mapping that the expression is tested on; in the absence of exists expressions, the environment remains empty.

Definition 4.7. The *environment evaluation* $[E]^{\mu}_{G}$ of a $\text{Sparql}^{\exists}_{PD}$ expression E over a graph G with respect to a mapping μ is defined exactly the same as $[\![E]\!]^{\mu}_{G}$ in Section 2.3 except that $[E']^{\mu}_{G}$ is used instead of $[\![E']\!]^{\mu}_{G}$ for each participating subexpression E' of E (e.g., $[\text{bound}(?x)]^{\mu}_{G} = [\![\text{bound}(?x)]\!]^{\mu}_{G}$ and $[E_1 \wedge E_2]^{\mu}_{G}$ is *true* if both $[E_1]^{\mu}_{G}$ and $[E_2]^{\mu}_{G}$ are *true*, it is *false* if $[E_1]^{\mu}_{G}$ or $[E_2]^{\mu}_{G}$ is *false*, and it is *error* otherwise), and for the case $E = \text{exists}(P)$ it is defined as follows:

$$[\text{exists}(P)]^{\mu}_{G} = \begin{cases} \mathit{true} & \text{if } [P]^{\mu}_{G} \neq \emptyset, \\ \mathit{false} & \text{otherwise.} \end{cases}$$

The *environment evaluation* $[P]^{\nu}_{G}$ of a $\text{Sparql}^{\exists}_{PD}$ pattern P over a graph G in an environment ν is defined as follows:

- $[B]^{\nu}_{G}$ for a BGP B is the multiset that consists of all μ, with multiplicity 1, such that
  - $\mu \sim \nu$,
  - $\text{dom}(\mu) = \text{var}(B) \cup \text{dom}(\nu)$, and
  - $\mu(B) \subseteq G$;
- $[\mathit{Project}(X, P')]^{\nu}_{G} = \{\!\!\{ \mu \mid \mu' \in [P']^{\nu|_X}_{G}, \mu = \mu'|_X \cup \nu|_{\text{dom}(\nu)\setminus X} \}\!\!\}$;
- in all the remaining cases, the definition of $[P]^{\nu}_{G}$ is the same as the definition of $[\![P]\!]_G$ in Section 2.3 except that $[P']^{\nu}_{G}$ and $[E]^{\mu}_{G}$ are used instead of $[\![P']\!]_G$ and $[\![E]\!]^{\mu}_{G}$, respectively, for every participating subpattern P' and subexpression E of P (e.g., $[\mathit{Filter}(E, P')]^{\nu}_{G} = \{\!\!\{ \mu \mid \mu \in [P']^{\nu}_{G}, [E]^{\mu}_{G} = \mathit{true} \}\!\!\}$).

Finally, the *environment evaluation* $[Q]_G$ of a $\text{Sparql}^{\exists}_{PD}$ query Q over a graph G is the evaluation $[Q]^{\emptyset}_{G}$ of Q, seen as pattern, in the empty environment $\emptyset$.

These definitions require some discussion. First, note the difference to the normative semantics of exists(P): instead of constructing a pattern $\mu(\bar{P})$ and evaluating it over G, we now consider $[P]^{\mu}_{G}$ where μ is acting as the environment. Note also that for a BGP B, we can equivalently define the semantics as

$$[B]^{\nu}_{G} = \{\!\!\{ \mu \mid \mu' \in [\![B]\!]_G, \mu = \mu' \cup \nu \}\!\!\},$$

that is, $[B]^{\nu}_{G}$ consists of the mappings in the evaluation of B without any environment, but extended by ν; if the extension is not possible, the mapping is just dropped.

Next, the definition of $[\mathit{Project}(X, P)]^{\nu}_{G}$ reflects the intuition (mentioned in the errata document) that the variables that are projected out are local, and environments should not affect their values.

Finally, note that $[P]^{\emptyset}_{G} = [\![P]\!]_G$ for any Sparql_{PD} pattern P and graph G, that is, for patterns without exists the normative and the environment semantics coincide.

Example 4.8. Consider the evaluation of query (Q3) from Examples 4.1 and 4.3 on G_{ex} using our proposed semantics. As in Example 4.5, $[(?x, \mathit{name}, ?n)]^{\emptyset}_{G_{ex}}$ evaluates to mappings μ_1, μ_2, and μ_3.

The expression in the filter condition evaluates to *true* only on

$$\mu_3 = \{?x \mapsto c, ?n \mapsto \mathit{Charlie}\}.$$

Specifically, $[(?x, \mathit{department}, \mathit{CS})]^{\mu_3}_{G_{ex}}$ is the empty multiset, since the only mapping with domain $\{?x, ?n\}$ compatible with μ_3 is precisely μ_3, and the application of μ_3 to $(?x, \mathit{department}, \mathit{CS})$ does not belong to G_{ex}.

Example 4.9. Consider the problematic query in Example 4.6 and its evaluation over the graph $G = \{(a, a, a)\}$. To this end, let us compute the evaluation $[\mathit{Filter}(\mathsf{exists}(P), (?x, ?x, ?x))]^{\emptyset}_G$ for $P = \mathit{Filter}(\mathsf{bound}(?x), (?y, ?y, ?y))$. The only mapping in $[(?x, ?x, ?x)]^{\emptyset}_G$ is $\mu = \{?x \mapsto a\}$ (whose multiplicity is 1). Next, we show that μ is an answer to the whole query as well. By our semantics of exists, μ "survives" the filter condition only if $[P]^{\mu}_G$ is non-empty. This holds because $[P]^{\mu}_G$ consists of a single mapping $\mu' = \{?x \mapsto a, ?y \mapsto a\}$, which extends μ and has $[\mathsf{bound}(?x)]^{\mu'}_G = \mathit{true}$.

We conclude this section with a natural property of the environment semantics of $\mathsf{Sparql}^{\exists}_{PD}$ and, as we will see later, of every other language considered in this article.

Proposition 4.10. *Let P be a $\mathsf{Sparql}^{\exists}_{PD}$ pattern, G a graph, and ν an environment. Then $\nu \subseteq \mu$ for every $\mu \in [P]^{\nu}_G$.*

The inductive proof of this proposition is given in the Appendix.

Example 4.11. Let $P = (?x, \mathit{name}, ?n)$ and $\nu = \{?x \mapsto c\}$. Then $[P]^{\nu}_{G_{ex}}$ consists of a single mapping $\{?x \mapsto c, ?n \mapsto \mathit{Charlie}\}$, which is a superset of ν. The normative semantics, on the other hand, does not satisfy Proposition 4.10: $[\![\nu(P)]\!]_{G_{ex}}$ consists of a single mapping $\{?n \mapsto \mathit{Charlie}\}$, which does not bind $?x$. In fact, for every pattern P, mapping ν, graph G and mapping $\mu \in [\![\nu(P)]\!]_G$, we have $\mathsf{dom}(\mu) \cap \mathsf{dom}(\nu) = \emptyset$.

4.3 Comparison between Normative and Environment Semantics

Unlike the normative semantics, our proposed semantics is well-defined for all $\mathsf{Sparql}^{\exists}_{PD}$ patterns. We next show that our semantics is a conservative extension of the normative one in that the two coincide on all patterns and graphs for which the normative semantics is *well-defined*, that is, for which no ill-formed patterns appear in the process of evaluation.

The following lemma formally relates the two semantics, taking into account the differences observed in Example 4.11. For the lemma, it is convenient to generalise the notion of *rectification* to expressions: $\bar{E}$ for an expression E is defined exactly the same as $\bar{P}$ for a pattern P.

Lemma 4.12. *Let G be a graph, E a $\mathsf{Sparql}^{\exists}_{PD}$ expression with rectification $\bar{E}$, and P a $\mathsf{Sparql}^{\exists}_{PD}$ pattern with rectification $\bar{P}$.*

1. *For every two mappings μ, ν such that the variables introduced by the rectification $\bar{E}$ of E are not in $\mathsf{dom}(\mu) \cup \mathsf{dom}(\nu)$ and $[\![\nu(\bar{E})]\!]^{\mu}_G$ is well-defined, if $\mu \sim \nu$, then $[\![\nu(\bar{E})]\!]^{\mu}_G = [E]^{\mu \cup \nu}_G$.*
2. *For every two mappings μ, ν such that the variables introduced by the rectification $\bar{P}$ of P are not in $\mathsf{dom}(\nu)$ and $[\![\nu(\bar{P})]\!]_G$ is well-defined,*
   (a) *if $\mu \in [\![\nu(\bar{P})]\!]_G$, then $\mu \cup \nu \in [P]^{\nu}_G$ and $\mathsf{card}_{[\![\nu(\bar{P})]\!]_G}(\mu) = \mathsf{card}_{[P]^{\nu}_G}(\mu \cup \nu)$;*
   (b) *if $\mu \in [P]^{\nu}_G$, then $\mu|_{\mathsf{dom}(\mu) \setminus \mathsf{dom}(\nu)} \in [\![\nu(\bar{P})]\!]_G$ and $\mathsf{card}_{[P]^{\nu}_G}(\mu) = \mathsf{card}_{[\![\nu(\bar{P})]\!]_G}(\mu|_{\mathsf{dom}(\mu) \setminus \mathsf{dom}(\nu)})$.*

The proof of this lemma, which proceeds by simultaneous induction on the structure of E and P, is given in the Appendix. Note that, for the induction to go through, the lemma makes additional assumptions on the domain of environments, which are trivially satisfied by the empty environment.

We are now ready to prove that the two semantics are equivalent.

Theorem 4.13. *For any* Sparql$^{\exists}_{PD}$ *pattern* P *and graph* G *such that* $[\![P]\!]_G$ *is well-defined it holds that* $[\![P]\!]_G = [P]^{\emptyset}_G$.

Proof. By definition, $[\![\bar{P}]\!]_G = [\![P]\!]_G$. Statement 2(a) in Lemma 4.12 applied to the empty environment implies that every mapping in $[\![P]\!]_G$ belongs to $[P]^{\emptyset}_G$ with the same multiplicity. Statement 2(b) in the same lemma implies that every mapping in $[P]^{\emptyset}_G$ belongs to $[\![P]\!]_G$ also with the same multiplicity. □

In the rest of the article, we use (often silently) the environment semantics (i.e., $[.]^{\cdot}_{\cdot}$) instead of the normative one (i.e., $[\![.]\!]_.$). Note, however, that the notion of pattern equivalence is now parametrised by an environment.

Definition 4.14. Given a Sparql extension Sparql$_\star$ and an environment ν, Sparql$_\star$ patterns P_1 and P_2 are *ν-equivalent* if $[P_1]^{\nu}_G = [P_2]^{\nu}_G$ for any graph G.

The presence of environments may have non-trivial consequences. For example, the proof of Lemma 3.3 on the expressibility of *SetMinus* works only for the empty environment: the justifying rewriting involves the renaming $P_2'\theta$, whose evaluation $[P_2'\theta]^{\nu}_G$ may be different, for some nonempty ν, from $[P_2']^{\nu}_G$ renamed according to θ, because ν may interact differently with the mappings appearing during the evaluation of P_2' and $P_2'\theta$. Therefore, we shall be careful when exploiting statements such as Lemma 3.3 later on and make sure that we never apply them in the scope of an exists expression.

Note also that the natural requirements we impose on extensions of Sparql$^{\exists}$ apply only to the empty environment: for any pattern or query S and any graph G

—$\mathsf{dom}(\mu) \subseteq \mathsf{var}(S)$ for any $\mu \in [S]^{\emptyset}_G$, and
—for any renaming θ of $\mathsf{var}(S)$, $\mu\theta \in [S\theta]^{\emptyset}_G$ if and only if $\mu \in [S]^{\emptyset}_G$.

4.4 Expressive Power of exists

The exists construct seems rather powerful as it makes the languages of patterns and expressions mutually recursive. Furthermore, expressions can occur not only as parameters of *Filter*, but also in *LeftJoin* patterns. As we show next, however, exists provides no additional expressive power, since it can be simulated by other constructs.

We proceed according to the following three steps.

(1) We show that exists can be eliminated from *LeftJoin* patterns. This is achieved by "pushing" complex expressions from *LeftJoin* into *Filter* patterns.
(2) We prove that any *Filter* pattern can be expressed in terms of exists-free *Filter* patterns and patterns of the form $\mathit{Filter}(\mathsf{exists}(P_1), P_2)$ or $\mathit{Filter}(\neg\mathsf{exists}(P_1), P_2)$.
(3) We show that patterns of the form $\mathit{Filter}(\mathsf{exists}(P_1), P_2)$ and $\mathit{Filter}(\neg\mathsf{exists}(P_1), P_2)$ can be rewritten in terms of exists-free patterns.

The following example demonstrates the ideas behind these steps.

Example 4.15. Consider the Sparql$^{\exists}$ pattern P_0 defined as follows:

$$\mathit{LeftJoin}(\mathsf{exists}((?x, \mathit{role}, \mathit{Professor})) \vee (?y > 4000), (?x, \mathit{name}, ?n), (?x, \mathit{salary}, ?y)).$$

Intuitively, the pattern returns the names of people as well as their salaries, where salaries are only reported for professors and people earning more than 4000. Evaluated over G_{ex}, the pattern returns three mappings: $\mu_1 = \{?x \mapsto a, ?n \mapsto \mathit{Alice}, ?y \mapsto 4000\}$, $\mu_2 = \{?x \mapsto b, ?n \mapsto \mathit{Bob}, ?y \mapsto 5000\}$, and $\mu_3 = \{?x \mapsto c, ?n \mapsto \mathit{Charlie}\}$.

We begin by demonstrating how to move the join condition of P_0 into *Filter* patterns. Let P_1' be the following pattern, capturing people whose salary is to be reported:

$$\mathit{Filter}(\mathsf{exists}((?x, \mathit{role}, \mathit{Professor})) \vee (?y > 4000), \{(?x, \mathit{name}, ?n),\ (?x, \mathit{salary}, ?y)\}).$$

Then P_0 can be equivalently restated as the following pattern P_1:

$$\mathit{Union}(P_1', \mathit{Filter}(\neg\mathsf{bound}(?u), \mathit{LeftJoin}(\mathit{true}, (?x, \mathit{name}, ?n), \mathit{Join}(P_1', (?u, ?v, ?w))))).$$

Next, we rewrite P_1 to an equivalent pattern P_2 in which all occurrences of exists are of the form $\mathit{Filter}((\neg)\mathsf{exists}(P'), P'')$. For P_1, it suffices to rewrite P_1' to $P_2' = \mathit{Union}(P_{21}', P_{22}')$, where

$$P_{21}' = \mathit{Filter}(\mathsf{exists}((?x, \mathit{role}, \mathit{Professor})), \{(?x, \mathit{name}, ?n),\ (?x, \mathit{salary}, ?y)\}),$$
$$P_{22}' = \mathit{Filter}(\neg\mathsf{exists}((?x, \mathit{role}, \mathit{Professor})), \mathit{Filter}(?y > 4000, \{(?x, \mathit{name}, ?n),\ (?x, \mathit{salary}, ?y)\})).$$

Pattern P_2 is then obtained from P_1 by replacing all occurrences of P_1' with P_2'.

Finally, we eliminate exists from P_2. Consider first the negative occurrence in P_{22}'. It can be rewritten using *Filter*, bound, and *LeftJoin*, yielding the pattern

$$\mathit{Filter}(\neg\mathsf{bound}(?u'), \mathit{Filter}(?y > 4000, \mathit{LeftJoin}(\mathit{true}, \{(?x, \mathit{name}, ?n), (?x, \mathit{salary}, ?y)\}, \{(?x, \mathit{name}, ?n), (?x, \mathit{salary}, ?y), (?x, \mathit{role}, \mathit{Professor}), (?u', ?v', ?w')\}))).$$

The positive occurrence of exists in P_{21}' can, in turn, be rewritten using *SetMinus* in terms of its negation yielding the pattern

$$\mathit{SetMinus}(\{(?x, \mathit{name}, ?n), (?x, \mathit{salary}, ?y)\}, \mathit{Filter}(\neg\mathsf{exists}((?x, \mathit{role}, \mathit{Professor})), \{(?x, \mathit{name}, ?n), (?x, \mathit{salary}, ?y)\})).$$

This negative occurrence can then be eliminated analogously to the one in P_{22}'. By replacing P_{21}' and P_{22}' by their exists-free counterparts, we obtain an exists-free pattern equivalent to P_0.

We show correctness of the three steps only with respect to the empty environment. Thus, in the presence of nested exists expressions, these steps must be applied iteratively in a top-down manner. To this end, we should also guarantee that no step increases the nesting depth of exists (and the third step decreases it by one). We refer to subpatterns of patterns not in the scope of any exists expressions as *top-level*.

The first step is justified by the following lemma.

