Semantics and Expressive Power of Subqueries and Aggregates in SPARQL 1.1

Mark Kaminski
kaminski@cs.ox.ac.uk

Egor V. Kostylev
ekostyle@cs.ox.ac.uk

Bernardo Cuenca Grau
berg@cs.ox.ac.uk

Department of Computer Science, University of Oxford, UK

ABSTRACT

Answering aggregate queries is a key requirement of emerging applications of Semantic Technologies, such as data warehousing, business intelligence and sensor networks. In order to fulfill the requirements of such applications, the standardisation of SPARQL 1.1 led to the introduction of a wide range of constructs that enable value computation, aggregation, and query nesting. In this paper 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.

1. INTRODUCTION

An increasing number of RDF-based applications require support for *aggregate queries*, which, rather than simply retrieving data, involve some form of computation or summarisation. Answering aggregate queries is a key requirement in data warehousing and business intelligence, where data is aggregated across many dimensions looking for patterns [1, 9, 19–21, 25, 42], as well as in emerging applications involving sensor networks and streaming RDF data [5, 10, 13].

The first version of SPARQL [36], however, did not provide support for aggregation, which limited its applicability in such applications. The standardisation of SPARQL 1.1 [23] addressed these limitations by introducing a wide range of constructs in line with those 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.

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 intensive research and has laid the foundations for subsequent implementations [3, 29, 30, 34, 35, 38, 40]. Similarly to its predecessor, the semantics of SPARQL 1.1 is specified by means of an (extended) normative algebra and many of the new features such as property paths [6, 28, 33, 41], query federation [11, 12], or entailment regimes [2, 7, 26, 27] 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. This is in stark contrast to the case of relational databases, where the formal properties of arithmetic and aggregation have been studied in depth [14–18, 24, 31, 32, 39].

Our aim is to provide a systematic study of the semantics and expressive power of the SPARQL 1.1 algebra with aggregates and 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.

In our investigation we take the well-known SPARQL algebra as a starting point, which we recapitulate in Section 2. Most existing works on SPARQL assume that graph patterns are interpreted as sets of solution mappings rather than multisets (or bags) as in the normative specification. This simplifying assumption is, however, no longer reasonable once aggregation comes into play and hence we adopt multiset semantics from the word go in this paper.

In Section 3 we study the query nesting mechanisms available in SPARQL 1.1. We first consider in Section 3.1 the nesting of `SELECT` and `SELECT DISTINCT` query blocks. In algebraic terms, this amounts to allowing the unrestricted use of the operators *Project* and *Distinct* rather than restricting them to the outermost level of queries as in SPARQL. We show that there is no gain of expressive power by allowing the unrestricted use of just one of these operators. In contrast, if *both* operators are allowed unrestrictedly, we show how to construct queries that cannot be equivalently expressed in SPARQL. 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 [15], and was later conjectured by Angles and Gutierrez to also yield additional expressive power in SPARQL [4]. As argued in Section 3.1, however, the evidence given in [4] in support of their conjecture is unsatisfactory. Our results

WWW 2016, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4143-1/16/04.
http://dx.doi.org/10.1145/2872427.2883022.

settle this question and provide a detailed account of which combinations of constructs lead to expressivity gains and which ones are redundant. Besides subqueries as patterns, SPARQL 1.1 also provides a mechanism where graph patterns can be embedded within expressions in lter conditions. We investigate this form of query nesting in Section 3.2 and show that it can be simulated within SPARQL, thus not resulting in additional expressive power.