Lemma 4.16. *Let* $\mathsf{Sparql}_\star$ *be any* Sparql *extension. Every* $\mathsf{Sparql}_\star^\exists$ *pattern is* $\emptyset$*-equivalent to a* $\mathsf{Sparql}_\star^\exists$ *pattern with the same nesting depth of* exists *all of whose top-level LeftJoin subpatterns are of the form* $\mathit{LeftJoin}(\mathsf{eq}(X, \theta), P_1', P_2')$.

Proof. Consider an arbitrary $\mathsf{Sparql}_\star^\exists$ pattern of the form $\mathit{LeftJoin}(E, P_1, P_2)$. We first construct P such that, for any graph G, $[P]_G^\emptyset$ captures (with possibly incorrect multiplicities) all the mappings μ_1 in $[P_1]_G^\emptyset$ having a compatible μ_2 in $[P_2]_G^\emptyset$ such that $[E]_G^{\mu_1 \cup \mu_2}$ is *true*, extended with a "certificate" in the form of a possible compatible extension. Such a P can be defined as follows, where $X = \mathsf{var}(P_1) \cup \mathsf{var}(P_2)$, and θ_1 is a renaming of X:

$$\mathit{Filter}(E\theta_1, \mathit{Join}(\mathit{Filter}(\mathsf{eq}(X, \theta_1), \mathit{Join}(P_1, P_1\theta_1)), P_2\theta_1)).$$

More formally, by construction, we have that $\mu_1 \in [P_1]_G^\emptyset$, $\mu_2 \in [P_2]_G^\emptyset$, $\mu_1 \sim \mu_2$, and $[E]_G^{\mu_1 \cup \mu_2} = \mathit{true}$, if and only if $\mu_1 \cup (\mu_1 \cup \mu_2)\theta_1 \in [P]_G^\emptyset$.

We then construct P' such that $[P']^{\emptyset}_G$ captures, with correct multiplicities, all mappings $\mu_1 \in [P_1]^{\emptyset}_G$ that do not have any corresponding μ_2. For this, we use the following construction, which involves fresh variables $?x$, $?y$, $?z$ and another renaming θ_2 of X:

$$P' = \mathit{Filter}(\neg\mathsf{bound}(?x), \mathit{LeftJoin}(\mathsf{eq}(X, \theta_2), P_1, P^x\theta_2)),$$

where $P^x = \mathit{Join}(P, (?x, ?y, ?z))$. By construction, $\mu_1 \in [P']^{\emptyset}_G$ if and only if $\mu_1 \in [P_1]^{\emptyset}_G$ and there is no μ_2 such that $\mu_1 \cup (\mu_1 \cup \mu_2)\theta_1 \in [P]^{\emptyset}_G$ (recall that G is non-empty, so $(?x, ?y, ?z)$ always matches); moreover, whenever $\mu_1 \in [P']^{\emptyset}_G$ it holds that $\mathsf{card}_{[P']^{\emptyset}_G}(\mu_1) = \mathsf{card}_{[P_1]^{\emptyset}_G}(\mu_1)$. In other words, P' captures the second operand of the bag union in the definition of *LeftJoin*.

Hence, the pattern $\mathit{LeftJoin}(E, P_1, P_2)$ is equivalent to $\mathit{Union}(\mathit{Filter}(E, \mathit{Join}(P_1, P_2)), P')$. □

In the second step, we show that all patterns involving exists can be reduced to patterns where exists appears in expressions only at the top or under a single negation.

Lemma 4.17. *Let* Sparql$_\star$ *be any* Sparql *extension. Every* Sparql$^{\exists}_\star$ *pattern is* $\emptyset$*-equivalent to a* Sparql$^{\exists}_\star$ *pattern with the same nesting depth of* exists *each of whose top-level Filter subpatterns involving* exists *has the form* $\mathit{Filter}(\mathsf{exists}(P_2), P_1)$ *or* $\mathit{Filter}(\neg\mathsf{exists}(P_2), P_1)$.

Proof. Consider a Sparql$^{\exists}_\star$ pattern $\mathit{Filter}(E, P)$, where E contains a top-level occurrence of an expression $\mathsf{exists}(P')$ (i.e., an occurrence that is not inside any other exists expression). Since $\mathsf{exists}(P')$ must evaluate to either *true* or *false*, we can capture each possibility by replacing $\mathsf{exists}(P')$ in E by the respective truth value to obtain the equivalent pattern

$$\mathit{Union}(\mathit{Filter}(\mathsf{exists}(P'), \mathit{Filter}(E[\mathit{true}], P)), \mathit{Filter}(\neg\mathsf{exists}(P'), \mathit{Filter}(E[\mathit{false}], P))),$$

where $E[E']$ is E with $\mathsf{exists}(P')$ replaced by E'. □

For the final step, we first show that patterns of the form $\mathit{Filter}(\neg\mathsf{exists}(P_2), P_1)$ can be expressed in Sparql by means of a structural transformation $\mathsf{Sub}_\theta(P_2, P_1)$, for θ a renaming of $\mathsf{var}(P_1)$. Intuitively, $\mathsf{Sub}_\theta(P_2, P_1)$ joins each subpattern of P_2 with a fresh copy of P_1 in such a way that every mapping in the evaluation of the transformed pattern over a graph G has the form $\mu_2 \cup \mu_1\theta \cup \ldots$, for $\mu_2 \in [P_2]^{\mu_1}_G$ and $\mu_1 \in [P_1]^{\emptyset}_G$. The construction of $\mathsf{Sub}_\theta(P_2, P_1)$ is complicated by the fact that $[P_1]^{\emptyset}_G$ may contain distinct but pairwise compatible mappings. Thus, to ensure that all subpatterns of P_2 are indeed evaluated in the same environment, the transformation employs techniques previously seen in the rewriting of the operator *SetMinus* in Lemma 3.3 and in the first step of exists elimination in Lemma 4.16.

Once we have a rewriting for $\mathit{Filter}(\neg\mathsf{exists}(P_2), P_1)$, patterns of the form $\mathit{Filter}(\mathsf{exists}(P_2), P_1)$ are reduced to $\mathit{Filter}(\neg\mathsf{exists}(P_2), P_1)$ using *SetMinus*.

Lemma 4.18. *Let* Sparql$_\star$ *be one of* Sparql, Sparql$_P$, Sparql$_D$ *and* Sparql$_{PD}$. *Any* Sparql$^{\exists}_\star$ *pattern with all top-level subpatterns with* exists *of the form* $\mathit{Filter}(\neg\mathsf{exists}(P_2), P_1)$ *or* $\mathit{Filter}(\mathsf{exists}(P_2), P_1)$ *is* $\emptyset$*-equivalent to a* Sparql$^{\exists}_\star$ *pattern with a smaller nesting depth of* exists.

Proof. We begin with $\mathit{Filter}(\neg\mathsf{exists}(P_2), P_1)$. We will use an abbreviation,

$$\mathit{Copy}_\theta(P') = \mathit{Filter}(\mathsf{eq}(\mathsf{var}(P'), \theta), \mathit{Join}(P', P'\theta)),$$

for a pattern P' and renaming θ of $\mathsf{var}(P')$. Intuitively, pattern $\mathit{Copy}_\theta(P')$ duplicates, for each mapping in the answer to P', the values of variables of P' and stores them in fresh copies $(\mathsf{var}(P'))\theta$ (note that the multiplicities of mappings with duplicated values may be different from the original,

but this is inessential, as we will see later). We also use the following abbreviations, for patterns P^1 and P^2, a set of variables Y, renamings θ^1 and θ^2 of Y, and an expression E':

$$EqJoin^{Y}_{\theta^1,\theta^2}(P^1,P^2) = Filter(\mathsf{eq}(Y\theta^1,(\theta^1)^{-1}\circ\theta^2),Join(P^1,P^2)),$$
$$EqLeftJoin^{Y}_{\theta^1,\theta^2}(E,P^1,P^2) = LeftJoin(E\wedge\mathsf{eq}(Y\theta^1,(\theta^1)^{-1}\circ\theta^2),P^1,P^2),$$

where $(\theta^1)^{-1}$ is the inverse of θ^1, that is the renaming of the range of θ^1 such that $?x((\theta^1)^{-1}) = ?y$ for each $?y \in Y$ with $?y\theta^1 = ?x$ (this renaming is well-defined, because θ^1 is injective by definition), and $(\theta^1)^{-1}\circ\theta^2$ is the composition of $(\theta^1)^{-1}$ and θ^2, that is, the renaming of the range of θ^1 such that $?x((\theta^1)^{-1}\circ\theta^2) = (?x(\theta^1)^{-1}))\theta^2$ for each $?x$ in the domain of $(\theta^1)^{-1}$. Intuitively, pattern $EqJoin^{Y}_{\theta^1,\theta^2}(P^1,P^2)$ yields all mappings in $Join(P^1,P^2)$ that coincide on the variables in $Y\theta^1$ and $Y\theta^2$, respectively. Pattern $EqLeftJoin^{Y}_{\theta^1,\theta^2}(E,P^1,P^2)$ is an analogue of $EqJoin^{Y}_{\theta^1,\theta^2}(P^1,P^2)$ for $LeftJoin(E,P^1,P^2)$.

Given patterns P and P_1 and a renaming θ of $\mathsf{var}(P_1)$, we define $\mathsf{Sub}_\theta(P,P_1)$ by induction on the structure of P as follows, where θ^1 and θ^2 are fresh renamings of $\mathsf{var}(P_1)$, and θ_Y is a fresh renaming of Y for every recursive application of Sub_θ:

$$\mathsf{Sub}_\theta(B,P_1) = Join(B,Copy_\theta(P_1)),\ \text{for a BGP } B,$$
$$\mathsf{Sub}_\theta(Join(P^1,P^2),P_1) = EqJoin^{\mathsf{var}(P_1)}_{\theta,\theta^1}(EqJoin^{\mathsf{var}(P_1)}_{\theta^1,\theta^2}(\mathsf{Sub}_{\theta^1}(P^1,P_1),\mathsf{Sub}_{\theta^2}(P^2,P_1)),P_1\theta),$$
$$\mathsf{Sub}_\theta(Union(P^1,P^2),P_1) = Union(\mathsf{Sub}_\theta(P^1,P_1),\mathsf{Sub}_\theta(P^2,P_1)),$$
$$\mathsf{Sub}_\theta(Filter(E,P'),P_1) = Filter(E,\mathsf{Sub}_\theta(P',P_1)),$$
$$\mathsf{Sub}_\theta(LeftJoin(E,P^1,P^2),P_1) =$$
$$EqJoin^{\mathsf{var}(P_1)}_{\theta,\theta^1}(EqLeftJoin^{\mathsf{var}(P_1)}_{\theta^1,\theta^2}(E,\mathsf{Sub}_{\theta^1}(P^1,P_1),\mathsf{Sub}_{\theta^2}(P^2,P_1)),P_1\theta),$$
$$\mathsf{Sub}_\theta(Project(X,P'),P_1) = Project(X\cup\mathsf{var}(P_1)\cup(\mathsf{var}(P_1))\theta,\mathsf{Sub}_\theta(P'\theta_{\mathsf{var}(P')\setminus X},P_1)),$$
$$\mathsf{Sub}_\theta(Distinct(P'),P_1) = Distinct(\mathsf{Sub}_\theta(P',P_1)).$$

As formalised in the following claim, in the empty environment $\mathsf{Sub}_\theta(P,P_1)$ always gives the same answers as P, except that these answers have some additional variables defined; for example, the answers to P_1 are stored in $(\mathsf{var}(P_1))\theta$. Also, importantly, multiplicities are inessential in this construction, because everything we need is to preserve emptiness and non-emptiness of the set of answers.

Claim 4.19. *For any* $\mathsf{Sparql}_\star$ *patterns* P *and* P_1*, graph* G*, and mapping* $\mu_1 \in [P_1]^{\emptyset}_G$*, a mapping* μ *belongs to* $[P]^{\mu_1}_G$*, if and only if there is a mapping* μ_* *in* $[\mathsf{Sub}_\theta(P,P_1)]^{\emptyset}_G$ *for a renaming* θ *of* $\mathsf{var}(P_1)$ *such that* $\mu_*|_{\mathsf{var}(P_1)\cup\mathsf{var}(P)} = \mu$ *and* $\mu_*|_{(\mathsf{var}(P_1))\theta} = \mu_1\theta$.

The claim is shown in the Appendix by induction on the structure of P.

The key corollary of Claim 4.19 is that $[P]^{\mu_1}_G$ is empty for some $\mu_1 \in [P_1]^{\emptyset}_G$ if and only if $[\mathsf{Sub}_\theta(P,P_1)]^{\emptyset}_G$ does not have any mapping with a copy of μ_1 in $(\mathsf{var}(P_1))\theta$. Therefore, a pattern $Filter(\neg\mathsf{exists}(P_2),P_1)$, which tests the evaluation of P_2 for emptiness in answers to P_1 as environments, is $\emptyset$-equivalent to the following pattern, where $?x$, $?y$, and $?z$ are fresh variables, and θ is a renaming of $\mathsf{var}(P_1)$:

$$Filter(\neg\mathsf{bound}(?x),LeftJoin(\mathsf{eq}(\mathsf{var}(P_1),\theta),P_1,Join(\mathsf{Sub}_\theta(P_2,P_1),(?x,?y,?z)))).$$

Finally, patterns of the form $Filter(\mathsf{exists}(P_2),P_1)$ are dealt with by rewriting them to $\emptyset$-equivalent patterns $SetMinus(P_1,Filter(\neg\mathsf{exists}(P_2),P_1))$. □

Using Lemmas 4.16—4.18, we can now establish that the exists construct does not add any expressive power to any language between Sparql and Sparql_{PD}.

Theorem 4.20. *Let* $\text{Sparql}_\star$ *be one of the languages* Sparql, Sparql_P, Sparql_D, *and* Sparql_{PD}. *Then* $\text{Sparql}^\exists_\star$ *has the same expressive power as* $\text{Sparql}_\star$.

From this theorem and Corollary 3.12, we can conclude that neither *Project* nor *Distinct* in isolation add any expressive power in the presence of exists.

Corollary 4.21. *Languages* $\text{Sparql}^\exists_P$, $\text{Sparql}^\exists_D$, *and* $\text{Sparql}^\exists$ *have the same expressive power as* Sparql.

We conclude this section with the translation of the query in [4] to Sparql.

Example 4.22. Recall query (Q2) from Example 3.13. It is straightforward to check that (Q2) can be expressed using the exists operator as follows:

$$\mathit{Project}(\{?n\}, \mathit{Filter}(\mathsf{exists}(P_u), (?x, \mathit{name}, ?n))).$$

But then, applying Lemma 4.18 and some simplifications, we establish that (Q2) is equivalent to the Sparql query

$$\mathit{Project}(\{?n\}, \mathit{SetMinus}((?x, \mathit{name}, ?n), P)),$$

where P is

$$\mathit{Filter}(\neg\mathsf{bound}(?u), \mathit{LeftJoin}(\mathit{true}, (?x, \mathit{name}, ?n), \mathit{Join}(P_u, (?u, ?v, ?w)))).$$

5 VARIABLE ASSIGNMENT TO EXPRESSIONS

In addition to retrieving data, many applications require the ability to perform numeric computations. SQL provides a wide range of constructs to this effect: on the one hand, it allows for Boolean and arithmetic expressions for computing new data values, which can subsequently be assigned to variables; on the other hand, it is equipped with powerful constructs for grouping and aggregation. Formalising these features requires a significant extension to the relational algebra, which involves *grouping* and *generalised projection* operators, as well as *aggregate functions* (e.g., see [25, Chapter 5] for an excellent introduction).

The original SPARQL 1.0 recommendation, however, did not provide any such features. Although arithmetic expressions were available, computed values could only be used as part of filter conditions; thus, the means for assigning such values to variables and subsequently return them in query answers was missing. Similarly, SPARQL 1.0 did not provide any support for grouping and aggregation, which limited its applicability in many practical scenarios.

The standardisation of SPARQL 1.1 addressed these limitations. As in SQL, introducing these features into the language required an extended algebra, which, however, turned out rather unconventional when compared to the algebra underpinning SQL.

Our aim in this and the next sections is to provide an in-depth formal analysis of the SPARQL 1.1 assignment and aggregation algebra, which (to the best of our knowledge) has not been studied in the literature. In particular, in this section, we study the *Extend* operator, which captures the SPARQL 1.1 constructs `BIND` and `VALUES` and provides the algebraic means for assigning variables to expressions. In Section 5.1, we formalise the assignment operator, and in Section 5.2, we look at the expressive power of the languages with this operator.

5.1 Syntax and Semantics of Assignment

We give a semantics of *Extend* that correctly deals with expressions involving exists and analyse the expressive power of the variants of Sparql discussed so far when enriched with *Extend*.

Example 5.1. Consider the following query.
(Q4) *Return people's names with their salaries after 20% tax together with flags indicating whether they work in the CS department.*
This query can be written in SPARQL 1.1 as follows:

```
SELECT ?n (0.8 * ?s AS ?t) ?c
  WHERE {?x name ?n . ?x salary ?s . ?x department ?d BIND (?d=CS AS ?c)}.
```

Over graph G_{ex}, query (Q4) yields mappings such as $\{?n \mapsto \mathit{Alice}, ?t \mapsto 3200, ?c \mapsto \mathit{true}\}$, indicating Alice's net salary and the fact that she works in the *CS* department.

To support such queries, the SPARQL 1.1 algebra provides the *Extend* operator.

Definition 5.2. Given any extension $\text{Sparql}_\star$ of Sparql, the language $\text{Sparql}_{\star E}$ further extends $\text{Sparql}_\star$ by permitting patterns of the form $\mathit{Extend}(?x, E, P)$, where $?x$ is a variable not in $\text{var}(P)$, E is an expression, and P is a pattern. Let also $\text{var}(\mathit{Extend}(?x, E, P)) = \{?x\} \cup \text{var}(P)$.

The *evaluation* of a pattern $\mathit{Extend}(?x, E, P)$ over a graph G in an environment ν is defined as follows:

$$[\mathit{Extend}(?x, E, P)]^\nu_G =$$

$$\{\!\{\, \mu \mid \mu' \in [P]^\nu_G,\ [E]^{\mu'}_G \neq \mathit{error},\ \mu = \mu' \cup \{?x \mapsto [E]^{\mu'}_G\} \,\}\!\} \uplus \{\!\{\, \mu \mid \mu \in [P]^\nu_G,\ [E]^\mu_G = \mathit{error} \,\}\!\}.$$

Intuitively, *Extend* assigns to variable $?x$ the evaluation of E in each solution mapping to P, provided the evaluation does not yield *error*. Solution mappings in which E evaluates to *error* are also included, but with $?x$ unbound.

Example 5.3. Query (Q4) is translated into the normative algebra as follows:

$$\mathit{Project}(\{?n, ?t, ?c\}, \mathit{Extend}(?t, 0.8 * ?s,$$
$$\mathit{Extend}(?c, ?d = \mathit{CS}, \{(?x, \mathit{name}, ?n), (?x, \mathit{salary}, ?s), (?x, \mathit{department}, ?d)\}))).$$

Having defined the semantics, we next discuss the interactions between *Extend* patterns and exists expressions. To this end, we first illustrate that the issues with the normative semantics described in Section 4.1 are exacerbated in the presence of variable assignment. To make our argument precise, we need to formally extend the normative semantics to the new operator: $[\![\mathit{Extend}(?x, E, P)]\!]_G$ is defined in exactly the same way as the environment semantics $[\mathit{Extend}(?x, E, P)]^\nu_G$ in Definition 5.2, except that $[\![P]\!]_G$ and $[\![E]\!]^\mu_G$ are used instead of $[P]^\nu_G$ and $[E]^\mu_G$, respectively.

Example 5.4. Consider the SPARQL 1.1 query

```
SELECT ?x WHERE { ?x a b .
    FILTER EXISTS { ?x c d . { ?y ?y ?y BIND (?y AS ?x) } } }
```

and its algebraic translation to $\text{Sparql}^\exists_E$

$$\mathit{Project}(\{?x\}, \mathit{Filter}(\text{exists}(\mathit{Join}((?x, c, d), \mathit{Extend}(?x, ?y, (?y, ?y, ?y)))), (?x, a, b))).$$

Consider also the graph G consisting of the triples

$$(e, a, b), \quad (e, c, d), \quad (f, f, f).$$

As in Example 4.6, the normative semantics of the query on G is ill-defined as it requires evaluating $\mathit{Extend}(e, ?y, (?y, ?y, ?y))$, which is not a syntactically valid pattern.

A proposed fix to the normative semantics, discussed in the SPARQL 1.1 errata document [31] and associated forums, is to refrain from replacing the first argument of *Extend* when substituting

an environment. We argue, however, that this fix leads to unintuitive behaviour. Indeed, in this example, the exists expression should intuitively return *false*, because $\{?x \mapsto e\}$ is the only mapping satisfying the triple patterns $(?x, a, b)$ and $(?x, c, d)$, whereas $\{?x \mapsto f, ?y \mapsto f\}$ is the only mapping satisfying $\mathit{Extend}(?x, ?y, (?y, ?y, ?y))$, that is, there are no mappings in the evaluation of the first and second arguments of *Join* that have the same value of $?x$. However, the proposed fix would apply the mapping $\{?x \mapsto e\}$ to the pattern inside the exists expression to yield the pattern $\mathit{Join}((e, c, d), \mathit{Extend}(?x, ?y, (?y, ?y, ?y)))$, which evaluates to the mapping $\{?x \mapsto f, ?y \mapsto f\}$; as a result, the exists expression would evaluate to *true*.

In contrast to this fix, the environment semantics evaluates the exists expression in the query to *false*, as expected. Indeed, the environment ν, is which exists is evaluated, is $\{?x \mapsto e\}$, so $[\mathit{Extend}(?x, ?y, (?y, ?y, ?y))]^{\nu}_{G} = \emptyset$, because $[(?y, ?y, ?y)]^{\nu}_{G}$ consists of a single mapping $\{?x \mapsto e, ?y \mapsto f\}$, which is incompatible with the extension $\{?x \mapsto f\}$, and therefore

$$[\mathit{Join}((?x, c, d), \mathit{Extend}(?x, ?y, (?y, ?y, ?y)))]^{\nu}_{G} = \emptyset.$$

To summarise, the environment semantics of exists is always well-defined and provides intuitive answers.

We conclude this section with the observation that Theorem 4.13 generalises to the languages with *Extend*, that is, if the normative semantics is well-defined then it gives the same answers as the environment semantics.

Proposition 5.5. *For any* $\mathsf{Sparql}^{\exists}_{PDE}$ *pattern* P *and graph* G *such that* $[\![P]\!]_G$ *is well-defined it holds that* $[\![P]\!]_G = [P]^{\emptyset}_{G}$.

The proof, given in the Appendix, uses an extension of Lemma 4.12 to $\mathsf{Sparql}^{\exists}_{PDE}$.

5.2 Expressive Power of Languages with Assignment

First, we show that exists does not add expressive power if we enrich the languages considered so far with *Extend*. For this, we prove the following analogue of Theorem 4.20.

Theorem 5.6. *Let* $\mathsf{Sparql}_{\star}$ *be one of* Sparql_{E}, Sparql_{PE}, Sparql_{DE} *and* Sparql_{PDE}. *Language* $\mathsf{Sparql}^{\exists}_{\star}$ *has the same expressive power as* $\mathsf{Sparql}_{\star}$.

Proof. Lemmas 4.16 and 4.17 hold for language $\mathsf{Sparql}_{\star}$, so it is enough to extend Lemma 4.18 to $\mathsf{Sparql}_{\star}$. To this end, let

$$\mathsf{Sub}_{\theta}(\mathit{Extend}(?x, E, P'), P_1) = \mathit{Extend}(?x, E, \mathsf{Sub}_{\theta}(P', P_1)).$$

Claim 4.19 then generalises to $\mathsf{Sparql}_{\star}$ (with *Extend*) in a straightforward way: we just need to consider another case for the inductive step. The proof of this fact is in the Appendix, at the end of the proof of Claim 4.19. □

Next, we note that the proof of Theorem 3.10 goes through in the presence of *Extend* as well (in particular, Claim 3.11 generalises), meaning that every $\mathsf{Sparql}_{\star}$ query, for $\mathsf{Sparql}_{\star}$ one of Sparql_{PE}, Sparql_{DE}, or Sparql_{PDE} (and their extensions with exists), has an equivalent $\mathsf{Sparql}_{\star}$ query in s-normal form. Therefore, we obtain the following analogue of Corollary 3.12.

Corollary 5.7. *Languages* Sparql_{PE} *and* Sparql_{DE} *have the same expressive power as* Sparql_{E}.

Finally, the proof of Theorem 3.14 is unaffected by the presence of *Extend*, so we also have the following inexpressibility result.

Proposition 5.8. *Language* Sparql_{PDE} *is strictly more expressive than* Sparql_{E}.

Unsurprisingly, adding the *Extend* operator to any language considered so far increases its expressive power, since *Extend* provides queries with a means of returning values that do not occur in the queried graph.

Proposition 5.9. *Let* $\text{Sparql}_\star$ *be either* Sparql *or* Sparql_{PD}. *Language* $\text{Sparql}_{\star E}$ *is strictly more expressive than* $\text{Sparql}_\star$.

Proof. Let the query Q be defined as follows:

$$\mathit{Project}(\{?x\}, \mathit{Extend}(?x, \text{bound}(?y), (?y, a, a))).$$

Over the graph $G = \{(a, a, a)\}$, $[Q]^{\emptyset}_G$ contains the mapping $\{?x \mapsto \mathit{true}\}$. However, any Sparql_{PD} query Q' satisfies $\mu(?z) = a$ for each $\mu \in [Q']^{\emptyset}_G$ and $?z \in \text{dom}(\mu)$; as a result, Q' cannot be equivalent to Q. □

We conclude this section with the following observation. Language $\text{Sparql}^{\exists}_{PDE}$, as any other language considered in this article, possesses the property stated in Proposition 4.10: $\nu \subseteq \mu$ for every mapping $\mu \in [P]^{\nu}_G$. This property is natural, but it might not hold for some other potential extensions of Sparql. For such extensions, it seems to be reasonable to add the condition $\{?x \mapsto [E]^{\mu'}_G\} \sim \nu$ to the first argument of the bag union in the definition of the semantics of *Extend* (while for languages satisfying the above property this condition holds automatically).

6 AGGREGATION

We now turn our attention to aggregation in SPARQL 1.1. In Section 6.1, we discuss the normative algebra for aggregation and present a formalisation that makes ambiguous aspects of the specification precise. In Section 6.2, we demonstrate the power of the aggregate algebra by showing that it is capable of expressing variable assignment as well as nested queries in their full generality. In Section 6.3, we present a normal form, which leads to a substantial simplification of the SPARQL 1.1 aggregate algebra, where most of its unconventional aspects are eliminated.

6.1 Normative Aggregate Algebra

SPARQL 1.1 and SQL provide similar functionality for aggregation: grouping is first used to define equivalence classes of solution mappings over which aggregate functions are subsequently applied.

Example 6.1. Consider the following query.
(Q5) *Return the salary average in CS as well as the average over all other departments combined, where the averages are only relevant if the minimal salary exceeds 3500.*
It can be written in SPARQL 1.1 as follows:

```
SELECT ?d (AVG(?s) AS ?n)
     WHERE { ?x department ?d . ?x salary ?s }
GROUP BY (?d = CS AS ?d)
HAVING (MIN(?s) > 3500).
```

Note that the grouping expression `?d = CS` is Boolean. The value of this expression is then again assigned to variable `?d`. These two occurrences of `?d` are semantically unrelated: the one in `?d = CS` is local to the aggregation and bound in the `WHERE` clause, while the one being assigned to is global and referred to in `SELECT`. Over G_{ex}, (Q5) evaluates to $\{?d \mapsto \mathit{true}, ?n \mapsto 4500\}$, since the salary of *Charlie* is less than 3500, while the average over the salaries of *Alice* and *Bob* is 4500.

The SPARQL 1.1 aggregate algebra has several unconventional features when compared with SQL:

(F1) groups and aggregates are seen as first-class citizens of the algebra, which are defined independently using dedicated constructs *Group* and *Aggregate*;
(F2) grouping is allowed on arbitrary lists of expressions, and not just on lists of variables (as in the `GROUP BY` expression in query (Q5)); and
(F3) aggregation is also allowed on arbitrary lists of expressions, and not just on single expressions.

In what follows, we provide a rigorous formalisation of the normative aggregate algebra that addresses unspecified corner cases in the standard and makes ambiguous aspects of the specification precise. Our semantics also generalises the normative one to handle exists expressions correctly, and hence relies on the notion of environment introduced in Section 4.2.

We start our discussion by introducing groups as first-class citizens. A group induces a partitioning of a pattern's solution mappings into equivalence classes, each of which is determined by a key obtained from the evaluation of a list of expressions.

Definition 6.2. A *v-list* is a list of values in $\mathbf{T} \cup \{error\}$. The *evaluation* $[\mathbf{E}]^{\mu}_{G}$ of an expression list $\mathbf{E} = \langle E_1, \ldots, E_k \rangle$ over a graph G with respect to a mapping μ is the v-list $\langle [E_1]^{\mu}_{G}, \ldots, [E_k]^{\mu}_{G} \rangle$.

Definition 6.3. A *group* is a construct $Group(\mathbf{F}, P)$ with $\mathbf{F}$ a list of expressions and P a pattern. The *evaluation* $[\Gamma]_G$ of a group $\Gamma = Group(\mathbf{F}, P)$ over a graph G is a partial function from v-lists to multisets of mappings that is defined for all v-lists $Key = [\mathbf{F}]^{\mu}_{G}$ with $\mu \in [P]^{\emptyset}_{G}$ as follows:

$$[\Gamma]_G(Key) = \{\!\!\{ \mu' \mid \mu' \in [P]^{\emptyset}_{G}, [\mathbf{F}]^{\mu'}_{G} = Key \}\!\!\}.$$

Note that the evaluation of groups does not depend on environments. As we will see later on, groups can only appear in patterns that use the construct *AggregateJoin*, where the variables in the group are always local, that is, not visible outside the pattern. As a consequence, the environment in which the pattern is evaluated does not affect the semantics of the group.

As in SQL, *aggregate functions* in SPARQL 1.1 (e.g., `AVG` in query (Q5)) allow us to compute a single value for each group of solution mappings. In the relational case, they are functions from multisets of values to a single value [17]. Due to (F3), aggregate functions in SPARQL 1.1 deal with more complex structures involving multisets of v-lists. To handle them, SPARQL 1.1 introduces the function defined next.

Definition 6.4. Function Flatten maps each multiset Λ of v-lists to the multiset Θ of values in $\mathbf{T} \cup \{error\}$ whose base consists of values in Λ and whose multiplicity function is as follows, for each such value v:

$$\mathsf{card}_{\Theta}(v) = \sum\nolimits_{\lambda \in \Lambda} (\mathsf{card}_{\Lambda}(\lambda) \times n_{v,\lambda}),$$

where $n_{v,\lambda}$ is the number of appearances of v in λ.

SPARQL 1.1 provides aggregate functions analogous to those in SQL. Differences stem mostly from the treatment of lists and errors.

Definition 6.5. Let $\prec$ be a total order on values that extends the usual orders on literals and such that $error \prec b \prec u \prec \ell$ for any $b \in \mathbf{B}$, $u \in \mathbf{I}$, and $\ell \in \mathbf{L}$.

A SPARQL 1.1 *aggregate function* is one of the following functions, mapping multisets of v-lists Λ to values in $\mathbf{T} \cup \{error\}$, where $\Theta = \mathsf{Flatten}(\Lambda)$:

— $\mathsf{Count}(\Lambda) = \sum_{v \in \Theta, v \neq error} \mathsf{card}_{\Theta}(v)$;
— $\mathsf{Sum}(\Lambda)$ is $\sum_{v \in \Theta}(\mathsf{card}_{\Theta}(v) \times v)$ if all the values in Λ are numbers, and *error* otherwise;
— $\mathsf{Avg}(\Lambda)$ is 0 if $\mathsf{Count}(\Lambda) = 0$ and $\mathsf{Sum}(\Lambda) / \mathsf{Count}(\Lambda)$ otherwise (in particular, it is *error* if $\mathsf{Sum}(\Lambda) = error$);

— Min(Λ) is the ≺-minimal value in Θ if Θ ≠ ∅ and *error* otherwise;
— Max(Λ) is the ≺-maximal value in Θ if Θ ≠ ∅ and *error* otherwise.

Finally, CountD, SumD, and AvgD are defined as their counterparts Count, Sum, and Avg but applied to the multiset of v-lists obtained from Λ by removing duplicates.[3]

Note that *error* does not contribute to Count but may affect the results of other functions. We use Id as a synonym for Min whenever, by construction, Flatten(Λ) consists of a single value (with any multiplicity).

We define the aggregate construct, which computes a value for each group by means of aggregate functions.

Definition 6.6. An *aggregate* is a construct of the form $Aggregate(\mathbf{E}, f, \Gamma)$, for $\mathbf{E}$ a list of expressions, f an aggregate function, and $\Gamma = Group(\mathbf{F}, P)$ a group.