In Section 4 we turn our attention to variable assignment and aggregation. The former is enabled in the SPARQL 1.1 algebra by the *Extend* operator, which extends solution mappings with a fresh variable assigned to the value of an expression. In Section 4.1 we show that *Extend* can be simulated

valued. This is in contrast to the general case, where

- $\{\!\!\{ \mu' \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, Cond \}\!\!\}$ for Ω_1, Ω_2 and $Cond$ is the multiset with base set consisting of all μ' such that *Cond* holds for μ' and some μ_1 in Ω_1 and μ_2 in Ω_2, and where the multiplicity is defined as the following sum ranging over the pairs of contributing μ_1, μ_2:

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

The semantics of patterns and queries over a graph G is defined as follows, where $\mu(P)$ is the pattern obtained from P by replacing its variables according to μ:

- $[\![t]\!]_G$ for a triple pattern t is the multiset with $S_{[\![t]\!]_G}$ consisting of all μ such that $\mathsf{dom}(\mu) = \mathsf{var}(t)$ and $\mu(t)$ belongs to G, and $\mathsf{card}_{[\![t]\!]_G}(\mu) = 1$ for each such μ;
- $[\![\mathit{Join}(P_1, P_2)]\!]_G =$
  $\{\!\!\{ \mu \mid \mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_G, \mu = \mu_1 \cup \mu_2 \}\!\!\}$;
- $[\![\mathit{Union}(P_1, P_2)]\!]_G = [\![P_1]\!]_G \uplus [\![P_2]\!]_G$;
- $[\![\mathit{Filter}(E, P_1)]\!]_G = \{\!\!\{ \mu \mid \mu \in [\![P_1]\!]_G, [\![E]\!]_{\mu,G} = \mathit{true} \}\!\!\}$; and
- $[\![\mathit{LeftJoin}(E, P_1, P_2)]\!]_G =$
  $\{\!\!\{ \mu \mid \mu \in [\![\mathit{Join}(P_1, P_2)]\!]_G, [\![E]\!]_{\mu,G} = \mathit{true} \}\!\!\} \uplus$
  $\{\!\!\{ \mu \mid \mu \in [\![P_1]\!]_G,$
  $\forall \mu_2 \in [\![P_2]\!]_G . (\mu \not\sim \mu_2 \text{ or } [\![E]\!]_{\mu \cup \mu_2, G} = \mathit{false}) \}\!\!\}$.

We conclude with the semantics of Sparql queries, which are also evaluated as multisets in our formalisation:

- $[\![\mathit{Project}(X, P)]\!]_G$ is such that its base set consists of the restrictions μ' to X of all μ in $[\![P]\!]_G$, and the multiplicity of μ' is the sum of multiplicities of all corresponding μ;
- $[\![\mathit{Distinct}(Q)]\!]_G$ is the multiset with the same base set as $[\![Q]\!]_G$, but with multiplicity 1 for all mappings.

Expressive Power of Query Languages We consider extensions of Sparql with various constructs. This may lead to an increase in expressive power, that is, some queries in the extended language may not be equivalently rewritable to the original language. We next make this notion precise.

A *query language for RDF* is a pair $(\mathsf{Q}, [\![.]\!]_.)$ where Q is the class of *queries* and $[\![.]\!]_.$ is the *evaluation function* that maps queries and RDF graphs to multisets of solution mappings (e.g., algebra Sparql is a query language for RDF). A language $\mathcal{L}_2 = (\mathsf{Q}_2, [\![.]\!]^2_.)$ *extends* a language $\mathcal{L}_1 = (\mathsf{Q}_1, [\![.]\!]^1_.)$ if $\mathsf{Q}_1 \subseteq \mathsf{Q}_2$ and the restriction of $[\![.]\!]^2_.$ to Q_1 coincides with $[\![.]\!]^1_.$. A query Q in a language $\mathcal{L} = (\mathsf{Q}, [\![.]\!]_.)$ is *equivalent* to Q' in $\mathcal{L}' = (\mathsf{Q}', [\![.]\!]'_.)$ 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. We say that $\mathcal{L}_1$ and $\mathcal{L}_2$ have the *same expressive power* if each of them is more expressive than the other one. Finally, $\mathcal{L}_1$ is *strictly more expressive* than $\mathcal{L}_2$ if it is more expressive, but does not have the same expressive power.

3. NESTED QUERIES

In this section we investigate the expressive power provided by nested queries; that is, those having another query (a *subquery*) embedded within. The subquery can itself be a nested query; thus, queries can have a deep nested structure.

SPARQL 1.1 allows for two kinds of nesting. First, subqueries can play the role of patterns within the `WHERE` clause of another query. In algebraic terms, this is tantamount to allowing the arbitrary use of the algebraic 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. We investigate this basic form of nested queries in Section 3.1. Second, graph patterns can be embedded within expressions in filter conditions by means of the exists construct. We investigate this form of nesting in Section 3.2.

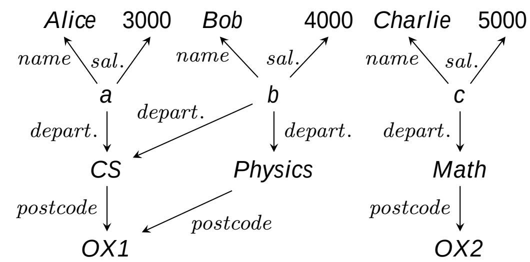

Figure 1: Example RDF graph G_{ex}

Before moving into further particulars, we first show that Sparql can express a "set difference" operator, which we will exploit throughout this section to encode other constructs.[1]

Definition 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:*

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

In contrast to the relational multiset difference operator, where the occurrences of μ in $[\![P_1]\!]_G$ are subtracted from those in $[\![P_2]\!]_G$, this operator yields μ, with the same cardinality as in $[\![P_1]\!]_G$, if $\mu \notin [\![P_2]\!]_G$. Thus, μ is not returned whenever $\mu \in [\![P_2]\!]_G$, regardless of its cardinality in $[\![P_1]\!]_G$.

Proposition 1. *For any extension* Sparql_X *of* Sparql *the pattern SetMinus(P_1, P_2) with* Sparql_X *patterns P_1 and P_2 is expressible in* Sparql_X.

Proof Sketch. We need to establish a mechanism to distinguish between the mappings in $[\![P_1]\!]_G$ that occur in $[\![P_2]\!]_G$ from those mappings that do not. The idea is to first construct a pattern P_3 whose evaluation captures the latter together with all mappings in $[\![P_2]\!]_G$ extended with fresh variables $?x$, $?y$, and $?z$. We do this as follows, where $X = \mathsf{var}(P_1) \cup \mathsf{var}(P_2)$ and θ is a renaming of X:

$$P_2' = \mathit{Join}(P_2, (?x, ?y, ?z));\ P_3 = \mathit{LeftJoin}(\mathsf{eq}(X, \theta), P_1, P_2'\theta).$$

Now, we define $SetMinus(P_1, P_2) = \mathit{Filter}(\neg \mathsf{bound}(?x), P_3)$, where the filter condition eliminates all mappings with fresh variables, thus producing the required result. □

3.1 Subqueries as Patterns

We start with a discussion of the basic nesting mechanism in SPARQL 1.1, which allows queries to be subpatterns of other patterns. Consider the query (Q1) given next.

(Q1) *Find the names of people and the postcodes of the departments where they work.*

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

Let us evaluate (Q1) over the RDF graph G_{ex} depicted in Figure 1, which we will use as a running example. Variable $?d$ for departments is only visible within the subquery, whereas $?x$ and $?p$ are projected and hence visible in the

[1] Note that our operator is different from the `MINUS` operator in SPARQL or the difference operator discussed in [3,8].

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 $\mu_a = \{?n \mapsto \textit{Alice}, ?p \mapsto \textit{OX1}\}$, $\mu_b = \{?n \mapsto \textit{Bob}, ?p \mapsto \textit{OX1}\}$, and $\mu_c = \{?n \mapsto \textit{Charlie}, ?p \mapsto \textit{OX2}\}$, 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 mainframe language Sparql by allowing *Project* and *Distinct* in patterns. After this extension, there is no longer a meaningful distinction between patterns and queries in the language.