The *evaluation* $[A]_G$ of an aggregate $A = Aggregate(\mathbf{E}, f, \Gamma)$ over a graph G is the partial function from v-lists to values such that, for each *Key* in the domain of $[\Gamma]_G$,

$$[A]_G(Key) = f(\{\!|\, \Lambda \mid \mu \in [\Gamma]_G(Key),\ \Lambda = [\mathbf{E}]^{\mu}_G \,|\!\}).$$

Note that, for brevity, we omit a special case in SPARQL 1.1, which additionally allows aggregates to use Count and CountD with a distinguished symbol $*$ in place of the list $\mathbf{E}$. However, such special aggregates can be expressed in terms of the ones in Definition 6.6, as will be discussed in Section 9.

Finally, the normative SPARQL 1.1 algebra provides the *AggregateJoin* construct, which combines a list of aggregates $A_1, \ldots, A_n$ to form a pattern P. The semantics mandates that $[P]^{\nu}_G$ contain a mapping for each v-list in the domains of all $[A_i]_G$; each such mapping assigns variables $?x_i$ to record the values of aggregates A_i for the corresponding v-list.

Definition 6.7. Let $\text{Sparql}_\star$ be an extension Sparql. The language $\text{Sparql}_{\star A}$ extends $\text{Sparql}_\star$ by allowing patterns of the form $AggregateJoin_{\mathbf{x}}(\mathbf{A})$, with $\mathbf{x} = \langle ?x_1, \ldots, ?x_n \rangle$ a non-empty list of variables and $\mathbf{A} = \langle A_1, \ldots, A_n \rangle$ a list of aggregates of the same size.

The *evaluation* of a pattern $AggregateJoin_{\mathbf{x}}(\mathbf{A})$ over a graph G in an environment ν is as follows, where Λ is the intersection of the domains of all A_i:

$$[AggregateJoin_{\mathbf{x}}(\mathbf{A})]^{\nu}_G = \biguplus_{Key \in \Lambda} \{\!|\, \mu \mid \mu = \mu' \cup \nu, \mu' = \{?x_i \mapsto v \mid 1 \le i \le n,\ v = [A_i]_G(Key),\ v \neq error\} \,|\!\}.$$

We also set $\text{var}(AggregateJoin_{\mathbf{x}}(\mathbf{A})) = \{?x_1, \ldots, ?x_n\}$.

Example 6.8. Query (Q5) in Example 6.1 translates into the normative algebra as given next, where we have a single group over departments, and aggregates A_2 and A_3 for computing the average and minimum; an additional aggregate A_1 is required to store the keys of the groups and incorporate them into a pattern using *AggregateJoin*:

$$Project(\{?d, ?n\}, Extend(?n, ?v_2, Extend(?d, ?v_1, P_1))),$$

[3]According to this definition Θ may still contain repeated values, because duplicate elimination is applied before Flatten; this may seem counterintuitive, but we follow the SPARQL 1.1 specification here.

where pattern P_1 is defined as given next:

$$P_1 = \mathit{Filter}(?v_3 > 3500, \mathit{AggregateJoin}_{\langle ?v_1, ?v_2, ?v_3\rangle}(\langle A_1, A_2, A_3\rangle)),$$
$$A_1 = \mathit{Aggregate}(\langle ?d = CS\rangle, \mathsf{Id}, \mathit{Group}(\langle ?d = CS\rangle, P_2)),$$
$$A_2 = \mathit{Aggregate}(\langle ?s\rangle, \mathsf{Avg}, \mathit{Group}(\langle ?d = CS\rangle, P_2)),$$
$$A_3 = \mathit{Aggregate}(\langle ?s\rangle, \mathsf{Min}, \mathit{Group}(\langle ?d = CS\rangle, P_2)),$$
$$P_2 = \{(?x, \mathit{department}, ?d), (?x, \mathit{salary}, ?s)\}.$$

Note how variable $?d$ is handled: first an expression with the inner occurrence of $?d$ is aggregated to an auxiliary variable $?v_1$, and then the outer occurrence is assigned by *Extend* to the (trivial) expression $?v_1$. The two occurrences of $?d$ are, essentially, different variables, and the inner one is local to the *AggregateJoin* (as are all other variables of *AggregateJoin*). This justifies the evaluation of groups in the empty environment.

In conclusion, we note that, similarly to the case of *Extend*, Theorem 4.13 generalises to the languages with *AggregateJoin*, that is, if the normative semantics is well-defined then it gives the same answers as the environment semantics. To make this statement formal, we need to define the normative semantics for aggregates, which requires a generalisation of the notion of rectification: a pattern (with aggregates) is *rectified* if, same as in the basic case, for each subpattern $\mathit{Project}(X, P')$ no variable in $\mathrm{var}(P') \setminus X$ appears outside this subpattern and, additionally, for each subpattern $\mathit{AggregateJoin}_{\mathbf{x}}(\mathbf{A})$ no variable in $\mathbf{A}$ appears outside this subpattern. Each pattern can be converted to a rectified pattern in the same way as in the case without aggregation. This treatment of aggregates is justified by the fact that all variables in aggregates are essentially local, and their precise names should not influence the semantics. Then, $[\![\mathit{AggregateJoin}_{\mathbf{x}}(\mathbf{A})]\!]_G$ is defined in the same way as $[\mathit{AggregateJoin}_{\mathbf{x}}(\mathbf{A})]^{\nu}_G$ except that (*i*) $\mathbf{A}$ is rectified, (*ii*) the condition $\mu = \mu' \cup \nu$ in Definition 6.7 is replaced by $\mu = \mu'$, and (*iii*) subexpressions and subpatterns are evaluated in the obvious way according to the normative semantics rather than the environment semantics.

Proposition 6.9. *For any* $\mathsf{Sparql}^{\exists}_{PDEA}$ *pattern* P *and graph* G *such that* $[\![P]\!]_G$ *is well-defined it holds that* $[\![P]\!]_G = [P]^{\emptyset}_G$.

The proof, given in the Appendix, is based on an extension of Lemma 4.12 to $\mathsf{Sparql}^{\exists}_{PDEA}$.

6.2 Expressive Power of Languages with Aggregates

Operators *Group*, *Aggregate*, and *AggregateJoin* provide a great deal of power and flexibility to the query language. We next show that, when added to Sparql, these operators are sufficiently expressive to capture all forms of query nesting and variable assignment discussed so far.

Example 6.10. To see how *Extend* can be expressed in terms of aggregates, consider the pattern $P_0 = \mathit{Extend}(?x, ?y_1 + ?y_2, (?y_1, p, ?y_2))$. This pattern can be equivalently expressed by a Sparql_A pattern P_1 defined as follows:

$$A_{?y_i} = \mathit{Aggregate}(\langle ?y_i\rangle, \mathsf{Id}, \mathit{Group}(\langle ?y_1, ?y_2\rangle, (?y_1, p, ?y_2))), \qquad i = 1, 2,$$
$$A_{?x} = \mathit{Aggregate}(\langle ?y_1 + ?y_2\rangle, \mathsf{Id}, \mathit{Group}(\langle ?y_1, ?y_2\rangle, (?y_1, p, ?y_2))),$$
$$P_1 = \mathit{AggregateJoin}_{\langle ?x, ?y_1, ?y_2\rangle}(A_{?x}, A_{?y_1}, A_{?y_2}).$$

To eliminate *Extend* over arbitrary patterns rather than simple triple patterns, this construction needs to be generalised to account for partial mappings as well as mappings with multiplicities greater than 1 (in this example it is not needed, because P_0 is equivalent to $\mathit{Distinct}(P_0)$ and all mappings in its evaluation have the same domain).

Theorem 6.11. *Language* Sparql_{PDEA} *has the same expressive power as* Sparql_{A}.

Proof. We start by expressing patterns $Extend(?x, E, P)$ in Sparql_{PA}. For an enumeration $\mathbf{x} = \langle ?x_1, \ldots, ?x_n \rangle$ of $\mathsf{var}(P)$, let

$$\begin{aligned} P' &= AggregateJoin_{\langle ?x, ?x_1, \ldots, ?x_n \rangle}(\langle A, A_1, \ldots, A_n \rangle), \text{ where} \\ A_i &= Aggregate(\langle ?x_i \rangle, \mathsf{Id}, Group(\mathbf{x}, P)), \quad 1 \leq i \leq n, \\ A &= Aggregate(\langle E \rangle, \mathsf{Id}, Group(\mathbf{x}, P)). \end{aligned}$$

By construction, the evaluation of P' consists of the same mappings as the evaluation of the pattern $Extend(?x, E, P)$, but all with multiplicities being 1: in particular, for any distinct mapping μ in $[P]^{\emptyset}_{G}$ there is a v-list in the domain of the evaluation of $Group(\mathbf{x}, P)$, namely the list of values of $\mathbf{x}$ under μ, where the unbound variables are represented as *error*; each A_i then evaluates to the ith value in the list, while *AggregateJoin* combines these values back to the mapping extended with the evaluation of E in $?x$ in such a way that variables corresponding to *error* values are left unbound.

Then the following pattern is fully equivalent to $Extend(?x, E, P)$, where θ is a renaming of $\mathsf{var}(P)$:

$$Project(\{?x\} \cup \mathsf{var}(P), Filter(\mathsf{eq}(\mathsf{var}(P), \theta), Join(P, P'\theta))).$$

Patterns of the form $Distinct(P)$ are expressed similarly: we can construct P' as above, but without aggregate A and variable $?x$, and take P' as an equivalent pattern.

Finally, *Project* can be pushed upwards through Sparql operators as in Theorem 3.10. Thus, it suffices to show that *Project* can be eliminated from any group Γ of the form $Group(\mathbf{F}, Project(X, P))$. For this, replace $Project(X, P)$ in Γ by $P\theta'$, with θ' a renaming of $\mathsf{var}(P) \setminus X$. □

Since all variables in groups are considered local, that is, groups are always evaluated in empty environments, we have the following corollary of Theorem 5.6, saying that exists does not add any expressive power in the presence of aggregation.

Corollary 6.12. *Language* $\mathsf{Sparql}^{\exists}_{PDEA}$ *has the same expressive power as* Sparql_{PDEA}.

Proof. Lemmas 4.16 and 4.17 hold for language Sparql_{PDEA}, so it is enough to extend Lemma 4.18 to Sparql_{PDEA}. To this end, let

$$\mathsf{Sub}_{\theta}(AggregateJoin_{\mathbf{x}}(\mathbf{A}), P_1) = Join(AggregateJoin_{\mathbf{x}}(\mathbf{A}), Copy_{\theta}(P_1)).$$

Claim 4.19 straightforwardly generalises to Sparql_{PDEA} by adding another base case. □

We conclude this section by showing that aggregation does provide additional expressive power to the language.

Proposition 6.13. *Language* Sparql_{A} *is strictly more expressive than* Sparql_{PDE}.

Proof. Let $\mathcal{G}$ be the set of all graphs that contain no numbers.

On the one hand, for every Sparql_{PDE} query Q and graph G in $\mathcal{G}$, the maximal number occurring in any mapping μ in $[Q]^{\emptyset}_{G}$ is bounded by a function in the size $|Q|$ of Q and the maximal number n occurring in Q. Indeed, whenever $\mu(?x)$ is a number, Q contains a subpattern of the form $Extend(?x, E, P)$. Therefore, by a straightforward induction on the nesting depth d of *Extend* in Q, one can establish that all numbers in μ are bounded by $n^{|Q|^d}$. In particular, this bound does not depend on G.

On the other hand, let Q be the Sparql_{A} query

$$Project(\{?c\}, Extend(?c, ?v, AggregateJoin_{\langle ?v \rangle}(\langle A \rangle))),$$

where $A = Aggregate(\langle ?x\rangle, \mathsf{Count}, Group(\langle\rangle, (?x, ?y, ?z)))$; this query is obtained from the SPARQL 1.1 query

```
SELECT (COUNT(?x) AS ?c) WHERE { ?x ?y ?z }.
```

Then, for every $G \in \mathcal{G}$, the multiset $[Q]^{\emptyset}_G$ consists of a single mapping μ such that $\mu(?c)$ is the number of triples in G. Since $\mu(?c)$ cannot be bounded independently of G, it follows that Q has no equivalent query in $\mathrm{Sparql}|_{PDE}$. □

6.3 Normalisation and Simplification

We next show that features (F2) and (F3) in the normative algebra do not add expressive power: every query can be rewritten into a normal form where grouping is only allowed over lists of variables rather than arbitrary expressions, and aggregation is done only over singleton lists. Moreover, our normal form dispenses with the functions CountD, SumD, and AvgD, and hence shows that it suffices to consider aggregate functions that do not involve duplicate elimination.

Definition 6.14. A $\mathrm{Sparql}|_A$ pattern is in *a-normal form* if each group is of the form $Group(\mathbf{y}, P)$ with $\mathbf{y}$ a list of variables and each aggregate is of the form $Aggregate(\langle E\rangle, f, \Gamma)$ with f different from CountD, SumD, and AvgD.

Example 6.15. The algebraic translation of query (Q5), given in Example 6.8, is not in a-normal form, since grouping is not performed over variables but a complex expression $?d = CS$. However, (Q5) can be easily brought to a-normal form by replacing every occurrence of $Group(\langle ?d = CS\rangle, P_2)$ by the group

$$Group(\langle ?y\rangle, Extend(?y, ?d = CS, P_2)),$$

where $?y$ is a fresh variable, and then eliminating *Extend* as done in the proof of Theorem 6.11. Note that elimination of *Extend* from otherwise a-normal formulas yields a-normal formulas.

Similarly, the following aggregate A_0 is not a-normal, since it uses AvgD and aggregates over a list with more than one component:

$$Aggregate(\langle ?x - ?z, ?z\rangle, \mathsf{AvgD}, Group(\langle ?y\rangle, (?x, ?y, ?z))).$$

To bring A_0 to a-normal form, we first rewrite it to the following equivalent aggregate A_1, which uses Avg instead of AvgD and fresh variables $?u$, $?v$, $?w$:

$$\begin{aligned}
A_{?u} &= Aggregate(\langle ?x - ?z\rangle, \mathsf{Id}, Group(\langle ?x - ?z, ?z, ?y\rangle, (?x, ?y, ?z))),\\
A_{?v} &= Aggregate(\langle ?z\rangle, \mathsf{Id}, Group(\langle ?x - ?z, ?z, ?y\rangle, (?x, ?y, ?z))),\\
A_{?w} &= Aggregate(\langle ?y\rangle, \mathsf{Id}, Group(\langle ?x - ?z, ?z, ?y\rangle, (?x, ?y, ?z))),\\
P_1' &= AggregateJoin_{\langle ?u, ?v, ?w\rangle}(\langle A_{?u}, A_{?v}, A_{?w}\rangle),\\
A_1 &= Aggregate(\langle ?u, ?v\rangle, \mathsf{Avg}, Group(\langle ?w\rangle, P_1')).
\end{aligned}$$

Note that pattern P_1' "simulates" $(?x, ?y, ?z)$ in the sense that, for any graph G and v-list λ, we have $\lambda \in [\langle ?x - ?z, ?z, ?y\rangle]^{\mu}_G$ for some $\mu \in [(?x, ?y, ?z)]^{\emptyset}_G$, if and only if $\lambda \in [\langle ?u, ?v, ?w\rangle]^{\mu'}_G$ for some $\mu' \in [P_1']^{\emptyset}_G$. However, by construction, $[P_1']^{\emptyset}_G$ contains each mapping at most once, and each pair of distinct mappings in $[P_1']^{\emptyset}_G$ yields two distinct v-lists, which justifies using Avg in A_1 in place of AvgD in A_0.

Note also that removing AvgD introduced grouping over a complex expression, which can in turn be removed as discussed above, resulting in an aggregate A_2 of the form $Aggregate(\langle ?u, ?v\rangle, \mathsf{Avg}, \Gamma)$, for Γ an a-normal $\mathrm{Sparql}|_A$ rewriting of $Group(\langle ?w\rangle, P_1')$.

Thus, it remains to remove the aggregation over $\langle ?u, ?v\rangle$, for which it suffices to note that A_2 is equivalent to $\mathit{Aggregate}(\langle(?u + ?v)/2\rangle, \mathsf{Avg}, \Gamma)$, which is in a-normal form.

Next, we show that a-normalisation is always feasible.