DEFINITION 2. *The language* Sparql$_{PD}$ *extends* Sparql *by allowing the query constructs Project*(X, P) *and Distinct*(P) *as patterns, called* subquery patterns*. The intermediate languages* Sparql$_P$ *and* Sparql$_D$ *allow only for Project and only for Distinct in patterns, respectively.*

The language Sparql$_{PD}$ captures the query nesting functionality in SPARQL 1.1. Specifically, (Q1) is as follows:

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

At first sight, query nesting provides a great deal of power and flexibility to the language. As we have seen, it can lead to sophisticated interactions between set and bag semantics, which may be difficult (or 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.

We now show that every Sparql$_{PD}$ query can be brought into a normal form in which query nesting is bounded by depth two; that is, there exists a natural bound on the level of nesting after which no further increase in expressive power can be achieved. This normal form is defined as given next, and one can check that query (Q1) satisfies its requirements.

DEFINITION 3. *A* Sparql$_{PD}$ *query* Q *is in* s-normal form *if either* $Q = \textit{Distinct}(\textit{Project}(X, P))$ *with* P *a subquery-free pattern, or* $Q = \textit{Project}(X, P)$ *where all subquery patterns in* P *are of the form Distinct*$(\textit{Project}(X', P'))$ *with* $X' \subsetneq \mathsf{var}(P')$ *and* P' *subquery-free.*

This normal form not only limits nesting depth, but also restricts the ways in which *Project* and *Distinct* can be combined. In particular, if Q is in Sparql$_P$ (or in Sparql$_D$) and hence *Distinct* (respectively, *Project*) only occurs in the outermost level, then Definition 3 requires that P is subquery-free and hence Q is a Sparql query. We show that each Sparql$_{PD}$ query can be brought into s-normal form.

THEOREM 1. *Let* X *be one of* P, D, PD*. Every query in* Sparql$_X$ *has an equivalent* Sparql$_X$ *query in s-normal form.*

PROOF SKETCH. The first relevant observation is that all occurrences of *Distinct* in a Sparql$_X$ query Q that are in scope of other *Distinct* subpatterns can be removed upfront without affecting the semantics. Indeed, if Q has a subpattern *Distinct*(P) where P has in turn a subpattern *Distinct*(P'), we can simply replace *Distinct*(P') with P'. Second, *Project* can be \pushed" upwards through all operators except *Distinct*. For instance, *Join*$(\textit{Project}(X, P_1), P_2)$ can be rewritten as *Project*$(X \cup \mathsf{var}(P_2), \textit{Join}(P_1\theta, P_2))$ with θ a renaming of $\mathsf{var}(P_1) \setminus X$. Moreover, subsequent occurrences of *Project* can be merged since *Project*$(X_1 \cap X_2, P)$ and *Project*$(X_1, \textit{Project}(X_2, P))$ are equivalent.

To complete the proof, it suffices to show that *Distinct* can be eliminated in every subpattern of Q of the form *Distinct*(P) with P subquery-free. For this, we bring P into the normal form of [34, Proposition 3.8] where *Union* patterns are arguments only of other *Union* operators, and then observe that *Distinct*$(\textit{Union}(P_1, P_2))$ can be written as

$$\textit{Project}(X, \textit{Union}(\textit{Union}(\textit{SetMinus}(\textit{Distinct}(P_1), P_2), \\ \textit{SetMinus}(\textit{Distinct}(P_2), P_1)), \\ \textit{Filter}(\mathsf{eq}(X, \theta), \textit{Distinct}(\textit{Join}(P_1, P_2\theta))))),$$

where $X = \mathsf{var}(P_1) \cup \mathsf{var}(P_2)$ and θ is a renaming of X. Although this rewriting introduces an additional occurrence of *Project*, this occurrence can be pushed up to the outermost level (since, by construction, there is no *Distinct* between the occurrence of *Project* and the outermost level of Q).

When all occurrences of *Project* and *Union* are above *Distinct* we can simply erase all *Distinct* operators; indeed, this does not change the semantics because in the absence of *Project* or *Union*, mappings in the evaluation of a pattern are pairwise incompatible (see [34] for details), and hence *Join*, *LeftJoin*, and *Filter* cannot produce duplicates. □

COROLLARY 1. *The languages* Sparql$_P$, Sparql$_D$ *and* Sparql *have the same expressive power.*

Under set semantics, *Distinct* is redundant so we can conclude that in this setting subqueries as patterns do not add expressivity to Sparql. Thus, the obvious question is whether subqueries in patterns can be completely eliminated under bag semantics. Cohen [15] observed that query nesting in SQL can cause a complex interplay between bag and set semantics, which was then used by Angles and Gutierrez [4] to conjecture that query nesting adds expressive power to SPARQL. The claim in [4], however, comes without a proof, and the example given of an inexpressible nested query can in fact be rewritten in Sparql. A closer look at Cohen's techniques also reveals that they cannot be used for showing inexpressibility. We now settle this question and show that there exist Sparql$_{PD}$ queries that cannot be expressed in Sparql (and by Corollary 1 also in Sparql$_P$ or Sparql$_D$); that is, query nesting cannot be fully eliminated.

THEOREM 2. *The language* Sparql$_{PD}$ *is strictly more expressive than* Sparql*.*

PROOF SKETCH. We claim that the following query Q in s-normal form is not expressible in Sparql:

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

Indeed, consider graphs $G_{m,n}$, $m, n \geq 1$, of the form

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

Query Q evaluates on $G_{m,n}$ to the multiset with m copies of the mapping $\mu = \{?x \mapsto a, ?y \mapsto c\}$. In contrast, every Sparql query equivalent to Q modulo multiplicities evaluates on $G_{m,n}$ to a multiset $'$ such that either $\mathsf{card}_{'}(\mu) = 1$ or $\mathsf{card}_{'}(\mu) = m \cdot n$. Intuitively, this is so because a Sparql query cannot distinguish between the different b_i and d_j and hence if a query of the form *Project*(X, P) returns μ once, it must return all its $m \cdot n$ copies. Of course, we can always use *Distinct* in the query's outermost level, but then we obtain a single copy of μ instead of required m copies. □

3.2 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.

Consider the following query (Q2), which evaluates to the single mapping $\{?n \mapsto \mathit{Charlie}\}$ on graph G_{ex} in Figure 1.

(Q2) *Find the names of people not working in CS.*

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

To support queries such as (Q2), we introduce the exists construct in the algebra as defined next.

DEFINITION 4. *Given any extension* Sparql_X *of* Sparql*, the language* $\mathsf{Sparql}_X^\exists$ *further extends* Sparql_X *by permitting expressions of the form* exists(P) *for a pattern* P*. Its semantics is as follows, for a mapping* μ *and graph* G*:*

$$[\![\mathsf{exists}(P)]\!]_{\mu,G} = \begin{cases} \mathit{true}, & \text{if } [\![\mu(P)]\!]_G \text{ is not empty}, \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

Contrary to Sparql expressions, the semantics of exists depends on the relevant RDF graph. Query (Q2) is written in $\mathsf{Sparql}^\exists$ as follows, where the expression in the filter evaluates to *true* only for the mapping $\{?x \mapsto c, ?n \mapsto \mathit{Charlie}\}$:

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

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 does not provide additional expressive power, and can be fully simulated by means of other constructs. We proceed according to the following three steps.

1. We first show that exists can be eliminated from *LeftJoin* patterns. This is intuitively achieved by "pushing" complex expressions from *LeftJoin* into *Filter* patterns.
2. We then show that any *Filter* pattern can be expressed in terms of exists-free *Filter* patterns and patterns of the form *Filter*(exists$(P_1), P_2$) and *Filter*($\neg$exists$(P_1), P_2$).
3. In the last step, we show that all patterns of the form *Filter*(exists$(P_1), P_2$) and *Filter*($\neg$exists$(P_1), P_2$) can be rewritten in terms of exists-free patterns only.

The first step is justified by the following lemma.