THEOREM 6.16. *Every* Sparql_A *pattern has an* $\emptyset$*-equivalent* Sparql_A *pattern in a-normal form.*

PROOF. We first show that aggregate functions with duplicate elimination can be rewritten using their ordinary counterparts. Let $f\mathsf{D} \in \{\mathsf{CountD}, \mathsf{SumD}, \mathsf{AvgD}\}$ and

$$A_1 = \mathit{Aggregate}(\mathbf{E}, f\mathsf{D}, \mathit{Group}(\mathbf{F}, P_1)),$$

with $\mathbf{E} = \langle E_1, \dots, E_m\rangle$ and $\mathbf{F} = \langle F_1, \dots, F_k\rangle$. Then A_1 is equivalent to the aggregate A_1' defined as follows, where $\mathbf{x} = \langle ?x_1, \dots, ?x_m\rangle$ and $\mathbf{y} = \langle ?y_1, \dots, ?y_k\rangle$ are lists of fresh variables, and $\cdot$ denotes list concatenation:

$$\begin{aligned}
A_{?x_i} &= \mathit{Aggregate}(\langle E_i\rangle, \mathsf{Id}, \mathit{Group}(\mathbf{E}\cdot\mathbf{F}, P_1)), \quad 1 \le i \le m,\\
A_{?y_j} &= \mathit{Aggregate}(\langle F_j\rangle, \mathsf{Id}, \mathit{Group}(\mathbf{E}\cdot\mathbf{F}, P_1)), \quad 1 \le j \le k,\\
P_1' &= \mathit{AggregateJoin}_{\mathbf{x}\cdot\mathbf{y}}(\langle A_{?x_1}, \dots, A_{?x_m}, A_{?y_1}, \dots, A_{?y_k}\rangle),\\
A_1' &= \mathit{Aggregate}(\mathbf{x}, f, \mathit{Group}(\mathbf{y}, P_1')).
\end{aligned}$$

Second, we argue that grouping over lists of expressions can be reduced to grouping over lists of variables by exploiting *Extend*. For this, we note that an aggregate

$$A_2 = \mathit{Aggregate}(\mathbf{E}, f, \mathit{Group}(\mathbf{F}, P_2)),$$

with $\mathbf{F} = \langle F_1, \dots, F_k\rangle$, is equivalent to an aggregate A_2' defined as follows, where $\mathbf{y} = \langle ?y_1, \dots, ?y_k\rangle$ is a list of fresh variables:

$$\begin{aligned}
P_2' &= \mathit{Extend}(?y_1, F_1, \dots \mathit{Extend}(?y_k, F_k, P_2)\dots),\\
A_2' &= \mathit{Aggregate}(\mathbf{E}, f, \mathit{Group}(\mathbf{y}, P_2')).
\end{aligned}$$

By Theorem 6.11, *Extend* in P_2' is inessential as it is expressible using normalised grouping and aggregation constructs (in particular, no grouping over non-trivial expressions is introduced by the transformation in the proof of Theorem 6.11).

To summarise, every aggregate A can be rewritten to the form $\mathit{Aggregate}(\mathbf{E}, f, \Gamma)$, where $\mathbf{E} = \langle E_1, \dots, E_m\rangle$, $\Gamma = \mathit{Group}(\mathbf{y}, P)$, and f is not CountD, SumD, or AvgD.

In the last step of the proof, we argue that every such A can be rewritten into an equivalent aggregate A' in a-normal form by reducing the list $\mathbf{E}$ to a single expression; for example, computing Avg over a list $\langle E_1, \dots, E_m\rangle$ is equivalent to aggregating over the expression $(\Sigma_{i=1}^m E_i)/m$. Unlike the previous two steps, this step is sensitive to the particular aggregate functions available in SPARQL 1.1, and hence, we perform a case-by-case analysis.

Let $f = \mathsf{Count}$. Let $?x_1, \dots, ?x_m$ and $?z_1, \dots, ?z_m$ be sets of fresh variables. We define

$$A' = \mathit{Aggregate}\left(\left\langle \sum\nolimits_{i=1}^{m} ?x_i \right\rangle, \mathsf{Sum}, \mathit{Group}(\mathbf{y}, P_m)\right),$$

where the patterns P_i are as follows:

$$\begin{aligned}
P_0 &= \mathit{Extend}(?z_m, E_m, \dots \mathit{Extend}(?z_1, E_1, P)\dots),\\
P_{i+1} &= \mathit{Union}(\mathit{Filter}(\mathsf{bound}(?z_{i+1}), \mathit{Extend}(?x_{i+1}, 1, P_i)),\\
&\qquad \mathit{Filter}(\neg\mathsf{bound}(?z_{i+1}), \mathit{Extend}(?x_{i+1}, 0, P_i))).
\end{aligned}$$

Let $f = \mathsf{Sum}$. We define $A' = \mathit{Aggregate}(\langle\sum_{i=1}^m E_i\rangle, \mathsf{Sum}, \Gamma)$.

Let $f = \mathsf{Avg}$. We define $A' = \mathit{Aggregate}(\langle(\sum_{i=1}^{m} E_i)/m\rangle, \mathsf{Avg}, \Gamma)$.

Let $f = \mathsf{Min}$. We define $A' = \mathit{Aggregate}(\langle ?x\rangle, \mathsf{Min}, \mathit{Group}(\mathsf{y}, P'))$, where $?x$ is a fresh variable and

$$P' = \mathit{Union}(\mathit{AggregateJoin}_{\langle ?x\rangle}(\langle A_1\rangle), \ldots \\ \mathit{Union}(\mathit{AggregateJoin}_{\langle ?x\rangle}(\langle A_{m-1}\rangle), \mathit{AggregateJoin}_{\langle ?x\rangle}(\langle A_m\rangle))\ldots),$$

and $A_i = \mathit{Aggregate}(\langle E_i\rangle, \mathsf{Min}, \mathit{Group}(\mathsf{y}, P'))$ for $i = 1, \ldots m$.

The case $f = \mathsf{Max}$ is analogous to the case for Min.

In all cases, $[A]_G = [A']_G$ for every G by construction and the definitions of the aggregate functions. □

The normal form in Definition 6.14 already provides a significant simplification of the algebra. Indeed, it shows that features (F2) and (F3) are inconsequential. Our normal form also suggests that the definition of aggregate functions can be made more transparent than that in the standard: not only the functions involving duplicate elimination can be dispensed with, but also the function Flatten is inessential, since aggregation can always be performed over a single expression rather than a list.

We next show that feature (F1) is also immaterial; that is, we can collapse the *Group*, *Aggregate* and *AggregateJoin* constructs into a single pattern operator without affecting the expressive power of the language. This further simplification not only brings the normative aggregate algebra closer to its relational counterpart, but can also be exploited to make the mapping from the SPARQL 1.1 syntax into the algebra much more direct and transparent.

The following definition specifies the aforementioned combined operator.

Definition 6.17. Let $\mathsf{Sparql}_\star$ be an extension of Sparql. The language $\mathsf{Sparql}_{\star As}$ extends $\mathsf{Sparql}_\star$ by allowing patterns of the form $\mathit{GroupAgg}(Z, ?x, f, E, P)$, where Z is a set of variables, called *grouping variables*, $?x$ is another variable, called *aggregation variable*, f is an aggregate function, E is an expression, and P a is pattern.

Given a pattern $\mathit{GroupAgg}(Z, ?x, f, E, P)$, a graph G, and a mapping $\mu \in [P]^{\emptyset}_G$, let

$$v_\mu = f(\{\!|\, v \mid \mu' \in [P]^{\emptyset}_G,\ \mu'|_Z = \mu|_Z,\ v = [E]^{\mu'}_G \,|\!\}).$$

Then, the *evaluation* $[\mathit{GroupAgg}(Z, ?x, f, E, P)]^{\nu}_G$ in an environment ν over G is the multiset with the base set

$$\{\mu' \mid \mu \in [P]^{\emptyset}_G,\ v_\mu \neq \mathit{error}, \mu' = \mu|_Z \cup \{?x \mapsto v_\mu\} \cup \nu\} \cup \\ \{\mu' \mid \mu \in [P]^{\emptyset}_G,\ v_\mu = \mathit{error},\ \mu' = \mu|_Z \cup \nu\}$$

and multiplicity 1 for each mapping in the base set. We also set

$$\mathsf{var}(\mathit{GroupAgg}(Z, ?x, f, E, P)) = Z \cup \{?x\}.$$

The construct *GroupAgg* is close to the grouping operator in the relational algebra: Z represents the set of grouping variables, $?x$ is the fresh variable storing the aggregation result, f is the aggregate function, and E is the expression (often a variable) we are aggregating over. The treatment of environments in Definition 6.17 is consistent with the normative algebra operators. Specifically, $[\mathit{GroupAgg}(Z, ?x, f, E, P)]^{\nu}_G$ is defined by means of the evaluation $[P]^{\emptyset}_G$ in the empty environment.

Example 6.18. Query (Q5) in Example 6.8 can be written in $\text{Sparql}^{\exists}_{As}$ as follows (exists is just used for succinctness and can be dispensed with as usual):

$$\begin{aligned}
&\mathit{Project}(\{?d, ?n\}, \mathit{Filter}(\mathsf{exists}(P_2), P_1)), \text{ where}\\
&\quad P_1 = \mathit{GroupAgg}(\{?d\}, ?n, \mathsf{Avg}, ?s, P_3),\\
&\quad P_2 = \mathit{Filter}(?v > 3500, \mathit{GroupAgg}(\{?d\}, ?v, \mathsf{Min}, ?s, P_3)),\\
&\quad P_3 = \mathit{GroupAgg}(\{?x, ?c, ?s\}, ?d, \mathsf{Id}, ?c = \mathit{CS}, \{(?x, \mathit{department}, ?c), (?x, \mathit{salary}, ?s)\}).
\end{aligned}$$

The following theorem shows that the *GroupAgg* construct captures grouping and aggregation in SPARQL 1.1. While the reduction from Sparql_{As} to Sparql_{A} is straightforward, the other direction generalises Example 6.18. There, to express *AggregateJoin*, one needs to join *GroupAgg* subpatterns in a way that forces the respective mappings to coincide on a given set of variables (rather than just being compatible on the set). For this, we once again employ techniques developed in Sections 3.1 and 4.4.

Theorem 6.19. *For any extension* $\text{Sparql}_{\star P}$ *of* Sparql_{P}*, language* $\text{Sparql}_{\star PAs}$ *has the same expressive power as* $\text{Sparql}_{\star PA}$.

Proof. We first show that each $\text{Sparql}_{\star PAs}$ pattern

$$P = \mathit{GroupAgg}(\{?z_1, \ldots, ?z_k\}, ?x, f, E, P')$$

has a pattern in $\text{Sparql}_{\star PA}$ that is ν-equivalent to P for any environment ν. For this, let

$$\Gamma = \mathit{Group}(\langle ?z_1, \ldots, ?z_k\rangle, P').$$

We then use aggregates $A_i = \mathit{Aggregate}(\langle ?z_i\rangle, \mathsf{Id}, \Gamma)$, $1 \le i \le k$, to record the values of the grouping variables in each group; finally, we capture the value of E using $A = \mathit{Aggregate}(\langle E\rangle, f, \Gamma)$. Then, for any ν, P is ν-equivalent to

$$\mathit{AggregateJoin}_{\langle ?z_1, \ldots, ?z_k, ?x\rangle}(\langle A_1, \ldots, A_k, A\rangle).$$

For the other direction, consider a $\text{Sparql}_{\star PA}$ query Q with all *AggregateJoin* subpatterns in the a-normal form. We will replace each such subpattern P with a $\text{Sparql}_{\star PAs}$ subpattern P'' and then prove that the resulting $\text{Sparql}_{\star PAs}$ query is equivalent to Q. To this end, let

$$\begin{aligned}
P &= \mathit{AggregateJoin}_{\langle ?x_1, \ldots, ?x_n\rangle}(\langle A_1, \ldots, A_n\rangle), \text{ where}\\
A_i &= \mathit{Aggregate}(\langle E_i\rangle, f_i, \mathit{Group}(\mathbf{y}_i, P_i)), \quad 1 \le i \le n.
\end{aligned}$$

Assume, without loss of generality, that all $\mathbf{y}_i$ are of the same length k, since otherwise the evaluation of P is empty. Let $Z = \{?z_1, \ldots, ?z_k\}$ be fresh variables and, for $1 \le i \le n$, let θ_i be a renaming of variables $\mathbf{y}_i$ to the corresponding variables in Z. We simulate each A_i by the pattern

$$P'_i = \mathit{GroupAgg}(Z, ?x_i, f_i, E_i\theta_i, P_i\theta_i).$$

We then combine these patterns as follows, where each θ'_i, $1 \le i < n$, is a renaming of Z to fresh Z_i:

$$\begin{aligned}
P'' &= \mathit{Project}(\{?x_1, \ldots, ?x_n\}, P'), \text{ where}\\
P' &= \mathit{Filter}(\mathsf{eq}(Z, \theta'_1), \ldots \mathit{Filter}(\mathsf{eq}(Z, \theta'_{n-1}), \mathit{Join}(P'_1\theta'_1, \ldots \mathit{Join}(P'_{n-1}\theta'_{n-1}, P'_n)\ldots))\ldots).
\end{aligned}$$

We argue that P'' is ν-equivalent to P for every environment ν in which P can be evaluated as a subpattern of Q, that is, P can be safely replaced by P'' in Q without changing the semantics of the overall query. Note that, for any graph G, each $\mu \in [P]^{\nu}_{G}$ is of the form $\{?x_1 \mapsto u_1, \ldots, ?x_n \mapsto u_n\}$, and it corresponds to a single key $\mathbf{v}^{\mu} = \langle v^{\mu}_1, \ldots, v^{\mu}_k\rangle$, such that $(\mathbf{v}^{\mu} \mapsto u_i) \in [A_i]_G$ for all $1 \le i \le n$. Given such μ, let μ' be the mapping with $\mathsf{dom}(\mu') \subseteq Z$, such that $\mu'(?z_j) = v^{\mu}_j$ whenever $v^{\mu}_j \neq \mathit{error}$ and $?z_j \notin \mathsf{dom}(\mu')$ otherwise. Since variables Z_i and Z do not occur in Q, they cannot be bound in

any environment ν in which P is evaluated. Therefore, by construction, $\mu \in [P]^{\nu}_{G}$ if and only if the mapping

$$\mu \cup \mu'\theta'_1 \cup \cdots \cup \mu'\theta'_{n-1} \cup \mu'$$

is in $[P']^{\nu}_{G}$ for any such ν. Since

$$\{?x_1, \ldots, ?x_n\} \cap (Z_1 \cup \cdots \cup Z_{n-1} \cup Z) = \emptyset,$$

it follows that $[P'']^{\nu}_{G}$ and $[P]^{\nu}_{G}$ have the same mappings. Finally, the multiplicity of every mapping in each of these multisets is 1, by definition of $[P]^{\nu}_{G}$ and by construction of $[P'']^{\nu}_{G}$, so P can be replaced by P'' in Q. □

Note that in the theorem, the reduction from $\text{Sparql}_{\star PAs}$ to $\text{Sparql}_{\star PA}$ does not use *Project*. The reduction for the other direction introduces *Project*, which may occur at an intermediate level of the overall query. However, if $\text{Sparql}_{\star PAs}$ allows us to "push" projection up through all other pattern constructs, it follows that languages $\text{Sparql}_{\star As}$ and $\text{Sparql}_{\star A}$ are equivalent as well, because the subpattern $Project(X, P)$ in $GroupAgg(Z, ?x, f, E, Project(X, P))$ can be replaced by just P without changing the semantics. In particular, we have the following corollary.

Corollary 6.20. *Language* Sparql_{As} *has the same expressive power as* Sparql_{A}.

7 PROPERTY PATHS

Property paths were introduced in SPARQL 1.1 to allow for navigational queries, which give users a tool to discover how different resources in an RDF graph are connected. Property paths have been formally studied in recent articles (e.g., see [7, 40, 48]), and their complexity and expressive power is by now relatively well-understood.

Existing work on property paths is, however, restricted either to set semantics [40] or to bag semantics from the early draft of the specification [7, 48], which has changed since then. In this section, we study the normative bag semantics of property paths and describe the interplay between property paths and aggregation.

We start our discussion by introducing property paths syntactically in the algebra.

Definition 7.1. Path expressions are inductively defined as follows:

- $Link(u)$ is a path expression if u is an IRI,
- $Inv(PP)$ is a path expression if so is PP,
- $Seq(PP_1, PP_2)$ and $Alt(PP_1, PP_2)$ are path expressions if so are both PP_i,
- $NPS(U)$ is a path expression if U is a set of IRIs,
- $ZeroOrOnePath(PP)$, $ZeroOrMorePath(PP)$, and $OneOrMorePath(PP)$ are path expressions if so is PP.

Intuitively, a path expression specifies a possible route between two terms in a graph. In particular, $Link(u)$ represents all pairs of terms that are directly connected via property u; $Inv(PP)$ inverses the pairs represented by PP; $Seq(PP_1, PP_2)$ and $Alt(PP_1, PP_2)$ correspond to path concatenation and union, respectively; $NPS(U)$ (standing for "negated property set") represents pairs connected by a property not mentioned in U; $ZeroOrOnePath(PP)$ represents all pairs connected by PP and pairs with the same first and second elements; finally, $ZeroOrMorePath(PP)$ and $OneOrMorePath(PP)$) represent pairs of terms that are connected by a chain of PP edges, possibly empty or nonempty, respectively.

The *Path* construct provides a means to build patterns with path expressions.