LEMMA 1. *Let* $\mathsf{Sparql}_X^\exists$ *be any language extending* Sparql *and allowing for* exists *expressions. For every* $\mathsf{Sparql}_X^\exists$ *pattern there exists an equivalent* $\mathsf{Sparql}_X^\exists$ *pattern with each LeftJoin subpattern of the form LeftJoin*(eq$(X, \theta), P_1', P_2'$)*.*

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

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

Then, we construct P' such that $[\![P']\!]_G$ captures, with correct multiplicities, all mappings in $[\![P_1]\!]_G$ and not in $[\![P]\!]_G$. 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 = *Join*$(P, (?x, ?y, ?z))$. The semantics of *LeftJoin* then ensures that the pattern *LeftJoin*(E, P_1, P_2) is equivalent to *Union*(*Filter*$(E$, *Join*$(P_1, P_2)), P')$. □

In the second step, we show that patterns *Filter*(E, P_1) where exists may occur arbitrarily (and more than once) in E can be reduced to patterns *Filter*(E', P_2) where E' is either exists-free or is of the form $(\neg)$exists(P).

LEMMA 2. *Let* $\mathsf{Sparql}_X^\exists$ *be any extension of* Sparql *allowing for* exists*. Each* $\mathsf{Sparql}_X^\exists$ *pattern has an equivalent pattern where each Filter-subpattern involving* exists *is of the form Filter*(exists$(P_2), P_1$) *or Filter*($\neg$exists$(P_2), P_1$)*.*

PROOF. Consider a $\mathsf{Sparql}_X^\exists$ pattern of the form *Filter*(E, P), where an expression exists(P') occurs in E. Since exists(P') must evaluate to either *true* or *false*, we can capture each possibility by replacing 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 exists(P') replaced by E'. □

For the final step, we observe that patterns of the form *Filter*($\neg$exists$(P_2), P_1$) can be directly expressed in Sparql (e.g., see [3, 8]), whereas patterns *Filter*(exists$(P_2), P_1$) can be reduced to the former using the $SetMinus$ operator.

LEMMA 3. *Let* $\mathsf{Sparql}_X^\exists$ *be any language extending* Sparql *and allowing for* exists*, and let* P_1, P_2 *be* $\mathsf{Sparql}_X^\exists$ *patterns. Then, Filter*($\neg$exists$(P_2), P_1$) *and Filter*(exists$(P_2), P_1$) *are expressible in* Sparql_X*.*

PROOF. A pattern of the form *Filter*($\neg$exists$(P_2), P_1$) can be rewritten as follows, with $?x$, $?y$, and $?z$ fresh variables:

$$\mathit{Filter}(\neg\mathsf{bound}(?x), \mathit{LeftJoin}(\mathit{true}, P_1, \mathit{Join}(P_2, (?x, ?y, ?z)))).$$

In turn, a pattern of the form *Filter*(exists$(P_2), P_1$) is equivalent to *SetMinus*$(P_1$, *Filter*($\neg$exists$(P_2), P_1$)). □

The following result then follows from Lemmas 1–3.

THEOREM 3. *Let* $\mathsf{Sparql}_X^\exists$ *be a language extending* Sparql *and allowing for* exists*. Then,* $\mathsf{Sparql}_X^\exists$ *has the same expressive power as* Sparql_X *(i.e., the extension without* exists*).*

4. ASSIGNMENT AND AGGREGATION

In addition to retrieving data, many applications require the ability to perform some form of computation. 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 signi cant extension to the relational algebra, which involves *grouping* and *generalised projection* operators, and *aggregate functions* (e.g., see [22]).

The original SPARQL recommendation, however, did not provide any such features. Although arithmetic expressions were available, computed values could only be used as part of lter conditions; thus, the means for assigning such values to variables and subsequently return them in query answers was missing. Similarly, SPARQL 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 SQL.

Our aim in this section 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 Section 4.1 we study the *Extend* operator, which provides the means for assigning variables to complex expressions. In Section 4.2 we discuss the SPARQL 1.1 normative algebra for aggregation and present its equivalent formalisation that closes unspeci ed corner cases and makes ambiguous aspects of the speci cation precise. Then, we demonstrate the power of the aggregate algebra by showing that it is capable of expressing variable assignment (i.e., *Extend*) as well as nested queries in their full generality. In Section 4.3 we present a normal form for aggregate algebra queries, which leads to a substantial simpli cation of the SPARQL 1.1 aggregate algebra where most of its unconventional aspects have been eliminated.

4.1 Variable Assignment to Expressions

SPARQL 1.1 provides binding constructs `BIND` and `VALUES` and the alias construct `AS` for assigning values of complex expressions to variables. As an example, consider the following query, where variables are assigned to computed values.

(Q3) *Return people's names with their salaries after 20% tax and ags indicating whether they work in the CS department.*

```
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 (Q3) yields mappings such as $\{?n \mapsto \textit{Alice}, ?t \mapsto 2400, ?c \mapsto \textit{true}\}$, 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 as de ned next.

Definition 5. *Given any extension* Sparql_X *of* Sparql, *the language* Sparql_{XE} *further extends* Sparql_X *by permitting patterns of the form* $\textit{Extend}(?x, E, P)$*, where* $?x$ *is a variable not in* $\mathsf{var}(P)$*,* E *is an expression, and* P *is a pattern. For a graph* G*, its semantics is as follows:*

$$\llbracket \textit{Extend}(?x, E, P) \rrbracket_G = \{\!\!\{ \mu' \mid \mu \in \llbracket P \rrbracket_G, \mu' = \mu \cup \{?x \mapsto \llbracket E \rrbracket_{\mu,G}\}, \llbracket E \rrbracket_{\mu,G} \neq \textit{error} \}\!\!\} \uplus \{\!\!\{ \mu \mid \mu \in \llbracket P \rrbracket_G, \llbracket E \rrbracket_{\mu,G} = \textit{error} \}\!\!\}.$$

We also set $\mathsf{var}(\textit{Extend}(?x, E, P)) = \{?x\} \cup \mathsf{var}(P)$.

For instance, (Q3) is translated into the algebra as follows:

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

Unsurprisingly, adding *Extend* to the language increases its expressive power, since it provides means for queries to return values that do not occur in the queried graph.

Proposition 2. *The language* Sparql_{PDE} *is strictly more expressive than* Sparql_{PD}.

Proof. Let the query Q be de ned as follows:

$$\textit{Project}(\{?x\}, \textit{Extend}(?x, \mathsf{bound}(?y), (?y, a, a))).$$

$\llbracket Q \rrbracket_G$ contains the mapping $\{?x \mapsto \textit{true}\}$ for $G = \{(a, a, a)\}$. However, any Sparql_{PD} query Q' has $\mu(?z) = a$ for each $\mu \in \llbracket Q' \rrbracket_G$ and $?z \in \mathsf{dom}(\mu')$, so it is not equivalent to Q. □

The standard notion of expressive power, however, is not well-suited for dealing with constructs such as *Extend*. Assume a very restricted use of *Extend* where only a Boolean expression that always evaluates to *false* is allowed; although a query could still introduce a fresh value not occurring in the graph, this is done in a trivial way and hence one could argue that there is no actual gain in expressive power in this case. We next introduce a more liberal notion of expressive power derived from [3,37]. This notion is based on a generalisation of query equivalence which allows for changes in the input graph; these changes are, however, far from arbitrary and need to be uniform across all queries.

Definition 6. *Let* $\mathcal{L}_1 = (\mathsf{Q}_1, \llbracket . \rrbracket^1_\cdot)$ *and* $\mathcal{L}_2 = (\mathsf{Q}_2, \llbracket . \rrbracket^2_\cdot)$ *be languages,* I' *be a nite subset of the set* I *of IRIs, and* $\mathsf{T}' = \mathsf{I}' \cup \mathsf{B} \cup \mathsf{L}$*. Let* f *be a (computable) function from graphs over* T' *to general graphs over* T *such that* $f(G) = G \cup G'$ *for any* G*, where* G' *is a graph not using IRIs in* I'*. A query* $Q_1 \in \mathsf{Q}_1$ *over* T' *is* f-expressible *by a query* $Q_2 \in \mathsf{Q}_2$ *if* $\llbracket Q_1 \rrbracket^1_G = \llbracket Q_2 \rrbracket^2_{f(G)}$ *for every graph* G *over* T'*. Language* $\mathcal{L}_2$ *is* weakly more expressive *than* $\mathcal{L}_1$ *if for every nite set of IRIs* I' *there exists some* f *such that each query in* $\mathcal{L}_1$ *over* T' *is* f*-expressible by a query in* $\mathcal{L}_2$*. The associated strict notion and the notion of equivalence in expressive power are de ned in the obvious way.*