Definition 7.2. Let $\text{Sparql}_\star$ be an extension Sparql. The language $\text{Sparql}_{\star PP}$ extends $\text{Sparql}_\star$ by allowing triple patterns of the form $Path(t_1, PP, t_2)$, where $t_1, t_2 \in \mathbf{I} \cup \mathbf{L} \cup \mathbf{X}$ and PP is a path expression that is not of the form $Link(u)$, $Inv(Link(u))$, or $Seq(PP_1, PP_2)$.[4]

Example 7.3. Consider the following query.
(Q6) *Compute all managers together with the total salary of the employees managed by them (directly or indirectly).*
The query can be written in SPARQL 1.1 as follows:

```
SELECT ?m (SUM(?s) AS ?n)
    WHERE { ?m manages+ ?x . ?x salary ?s }
GROUP BY ?m.
```

Query (Q6) translates into Sparql_{AsPP} as

$$Project(\{?m, ?n\}, GroupAgg(\{?m\}, ?n, \text{Sum}, ?s,$$
$$\{\, Path(?m, OneOrMorePath(Link(manages)), ?x), (?x, salary, ?s)\, \})).$$

Note that this query combines the navigational features of property paths with grouping and aggregation: property paths are used to identify managers and the employees they manage (directly or indirectly), whereas the *GroupAgg* construct is used to group employees according to their manager and to sum over their salaries.

We now specify the normative bag semantics of path expressions and patterns. To handle some cases uniformly, we extend Definition 7.2 and allow PP in $Path(t_1, PP, t_2)$ to be arbitrary path expressions and t_1 and t_2 to be blank nodes.

Definition 7.4. The *evaluation* $[Path(t_1, PP, t_2)]^\nu_G$ of a triple pattern $Path(t_1, PP, t_2)$ with property paths over a graph G in an environment ν is defined inductively as follows, assuming that mappings extend to terms as identity, and $\text{var}(t_1, t_2)$ is the set of all variables among t_1 and t_2:

- if $PP = Link(u)$ for IRI u, then it is $[(t_1, u, t_2)]^\nu_G$;
- if $PP = Inv(PP')$, then it is $[Path(t_2, PP', t_1)]^\nu_G$;
- if $PP = Seq(PP_1, PP_2)$, then it is $[Project(\text{var}(t_1, t_2), Join(P_1, P_2))]^\nu_G$ for $P_1 = Path(t_1, PP_1, ?x)$, $P_2 = Path(?x, PP_2, t_2)$, and $?x$ a fresh variable;
- if $PP = Alt(PP_1, PP_2)$, then it is $[Union(P_1, P_2)]^\nu_G$ for $P_1 = Path(t_1, PP_1, t_2)$ and $P_2 = Path(t_1, PP_2, t_2)$;
- if $PP = NPS(U)$, then it is the multiset with the base set

$$\{\mu \mid \exists \mu', \exists u \notin U : \mu = \mu' \cup \nu, \text{dom}(\mu') = \text{var}(t_1, t_2), (\mu'(t_1), u, \mu'(t_2)) \in G\}$$

  and multiplicity 1 for each such μ;
- if $PP = ZeroOrOnePath(PP')$, then it is the multiset consisting of all mappings μ (with multiplicity 1), such that either $\mu \in [Path(t_1, PP', t_2)]^\nu_G$ or $\mu = \mu' \cup \nu$ where $\text{dom}(\mu') = \text{var}(t_1, t_2)$ and $\mu'(t_1) = \mu'(t_2)$;
- if $PP = ZeroOrMorePath(PP')$, then it is the multiset consisting of all mappings $\mu = \mu' \cup \nu$ (with multiplicity 1), such that $\text{dom}(\mu') = \text{var}(t_1, t_2)$ and there is a sequence of terms $s_0, \ldots, s_n$, $n \geq 0$, with $s_0 = \mu'(t_1)$, $s_n = \mu'(t_2)$, and $[Path(s_i, PP', s_{i+1})]^\nu_G$ nonempty for all $0 \leq i < n$;

[4]The aforementioned path expressions are excluded explicitly in the standard: their counterparts are allowed in SPARQL 1.1 syntax but are rewritten away in the translation to the algebra. Their inclusion would not alter any results of this article.

— if $PP = \mathit{OneOrMorePath}(PP')$, then it is defined the same as for $\mathit{ZeroOrMorePath}(PP')$, except that n is restricted to be strictly greater 0.

The *evaluation* of a BGP with triple patterns $T_1, \ldots, T_n$, some of which have path expressions, over G in ν is then

$$[\mathit{Join}(\ldots \mathit{Join}(T_1, T_2), \ldots T_n)]^{\nu}_{G}.$$

Example 7.5. Query (Q6) in Example 7.3 computes mappings of the form $\{?m \mapsto v, ?n \mapsto w\}$ where v is an employee and w is the total salary of all employees managed (directly or via lower-ranked managers) by v. For instance, if we extend our example graph G_{ex} with the triples $(a, \mathit{manages}, b)$ and $(a, \mathit{manages}, c)$, we would obtain $\{?m \mapsto a, ?n \mapsto 8000\}$ as the only answer mapping.

Moving on to the expressive power of languages with property paths, we first note that all the relevant previous results, that is, Theorems 3.14, 4.13, 4.20, 5.6, 6.11, and 6.13, Corollaries 3.12, 5.7, 6.12, and 6.20, and Proposition 5.9 hold in the presence of property paths, with the proofs extending the existing ones in the obvious way.

On the other hand, using techniques in [45], it is possible to show that query (Q6) from Example 7.3 (as a variation of a reachability query) cannot be expressed in Sparql_{As}, and hence property paths add expressive power to this language. Same can be shown for Sparql, Sparql_{PD}, Sparql_{E}, and Sparql_{PDE}.

At first glance, the normative algebra allows for an intricate interaction between bag and set semantics in path expressions: some of the constructs keep multiplicities different from 1, while others performs implicit duplicate elimination. However, it is not difficult to see that this impression is wrong, and, as we formalise next, the algebra can be easily simplified in such a way that all paths are evaluated as sets.

Definition 7.6. Let *set path expressions* be defined exactly the same as path expressions, but using constructs *sLink*, *sInv*, and so on, instead of *Link*, *Inv*, and so on, respectively.

For any extension $\mathsf{Sparql}_{\star}$ of Sparql, let the language $\mathsf{Sparql}_{\star sPP}$ further extend $\mathsf{Sparql}_{\star}$ by permitting triple patterns of the form $\mathit{sPath}(t_1, \mathit{sPP}, t_2)$, where $t_1, t_2 \in \mathbf{I} \cup \mathbf{L} \cup \mathbf{X}$ and *sPP* is a set path expression that is not of the form $\mathit{sLink}(u)$, $\mathit{sInv}(\mathit{sPP}')$, $\mathit{sSeq}(\mathit{sPP}_1, \mathit{sPP}_2)$, or $\mathit{sAlt}(\mathit{sPP}_1, \mathit{sPP}_2)$.

The *evaluation* $[\mathit{sPath}(t_1, \mathit{sPP}, t_2)]^{\nu}_{G}$ over a graph G in an environment ν is defined in exactly the same way as for a triple pattern with the corresponding path expression, except that it always works with set semantics, that is, $\mathrm{card}_{\Omega}(\mu) = 1$ for any $\mu \in [\mathit{sPath}(t_1, \mathit{sPP}, t_2)]^{\nu}_{G}$.

Example 7.7. Consider the Sparql_{PP} pattern

$$P = \mathit{Path}(?x, \mathit{Alt}(\mathit{OneOrMorePath}(\mathit{Link}(p)), \mathit{Seq}(\mathit{Link}(q), \mathit{Link}(r))), ?y).$$

By definition, P can be restated as the following Sparql_{PPP} pattern, for $?z$ fresh:

$$\mathit{Union}(\mathit{Path}(?x, \mathit{OneOrMorePath}(\mathit{Link}(p)), ?y), \mathit{Project}(\{?x, ?y\}, \{(?x, q, ?z),\ (?z, r, ?y)\})).$$

Since the remaining path subpattern in this union is equivalent to its set path counterpart $\mathit{sPath}(?x, \mathit{sOneOrMorePath}(\mathit{sLink}(p)), ?y)$, it follows that P can be expressed in Sparql_{PsPP}.

As suggested by the above example, in the presence of projection, path expressions can always be replaced by set path expressions.

Proposition 7.8. *For any extension* $\mathsf{Sparql}_{\star}$ *of* Sparql_{P}*, language* $\mathsf{Sparql}_{\star sPP}$ *has the same expressive power as* $\mathsf{Sparql}_{\star PP}$.

The proof of this proposition is in the Appendix.

8 ANALYTIC AGGREGATE QUERIES

An increasing number of applications of semantic technologies require the analysis of data for effective decision making. In the data warehousing literature, this activity is referred to as online analytical processing (OLAP), and it involves the execution of complex aggregate queries.

In this section, we briefly recap the multidimensional model underpinning data warehousing applications and existing W3C standards for publishing multidimensional data using RDF. We then give semantics for cube and window queries in SPARQL and finally provide a rewriting of such queries into our algebra Sparql_{As}.

8.1 Data Representation

The natural way of thinking about OLAP queries is in terms of a multidimensional data model, which defines a collection of numeric measures (also called observations) based on a set of dimensions. For instance, in an online retail application a typical measure would be the total amount of merchandise sold in a given currency, and typical dimensions include the kind of product sold, the location and time of the sale, and the method of payment. We can think of the sales information as being arranged in a hypercube, where the coordinates of each sale value are given by specific value IDs for the product, location, time, and method of payment dimensions.

The RDF Data Cube specification [23] is a standard for publishing such multidimensional data on the Semantic Web, where dimensions and measures are represented as RDF properties of a specific kind.

Example 8.1. Consider our running example, where we take as measure the employee salary and define as dimensions the department, the employee name, and the departmental role. The RDF Data Cube vocabulary allows us to define the measure and dimensions using the IRIs *qb:MeasureProperty* and *qb:DimensionProperty*, respectively. For instance, the salary measure and the role dimension are defined as follows:

$$(salary, rdf{:}type, qb{:}MeasureProperty), \quad (salary, rdfs{:}range, xsd{:}decimal),$$
$$(role, rdf{:}type, qb{:}DimensionProperty), \quad (role, rdfs{:}range, xsd{:}string).$$

Each individual observation is represented as an instance of type *qb:Observation*. In the basic case, values of each of the dimensions and measures are attached directly to the observation. The standard introduces a collection of integrity constraints that must be satisfied by all datasets. For example, it requires that each observation has a value for each corresponding dimension or that every dimension has a defined range.

Example 8.2. Our running example graph G_{ex} can be written using the RDF Data Cube Vocabulary as a graph G'_{ex} specified in Figure 2 (for brevity, triples involving *rdfs:range*, *qb:MeasureProperty* and *qb:DimensionProperty* are omitted). Note that the data for *Bob* in G_{ex} has been split into two observations in G'_{ex}. The W3C specification requires a salary value for each tuple of name, department, and role values; as a result, we need to specify the salary earned by *Bob* for each of its individual departmental affiliations (specifically, 2500 per affiliation).

8.2 Cube Queries

In the relational world, a *cube query* involves aggregating over all possible subsets of dimensions at the same time. The query output involves the values of both the measure and the dimensions, and a special symbol is used to indicate that a particular dimension has been aggregated over [26]. In SQL, cube queries are supported by extending the `GROUP BY` construct with the `CUBE` keyword, which indicates that grouping must be performed on all subsets of the grouping attributes. In most

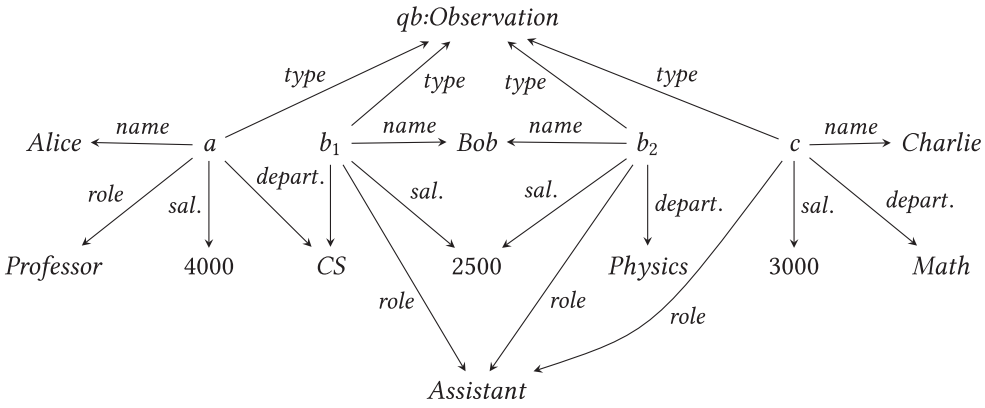

Fig. 2. Example RDF graph G'_{ex}.

cases, SQL cube queries are executed over databases that are structured according to a *star schema*—a faithful relational representation of multidimensional data. However, this is not required and the algebraic semantics of cube SQL queries is defined in general for any relational representation of the data.

Our algebra can be extended with a cube operator analogously to the relational case.

Definition 8.3. For Y a set of variables, $?x$ a variable not in Y, f an aggregate function, E an expression, and P a pattern, $Cube(Y, ?x, f, E, P)$ is a pattern, whose *evaluation* $[Cube(Y, ?x, f, E, P)]^{\nu}_{G}$ over a graph G in an environment ν is as follows, where *all* is a special value not in $\mathbf{T}$:

$$\biguplus_{Z \subseteq Y} \{\!| \mu' \mid \mu \in [GroupAgg(Z, ?x, f, E, P)]^{\nu}_{G}, \mu' = \mu \cup \{?y \mapsto all \mid ?y \in Y \setminus Z\} |\!\}.$$

The special term *all* in the definition is introduced analogously to the relational formalisation (see [26]). Rather than a dedicated value *all*, we could have left the relevant variables unbound; this, however, could yield counterintuitive results when further applying operators such as *Join*. The semantics of *Cube* suggests a simple translation to our algebra using *GroupAgg*, *Extend*, and *Union*, which is a straightforward adaptation to the SPARQL setting of the well-known translation of cube queries into standard relational algebra operators.

Proposition 8.4. *The extension of* $\text{Sparql}^{\exists}_{PDEAs}$ *that allows for Cube has the same expressive power as* Sparql_{As}.

Proof. By Theorems 6.11 and 6.19 as well as Corollaries 6.12 and 6.20, it is enough to show that for any pattern $P = Cube(Y, ?x, f, E, P')$ with P' in the extension of $\text{Sparql}^{\exists}_{PDEAs}$ with *Cube* there exists a pattern in the extension that is ν-equivalent to P for any environment ν and has smaller depth of nesting of *Cube*.

Given any $P = Cube(Y, ?x, f, E, P')$, let, for $Z \subseteq Y$,

$$P_Z = Extend(?y_1, all, \ldots Extend(?y_n, all, GroupAgg(Z, ?x, f, E, P')) \ldots),$$

where $\{?y_1, \ldots, ?y_n\} = Y \setminus Z$. Then, the pattern

$$Union(P_{Z_1}, \ldots Union(P_{Z_{m-1}}, P_{Z_m}) \ldots),$$

where $Z_1, \ldots, Z_m$ are all the subsets of Y, is as required. □

For RDF data conforming to the W3C Data Cube specification, we can define a simplified version of the cube operator where the construction of the pattern argument in the operator is implicit.

This simplified operator can thus be seen as a "macro" and its semantics is defined by expanding its definition using the generic cube operator.

Definition 8.5. For $D = \{d_1, \ldots, d_k\}$ a set of dimension properties (i.e., IRIs), $?x$ a variable, f an aggregate function, and m a measure property (i.e., an IRI), the pattern $Cube(D, ?x, f, m)$ is an abbreviation for the pattern

$$Cube(\{?d_1, \ldots, ?d_k\}, ?x, f, ?m, \{(?z, m, ?m), (?z, d_1, ?d_1), \ldots, (?z, d_k, ?d_k)\}).$$

Example 8.6. Consider the pattern $Cube(\{department, role\}, ?x, \mathsf{Sum}, salary)$, which aggregates *salary* over *department* and *role*. Over graph G'_{ex}, the pattern evaluates to the multiset consisting of the following mappings, all with multiplicity 1:

$$\{?department \mapsto all, ?role \mapsto all, ?x \mapsto 12000\},$$
$$\{?department \mapsto all, ?role \mapsto Professor, ?x \mapsto 4000\},$$
$$\{?department \mapsto all, ?role \mapsto Assistant, ?x \mapsto 8000\},$$
$$\{?department \mapsto CS, ?role \mapsto all, ?x \mapsto 6500\},$$
$$\{?department \mapsto Physics, ?role \mapsto all, ?x \mapsto 2500\},$$
$$\{?department \mapsto Math, ?role \mapsto all, ?x \mapsto 3000\},$$
$$\{?department \mapsto CS, ?role \mapsto Professor, ?x \mapsto 4000\},$$
$$\{?department \mapsto CS, ?role \mapsto Assistant, ?x \mapsto 2500\},$$
$$\{?department \mapsto Physics, ?role \mapsto Assistant, ?x \mapsto 2500\},$$
$$\{?department \mapsto Math, ?role \mapsto Assistant, ?x \mapsto 3000\}.$$

8.3 Window-Based Queries

Window-based queries are heavily used in data analysis applications. In the relational case, a window identifies a set of rows "around" each individual row in a relation. Once a window has been identified, we can aggregate over the window for each row and extend the row with the result. There is an important difference between groups and windows: the former partition the rows of a relation and compute a value for each partition, whereas the latter compute a different value for each row according to its associated window. Windows were introduced in the SQL:2003 standard and their semantics and efficient evaluation were studied in [16, 42].

The following definition introduces windows in the context of SPARQL. Intuitively, the expression F in the definition specifies a window as a multiset of "surrounding" mappings for a specific mapping μ; in turn, *AggWindow* is used to compute an aggregate value for each mapping based on its corresponding window, which is then appended using a fresh variable (provided that the computed value is not *error*).

Definition 8.7. Given a pattern P, a renaming θ of $\mathsf{var}(P)$ to fresh variables, an expression F over $\mathsf{var}(P) \cup \mathsf{var}(P\theta)$, a variable $?x \notin \mathsf{var}(P)$, an aggregate function f, and an expression E over $\mathsf{var}(P)$, $AggWindow(\theta, F, ?x, f, E, P)$ is also a pattern. For a pattern $AggWindow(\theta, F, ?x, f, E, P)$, graph G, environment ν, and mapping μ, let

$$v_\mu = f\left(\left\{\!\!\left\{ v \mid \mu' \in [P]^\nu_G,\ [F]^{\mu \cup \mu'\theta}_G = true, v = [E]^{\mu'}_G \right\}\!\!\right\}\right).$$

Then the *evaluation* $[AggWindow(\theta, F, ?x, f, E, P)]^\nu_G$ is

$$\{\!\!\{ \mu \mid \mu' \in [P]^\nu_G,\ v_{\mu'} \neq error,\ \mu = \mu' \cup \{?x \mapsto v_{\mu'}\} \}\!\!\} \ \uplus \ \{\!\!\{ \mu \mid \mu \in [P]^\nu_G,\ v_\mu = error \}\!\!\}.$$

Example 8.8. Consider the following pattern, which, for every employee, computes the average of all salaries that are lower than their salary:

$$AggWindow(\theta, ?s > ?s', ?n, \mathsf{Avg}, ?s, (?x, salary, ?s)),$$

where θ is the renaming sending each variable $?u$ to $?u'$. For instance, on the graph G_{ex}, we obtain the mappings

$$\{?x \mapsto a, ?s \mapsto 4000, ?n \mapsto 3000\},$$
$$\{?x \mapsto b, ?s \mapsto 5000, ?n \mapsto 3500\},$$
$$\{?x \mapsto c, ?s \mapsto 3000, ?n \mapsto 0\}.$$

Note that in the last mapping, $?n$ is bound to 0, since there are no salaries lower than 3000 (see Definition 6.5).

Finally, note that the *AggWindow* pattern can be equivalently expressed in $\mathsf{Sparql}|_{As}$:

$$GroupAgg(\{?x, ?s\}, ?n, \mathsf{Avg}, ?s, Filter(?s > ?s', Join((?x, salary, ?s), (?x', salary, ?s')))).$$

As suggested by the above example, *AggWindow* can also be expressed in our algebra. The difficulty in the proof of this result is to show that the exists operator can be eliminated in the presence of windows.

Proposition 8.9. *The extension of* $\mathsf{Sparql}^{\exists}_{PDEAs}$ *that allows for AggWindow has the same expressive power as* $\mathsf{Sparql}|_{As}$.

Proof. We first prove that the extension of $\mathsf{Sparql}^{\exists}_{PDEAs}$ with *AggWindow* has the same expressive power as the extension of $\mathsf{Sparql}|_{PDEAs}$ with only *AggWindow*; then, we show that any $P = AggWindow(\theta', F, ?x, f, E, P')$ is $\emptyset$-equivalent to a $\mathsf{Sparql}|_{PDEAs}$ pattern, which, together with Theorems 6.11 and 6.19 as well as Corollaries 6.12 and 6.20, would imply the statement of the theorem.

For the first part, we need to generalise Theorem 4.20 to the extension of $\mathsf{Sparql}|_{PDEAs}$ with *AggWindow*. By Theorem 6.19, it is enough to generalise it to the extension of $\mathsf{Sparql}|_{PDEA}$ with *AggWindow*. Lemmas 4.16 and 4.17 hold for this language, so we only need to generalise Lemma 4.18. By Corollary 6.12, it is already generalised to $\mathsf{Sparql}|_{PDEA}$, so we just need to extend it to *AggWindow*. Let

$$\mathsf{Sub}_\theta(AggWindow(\theta', F, ?x, f, E, P'), P_1) = AggWindow(\theta'', F \wedge \mathsf{eq}(\mathsf{var}(P_1), \theta''), ?x, f, E, \mathsf{Sub}_\theta(P', P_1)),$$

where θ'' is an extension of θ' to $\mathsf{var}(\mathsf{Sub}_\theta(P', P_1)) \setminus \mathsf{var}(P')$ that sends all these variables to fresh copies. Claim 4.19 then generalises: we just need to consider yet another case for the induction step.

For the second part of the proof, consider a pattern $P = AggWindow(\theta', F, ?x, f, E, P')$. It is $\emptyset$-equivalent to the pattern

$$Project(\mathsf{var}(P') \cup \{?x\}, Filter(\mathsf{eq}(\mathsf{var}(P'), \theta''), Join(P', P''\theta''))),$$

with θ'' another renaming of $\mathsf{var}(P')$ to fresh variables and

$$P'' = GroupAgg(\mathsf{var}(P'), ?x, f, E, Filter(F, Join(P', P'\theta'))).$$

Note that P'' and P coincide if interpreted as sets; the additional transformations are applied to P'' to obtain the correct multiplicities. □

Note that the $\emptyset$-equivalent patterns in the second part of the proof of Proposition 8.9 may not be ν-equivalent for some environments ν, so a straightforward proof such as the one for Proposition 8.4 does not work and the first part is required.

9 ADDITIONAL CONSIDERATIONS

We made several simplifying assumptions that made us deviate from the standard. First, we omitted the auxiliary SPARQL 1.0 pattern operator *Diff* and SPARQL 1.1 operator *Minus*. Second, we have considered only expressions already available in SPARQL 1.0, whereas SPARQL 1.1 defines a richer language for expressions. Third, we have assumed that triple patterns contain no blank nodes, which are, however, allowed in the standard. Fourth, we have assumed that the result of a query is a multiset of mappings, where the standard defines it as a list. Fifth, we did not consider counting functions aggregating over a special symbol $*$ instead of a list of expressions. Finally, we have omitted the non-deterministic aggregate functions Sample and GroupConcat.

The non-deterministic aggregate functions can be treated similarly to the other aggregate functions. We next discuss how the other five assumptions affect our results.

Difference and Minus. The pattern operator *Diff* in the SPARQL 1.0 algebra does not have a counterpart in the SPARQL 1.0 syntax, but plays an auxiliary role in the definition of the semantics of *LeftJoin*. Instead, a slightly different operator *Minus* is introduced in SPARQL 1.1. Next, we give the formal definitions of these two difference operators and present ways to express them via the core Sparql operators.

Definition 9.1. If E is an expression and P_1, P_2 are patterns, then $\mathit{Diff}(E, P_1, P_2)$ is a pattern with the following semantics, for a graph G and environment ν:

$$[\mathit{Diff}(E, P_1, P_2)]^{\nu}_G = \left\{\!\!\left\{ \mu \mid \mu \in [P_1]^{\nu}_G, \forall \mu_2 \in [P_2]^{\nu}_G . \left(\mu \nsim \mu_2 \text{ or } [E]^{\mu\cup\mu_2}_G \neq \mathit{true}\right) \right\}\!\!\right\}.$$

If P_1 and P_2 are patterns, then $\mathit{Minus}(P_1, P_2)$ is a pattern with the following semantics, for a graph G and environment ν:

$$[\mathit{Minus}(P_1, P_2)]^{\nu}_G = \{\!\!\{ \mu \mid \mu \in [P_1]^{\nu}_G, \forall \mu_2 \in [P_2]^{\nu}_G . (\mu \nsim \mu_2 \text{ or } (\mathrm{dom}(\mu) \cap \mathrm{dom}(\mu_2)) \setminus \mathrm{dom}(\nu) = \emptyset) \}\!\!\}.$$

Note that the semantics of *LeftJoin* can be stated using *Diff* as follows:

$$[\mathit{LeftJoin}(E, P_1, P_2)]^{\nu}_G = [\mathit{Filter}(E, \mathit{Join}(P_1, P_2))]^{\nu}_G \uplus [\mathit{Diff}(E, P_1, P_2)]^{\nu}_G.$$

This is the only way in which *Diff* is used in the specification.

The following example illustrates the difference between *Diff*, *Minus*, and operator *SetMinus*, introduced in Section 3.1.

Example 9.2. Let P_1, P_2 be patterns and G a graph such that $[P_1]^{\emptyset}_G$ and $[P_2]^{\emptyset}_G$ are the multisets with base sets $\{\mu_1, \mu_1'\}$ and $\{\mu_2\}$, respectively, where

$$\mu_1 = \{?x \mapsto a\}, \qquad \mu_1' = \{?x \mapsto a, ?y \mapsto b\}, \qquad \mu_2 = \{?y \mapsto b\},$$

and all mappings have multiplicity 1. Then, $[\mathit{Diff}(\mathit{true}, P_1, P_2)]^{\emptyset}_G$ is empty, $[\mathit{Minus}(P_1, P_2)]^{\emptyset}_G$ consists of a single occurrence of μ_1, while $[\mathit{SetMinus}(P_1, P_2)]^{\emptyset}_G = [P_1]^{\emptyset}_G$, that is, has both μ_1 and μ_1'.

As we already said, both *Diff* and *Minus* are expressible via the core Sparql operators, that is, these operators do not add any expressive power to the languages considered in this article. In particular, it is well-known that, for an expression E and patterns P_1, P_2 in any extension of Sparql, $\mathit{Diff}(E, P_1, P_2)$ is ν-equivalent, for any ν, to the pattern

$$\mathit{Filter}(\neg\mathrm{bound}(?x), \mathit{LeftJoin}(E, P_1, \mathit{Join}(P_2, (?x, ?y, ?z)))),$$

where $?x$, $?y$, and $?z$ are fresh variables (see, e.g., [9]). In turn, $Minus(P_1, P_2)$ is $\emptyset$-equivalent to $Diff(E, P_1, P_2\theta)$, where θ is a variable renaming of $\mathrm{var}(Minus(P_1, P_2))$ and

$$E = \bigwedge\nolimits_{?x \in \mathrm{dom}(\theta)} (\mathrm{bound}(?x) \wedge \mathrm{bound}(?x\theta) \rightarrow ?x = ?x\theta) \wedge \bigvee\nolimits_{?x \in \mathrm{dom}(\theta)} (\mathrm{bound}(?x) \wedge \mathrm{bound}(?x\theta)).$$

This equivalence is introduced in [37] for set semantics, but an immediate inspection shows that it holds under bag semantics as well. To deal with *Minus* in non-empty environments, we first have to eliminate exists, as it is done in Theorem 4.20, and then apply the equivalence for the empty environment. To this end, we show the following proposition.

Proposition 9.3. *Let* $\mathrm{Sparql}_\star$ *be any language considered in this article. The extension of* $\mathrm{Sparql}_\star$ *allowing for* exists *and Minus has the same expressive power as* $\mathrm{Sparql}_\star$.

The proof of this proposition is in the Appendix.

Expressions. We focus on two constructs due to their potential implications: ternary if, which computes one of two expressions depending on the evaluation of a third one, and coalesce, which allows us to "recover" from errors.

Definition 9.4. If E_1, E_2, and E_3 are expressions, then $\mathsf{if}(E_1, E_2, E_3)$ is an expression with the following semantics: for a given μ and G, the *evaluation* $[\mathsf{if}(E_1, E_2, E_3)]^\mu_G$ is $[E_2]^\mu_G$ if $[E_1]^\mu_G = \mathit{true}$, it is $[E_3]^\mu_G$ if $[E_1]^\mu_G = \mathit{false}$, and *error* otherwise. Let also $\mathrm{var}(\mathsf{if}(E_1, E_2, E_3)) = \mathrm{var}(E_1) \cup \mathrm{var}(E_2) \cup \mathrm{var}(E_3)$.

For $\mathbf{E} = \langle E_1, \ldots, E_n \rangle$ a non-empty list of expressions, $\mathsf{coalesce}(\mathbf{E})$ is an expression with the following semantics: for μ and G, the *evaluation* $[\mathsf{coalesce}(\mathbf{E})]^\mu_G$ is *error* if $[E_i]^\mu_G = \mathit{error}$ for each $1 \leq i \leq n$, and $[E_j]^\mu_G$ otherwise, for the smallest j with $[E_j]^\mu_G \neq \mathit{error}$. Let also $\mathrm{var}(\mathsf{coalesce}(\mathbf{E})) = \mathrm{var}(E_1) \cup \cdots \cup \mathrm{var}(E_n)$.

Example 9.5. The expression $\mathsf{if}(?h < 12, \text{"am"}, \text{"pm"})$ returns the string "am" if the value of the variable $?h$ is less than 12, and "pm" otherwise; $\mathsf{coalesce}(\langle ?x, 0 \rangle)$ returns the value of $?x$ if $?x$ is bound and 0 otherwise.

These expressions can be rewritten in terms of Sparql expressions, and hence their introduction is immaterial to our results in this article. Indeed, for any pattern P and expression E with a subexpression $\mathsf{if}(E_1, E_2, E_3)$, for expressions E_1, E_2, and E_3, all in any extension of Sparql, as well as for any environment ν, the pattern $Filter(E, P)$ is ν-equivalent to

$$Union(Union(Filter(E_1 \wedge E_2', P), Filter(\neg E_1 \wedge E_3'], P)), Filter(E_1', SetMinus(P, Filter(E_1 \vee \neg E_1, P)))),$$

where E_i', for $i = 1, 2, 3$, is obtained from E by replacing $\mathsf{if}(E_1, E_2, E_3)$ by E_i.

Similarly, for any P and E with a subexpression $\mathsf{coalesce}(\langle E_1, E_2 \rangle)$, for expressions E_1 and E_2, all in any extension of Sparql, as well as for any environment ν, the pattern $Filter(E, P)$ is ν-equivalent to

$$Union(Filter((E_1 \vee \neg E_1) \wedge E_1', P), Filter(E_2', SetMinus(P, Filter(E_1 \vee \neg E_1, P)))),$$

where E_i', for $i = 1, \ldots, 2$, is obtained from E by replacing $\mathsf{coalesce}(\langle E_1, E_2 \rangle)$ by E_i; coalesce over longer lists is treated analogously (while coalesce over singleton lists is vacuous). Similarly, if and coalesce can be eliminated from other constructs that mention expressions in all the languages considered in this article, such as *LeftJoin* and *GroupAgg*. Therefore, these expression constructs do not increase the expressive power of any considered extension of Sparql.

Blank Nodes. According to the SPARQL 1.1 standard, triple patterns in BGPs may contain blank nodes in subject and object positions. Roughly speaking, such blank nodes are treated as variables for the purposes of pattern-graph matching; in contrast to variables, however, the scope of blank nodes is confined to the BGP in which they occur. The effect of blank nodes within BGPs can be simulated in our algebra by using *Project*: any BGP B with variables X and blank nodes N is ν-equivalent, for any ν, to the pattern $Project(X, B\xi)$, where ξ is a renaming of N to fresh variables. As shown in Section 3 (Corollary 3.12), projection in patterns does not add expressive power to Sparql, and hence neither does allowing blank nodes in BGPs. Note, however, that in combination with *Distinct* at pattern level, blank nodes do lead to an increase in expressive power, since they introduce projection, and hence the proof of Theorem 3.14 can be easily adapted.

Lists of Solutions. We have defined the semantics of patterns and queries uniformly in terms of multisets of mappings, which simplifies the algebra by avoiding numerous type conversions. The standard, however, defines query evaluation in terms of lists of mappings. It also allows for solution modifiers, such as *Slice* and *OrderBy*, the semantics of which depends on the order of mappings in the solution sequence of the query.

We next argue that dispensing with lists altogether does not essentially affect any of our results. Given a multiset Ω of mappings, let Λ_Ω be the set of all lists of mappings that coincide with Ω when disregarding the order of elements. Also, let h be the function that translates queries from any language $\text{Sparql}_\star$ considered in this article into the normative one by adding the necessary type conversions between multisets and lists. Then, for every $\text{Sparql}_\star$ query Q and graph G, we have $[Q]_G = \Omega$ if and only if $[h(Q)]^{std}_G \in \Lambda_\Omega$, where $[\,\cdot\,]^{std}_{\cdot}$ is the list evaluation function defined in the SPARQL standard. This correspondence demonstrates an additional benefit of using multisets rather than lists: every query evaluates to a *unique* multiset (except those using the non-deterministic aggregate Sample), whereas the evaluation to a list of solutions is non-deterministic even for simple queries.

Counting Aggregates over $*$. Apart from aggregates of the form $Aggregate(\mathbf{E}, f, \Gamma)$ with $\mathbf{E}$ a list of expressions, SPARQL 1.1 allows for aggregates $A = Aggregate(*, f, \Gamma)$, where $*$ is a distinguished symbol, f is Count or CountD, and $\Gamma = Group(\mathbf{F}, P)$ is a group. The evaluation $[A]_G$ of such A over a graph G is the partial function from v-lists to values defined as follows, for each *Key* in the domain of $[\Gamma]_G$: if f is Count, then $[A]_G(Key)$ is the sum of the cardinalities of all mappings in $[\Gamma]_G(Key)$, and if f is CountD, then it is the number of distinct mappings in $[\Gamma]_G(Key)$. These aggregates can be rewritten to the algebra in Section 6.1, since $Aggregate(*, \text{Count}, \Gamma)$ is equivalent to $Aggregate(\text{bound}(?x) \vee \neg\text{bound}(?x), \text{Count}, \Gamma)$, for $?x$ an arbitrary variable, while $Aggregate(*, \text{CountD}, Group(\mathbf{F}, P))$ is equivalent to $Aggregate(*, \text{Count}, Group(\mathbf{F}, Distinct(P)))$. Note that the expression $\text{bound}(?x) \vee \neg\text{bound}(?x)$, which always evaluates to *true*, is needed in the first equivalence to avoid disregarding mappings with unbound $?x$. Also, $?x$ is not required to appear in Γ; in fact, if $?x$ is a fresh variable, then the expression can be simplified to $\neg\text{bound}(?x)$.

10 RELATED WORK

Query nesting, variable assignment, and aggregation are key features of the database query language SQL (e.g., see [25] for an excellent textbook covering these topics). In addition to being widely-used in practice, the formal properties of query nesting and aggregation are are also well-understood from a formal point of view in the relational setting. These include semantics and expressive power [18, 27, 28, 30, 44—47, 54]; query equivalence [17, 21]; and query rewriting using views [20]. Analytic OLAP queries have also been studied extensively in the relational case (e.g., see [16, 26, 42]).

$$\left\{\begin{array}{lll}\text{Sparql}, & \text{Sparql}_{P}, & \text{Sparql}_{D},\\ \text{Sparql}^{\exists}, & \text{Sparql}^{\exists}_{P}, & \text{Sparql}^{\exists}_{D}\end{array}\right\} \;<\; \left\{\begin{array}{l}\text{Sparql}_{PD},\\ \text{Sparql}^{\exists}_{PD}\end{array}\right\}$$

$$\wedge \qquad\qquad \wedge$$

$$\left\{\begin{array}{lll}\text{Sparql}_{E}, & \text{Sparql}_{PE}, & \text{Sparql}_{DE},\\ \text{Sparql}^{\exists}_{E}, & \text{Sparql}^{\exists}_{PE}, & \text{Sparql}^{\exists}_{DE}\end{array}\right\} \;<\; \left\{\begin{array}{l}\text{Sparql}_{PDE},\\ \text{Sparql}^{\exists}_{PDE}\end{array}\right\} \;<\; \left\{\begin{array}{lll}\text{Sparql}_{A}, & \ldots, & \text{Sparql}_{PDEA},\\ \text{Sparql}^{\exists}_{A}, & \ldots, & \text{Sparql}^{\exists}_{PDEA},\\ \text{Sparql}_{As}, & \ldots, & \text{Sparql}_{PDEAs},\\ \text{Sparql}^{\exists}_{As}, & \ldots, & \text{Sparql}^{\exists}_{PDEAs}\end{array}\right\}$$

Fig. 3. Main expressivity results of this article. Here, Sparql is the core SPARQL 1.0 algebra, while indices abbreviate its extensions as follows: P stands for subqueries with *Project*, D for subqueries with *Distinct*, $\exists$ for subqueries with exists, E for variable assignment *Extend*, A for aggregation *AggregateJoin*, and As for normalised aggregation *GroupAgg*; languages with the same expressive power are grouped together by {}, while "strictly more expressive" relation is denoted by <.

Out of the above works, the most closely related to ours is arguably the line of research in [27, 28, 46, 47], which studies theoretical properties of bag algebras. In particular, it studies the relative expressive power of bag algebra primitives, the relation to set algebras, with and without aggregation, and establishes a LogSpace data complexity bound for query answering. These results are complementary to the ones in our article, and we expect that many of them can be transferred to the setting of SPARQL.

In contrast to the relational case, query nesting and aggregation are relatively new features of SPARQL that were only standardised in its latest version. To the best of our knowledge, the formal properties of query nesting have only been discussed in [4] and more recently in [5]; a detailed discussion of these works has been provided in previous sections. We are not aware of any articles discussing formal aspects of aggregation in SPARQL 1.1; to the best of our knowledge, the efficient implementation and optimisation of SPARQL 1.1 aggregate queries also constitutes a new and open research area [32].

Analytic and OLAP queries in the context of Semantic Web applications have also been considered only recently. Colazzo et al. [22] proposed a semantics for analytical queries and OLAP operators in the context of RDF; their work, however, does not provide a clear connection with the semantics of aggregation in SPARQL 1.1 and does not study the expressive power of the introduced operators. The RDF Cube specification [23] provides a vocabulary and specific guidelines for representing multidimensional data in RDF; practical aspects and first concrete implementations of the standard have been recently discussed in [33, 35, 55].

This article extends a conference publication [34] by providing extended discussions and examples as well as the following major technical contributions: (*i*) complete proofs of all our results; (*ii*) a principled fix to the normative semantics of exists expressions, and a comprehensive analysis of the impact of this new semantics on the expressive power of query nesting and aggregation constructs; (*iii*) treatment of property paths; and (*iv*) extended treatment of OLAP queries and multidimensional data representation standards for RDF.

11 CONCLUSION AND FUTURE WORK

We have presented an in-depth analysis of the SPARQL 1.1 algebra. Our investigation has shed light on the complex inter-dependencies between the algebraic operators that enable query nesting, variable assignment, and aggregation, which are critical to many emerging applications of semantic technologies. Our main results on relative expressivity of the fragments of SPARQL 1.1 algebra are summarised in Figure 3.

We see many possible avenues for future work. We are planning to study the interaction between aggregation and query nesting operators with other features of SPARQL 1.1, such as entailment regimes and query federation. Furthermore, there have been proposals for an extension of SPARQL with stream reasoning and event processing features [6, 11, 15], as well as with analytical queries [22]; it would be interesting to study the connections between these languages and the SPARQL 1.1 normative algebra.

ELECTRONIC APPENDIX

The electronic Appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

[1] Alberto Abelló, Oscar Romero, Torben Bach Pedersen, Rafael Berlanga Llavori, Victoria Nebot, María José Aramburu Cabo, and Alkis Simitsis. 2015. Using Semantic Web technologies for exploratory OLAP: A survey. *IEEE TKDE* 27, 2 (2015), 571–588.

[2] Shqiponja Ahmetaj, Wolfgang Fischl, Reinhard Pichler, Mantas Simkus, and Sebastian Skritek. 2015. Towards reconciling SPARQL and certain answers. In *WWW 2015*. ACM, 23–33.

[3] Renzo Angles and Claudio Gutierrez. 2008. The expressive power of SPARQL. In *ISWC 2008 (LNCS)*, Vol. 5318. Springer, 114–129.

[4] Renzo Angles and Claudio Gutierrez. 2011. Subqueries in SPARQL. In *AMW 2011 (CEUR Workshop Proceedings)*, Vol. 749. CEUR-WS.org.

[5] Renzo Angles and Claudio Gutierrez. 2016. Negation in SPARQL. *CoRR* abs/1603.06053 (2016).

[6] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: A unified language for event processing and stream reasoning. In *WWW 2011*. ACM, 635–644.

[7] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW 2012*. ACM, 629–638.

[8] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2014. Expressive languages for querying the Semantic Web. In *PODS 2014*. ACM, 14–26.

[9] Marcelo Arenas and Jorge Pérez. 2011. Querying Semantic Web data with SPARQL. In *PODS 2011*. ACM, 305–316.

[10] Elham Akbari Azirani, François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. 2015. Efficient OLAP operations for RDF analytics. In *ICDE Workshops 2015*. IEEE Computer Society, 71–76.

[11] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2010. C-SPARQL: A continuous query language for RDF data streams. *Int. J. Semantic Comput.* 4, 1 (2010), 3–25.

[12] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked data—The story so far. *Int. J. Semantic Web Inf. Syst.* 5, 3 (2009), 1–22.

[13] Carlos Buil Aranda, Marcelo Arenas, Óscar Corcho, and Axel Polleres. 2013. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.* 18, 1 (2013), 1–17.

[14] Carlos Buil Aranda, Axel Polleres, and Jürgen Umbrich. 2014. Strategies for executing federated queries in SPARQL1.1. In *ISWC 2014, Part II (LNCS)*, Vol. 8797. Springer, 390–405.

[15] Jean-Paul Calbimonte, Hoyoung Jeung, Óscar Corcho, and Karl Aberer. 2012. Enabling query technologies for the Semantic Sensor Web. *Int. J. Semantic Web Inf. Syst.* 8, 1 (2012), 43–63.

[16] Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. 2012. Optimization of analytic window functions. *PVLDB* 5, 11 (2012), 1244–1255.

[17] Sara Cohen. 2005. Containment of aggregate queries. *SIGMOD Record* 34, 1 (2005), 77–85.

[18] Sara Cohen. 2006. Equivalence of queries combining set and bag-set semantics. In *PODS 2006*. ACM, 70–79.

[19] Sara Cohen. 2009. Equivalence of queries that are sensitive to multiplicities. *VLDB J.* 18, 3 (2009), 765–785.

[20] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2006. Rewriting queries with arbitrary aggregation functions using views. *ACM TODS* 31, 2 (2006), 672–715.

[21] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2007. Deciding equivalences among conjunctive aggregate queries. *J. ACM* 54, 2, Article 5 (2007), 50 pages.

[22] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatis. 2014. RDF analytics: Lenses over semantic graphs. In *WWW 2014*. ACM, 467–478.

[23] Richard Cyganiak and Dave Reynolds. 2014. *The RDF Data Cube Vocabulary*. W3C Recommendation. W3C.

[24] Lorena Etcheverry and Alejandro A. Vaisman. 2012. Enhancing OLAP analysis with web cubes. In *ESWC 2012 (LNCS)*, Vol. 7295. Springer, 469–483.

[25] Hector García-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database Systems: The Complete Book* (2nd ed.). Pearson Education.
[26] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.* 1, 1 (1997), 29–53.
[27] Stéphane Grumbach, Leonid Libkin, Tova Milo, and Limsoon Wong. 1996. Query languages for bags: Expressive power and complexity. *SIGACT News* 27, 2 (1996), 30–44.
[28] Stéphane Grumbach and Tova Milo. 1996. Towards tractable algebras for bags. *J. Comput. Syst. Sci.* 52, 3 (1996), 570–588.
[29] Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language*. W3C Recommendation. W3C.
[30] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. 2001. Logics with aggregate operators. *J. ACM* 48, 4 (2001), 880–907.
[31] Ivan Herman. 2016. *Errata in SPARQL 1.1*. v1.44. W3C. *Retrieved from http://www.w3.org/2013/sparql-errata*.
[32] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2015. Processing aggregate queries in a federation of SPARQL endpoints. In *ESWC 2015 (LNCS)*, Vol. 9088. Springer, 269–285.
[33] Evangelos Kalampokis, Andriy Nikolov, Peter Haase, Richard Cyganiak, Arkadiusz Stasiewicz, Areti Karamanou, Maria Zotou, Dimitris Zeginis, Efthimios Tambouris, and Konstantinos A. Tarabanis. 2014. Exploiting linked data cubes with OpenCube toolkit. In *ISWC 2014 (Posters & Demonstrations) (CEUR Workshop Proceedings)*, Vol. 1272. CEUR-WS.org, 137–140.
[34] Mark Kaminski, Egor V. Kostylev, and Bernardo Cuenca Grau. 2016. Semantics and expressive power of subqueries and aggregates in SPARQL 1.1. In *WWW 2016*. ACM, 227–238.
[35] Yasar Khan, Muhammad Saleem, Aftab Iqbal, Muntazir Mehdi, Aidan Hogan, Axel-Cyrille Ngonga Ngomo, Stefan Decker, and Ratnesh Sahay. 2014. SAFE: Policy aware SPARQL query federation over RDF data cubes. In *SWAT4LS 2014 (CEUR Workshop Proceedings)*, Vol. 1320. CEUR-WS.org.
[36] Anthony C. Klug. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM* 29, 3 (1982), 699–717.
[37] Roman Kontchakov and Egor V. Kostylev. 2016. On expressibility of non-monotone operators in SPARQL. In *KR 2016*. AAAI Press, 369–379.
[38] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev. 2014. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *ISWC 2014, Part I (LNCS)*, Vol. 8796. Springer, 552–567.
[39] Egor V. Kostylev and Bernardo Cuenca Grau. 2014. On the semantics of SPARQL queries with optional matching under entailment regimes. In *ISWC 2014, Part II (LNCS)*, Vol. 8797. Springer, 374–389.
[40] Egor V. Kostylev, Juan L. Reutter, Miguel Angel Romero Orth, and Domagoj Vrgoc. 2015. SPARQL with property paths. In *ISWC 2015, Part I (LNCS)*, Vol. 9366. Springer, 3–18.
[41] Egor V. Kostylev, Juan L. Reutter, and Martín Ugarte. 2015. CONSTRUCT queries in SPARQL. In *ICDT 2015 (LIPIcs)*, Vol. 31. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 212–229.
[42] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient processing of window functions in analytical SQL queries. *PVLDB* 8, 10 (2015), 1058–1069.
[43] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. 2013. Static analysis and optimization of Semantic Web queries. *ACM TODS* 38, 4, Article 25 (2013), 45 pages.
[44] Leonid Libkin. 2000. Logics with counting and local properties. *ACM TOCL* 1, 1 (2000), 33–59.
[45] Leonid Libkin. 2003. Expressive power of SQL. *Theor. Comput. Sci.* 296, 3 (2003), 379–404.
[46] Leonid Libkin and Limsoon Wong. 1994. New techniques for studying set languages, bag languages and aggregate functions. In *PODS 1994*. ACM, 155–166.
[47] Leonid Libkin and Limsoon Wong. 1997. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.* 55, 2 (1997), 241–272.
[48] Katja Losemann and Wim Martens. 2012. The complexity of evaluating path expressions in SPARQL. In *PODS 2012*. ACM, 101–112.
[49] Frank Manola and Eric Miller. 2004. *RDF Primer*. W3C Recommendation. W3C.
[50] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM TODS* 34, 3, Article 16 (2009), 45 pages.
[51] Axel Polleres and Johannes Peter Wallner. 2013. On the relation between SPARQL1.1 and answer set programming. *J. Appl. Non-Class. Log.* 23, 1–2 (2013), 159–212.
[52] Eric Prud'hommeaux and Andy Seaborne. 2008. *SPARQL Query Language for RDF*. W3C Recommendation. W3C.
[53] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In *ICDT 2010*. ACM, 4–33.

[54] Nicole Schweikardt. 2005. Arithmetic, first-order logic, and counting quantifiers. *ACM TOCL* 6, 3 (2005), 634–671.
[55] Efthimios Tambouris, Evangelos Kalampokis, and Konstantinos A. Tarabanis. 2015. Processing linked open data cubes. In *EGOV 2015 (LNCS)*, Vol. 9248. Springer, 130–143.
[56] Xiaowang Zhang and Jan Van den Bussche. 2014. On the primitivity of operators in SPARQL. *Inf. Process. Lett.* 114, 9 (2014), 480–485.
[57] Xiaowang Zhang and Jan Van den Bussche. 2015. On the power of SPARQL in expressing navigational queries. *Comput. J.* 58, 11 (2015), 2841–2851.
[58] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. 2011. Graph cube: On warehousing and OLAP multidimensional networks. In *SIGMOD 2011*. ACM, 853–864.

Received July 2016; revised February 2017; accepted April 2017