We next show a surprising result: Sparql_{PDE} and Sparql_{PD} are equivalent under this generalised notion of expressive power, provided that expressions in *Extend* patterns are restricted to be Boolean-valued. This implies that constructs such as `BIND` in queries such as (Q3) can be captured by query-independent functions for transforming the input graphs. Intuitively, if the values assigned to variables in *Extend*-patterns range over a nite domain D, applications of *Extend* can be simulated using *Filter* and *Union* when evaluated over a graph extended by an enumeration of D.

Theorem 4. *Let* $\mathsf{Sparql}^{bool}_{PDE}$ *extend* Sparql_{PD} *by allowing patterns* $\textit{Extend}(?x, E, P)$ *with expression* E *evaluating only to Boolean values. Then,* $\mathsf{Sparql}^{bool}_{PDE}$ *and* Sparql_{PD} *are weakly equivalent in expressive power.*

Proof. We show that Sparql_{PD} is weakly more expressive than $\mathsf{Sparql}^{bool}_{PDE}$, the other direction is straightforward. Given a nite I', let $u_t, u_f, u \in \mathsf{I} \setminus \mathsf{I}'$ and consider f such that $f(G) = G \cup \{(u_t, u, \textit{true}), (u_f, u, \textit{false})\}$ for any G. Then every $\mathsf{Sparql}^{bool}_{PDE}$ query Q_1 over T' is f-expressible by a Sparql_{PD} query constructed by the following two steps:

1. replace each (s, p, o) in Q_1 by *Filter*$(\neg(p \doteq u), (s, p, o))$;
2. replace each subpattern *Extend*$(?x, E, P)$ by

$$\mathit{Union}(\mathit{Union}(\mathit{Join}((u_t, u, ?x), \mathit{Filter}(E, P)), \mathit{Join}((u_f, u, ?x), \mathit{Filter}(\neg E, P))), \mathit{SetMinus}(P, \mathit{Filter}(E \vee \neg E, P))).$$

Note that the *SetMinus* subpattern corresponds to mappings for which E evaluates to *error*. □

If we consider general expressions, however, *Extend* introduces arbitrary arithmetic in the language | something that cannot be simulated by query-independent transformations.

THEOREM 5. *Language* Sparql_{PDE} *is strictly weakly more expressive than* Sparql_{PD}.

Similarly, Sparql_{PD} remains strictly more expressive than Sparql even under the generalised notion of expressive power, which can be proved similarly to Theorem 2.

4.2 The Aggregate Algebra

SPARQL 1.1 and SQL provide similar functionality for aggregation: grouping is used to de ne equivalence classes of solution mappings over which aggregate functions are subsequently applied. Consider the following example query.

(Q4) *Return the total employee salary per department, but considering only departments having at least two employees.*

```
SELECT ?d (SUM(?s) AS ?n)
  WHERE {?x department ?d . ?x salary ?s}
GROUP BY ?d
HAVING COUNT(?x) > 1
```

Over G_{ex}, (Q4) evaluates to $\{?d \mapsto CS, ?n \mapsto 7000\}$, since the CS department is the only one with several employees and the total salary of Bob and $Alice$ is 7000.

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

(F1) groups and aggregates are seen as rst-class citizens of the algebra, which are de ned independently using dedicated constructs *Group* and *Aggregate*;
(F2) grouping is allowed on arbitrary lists of expressions, and not just on lists of variables; and
(F3) aggregation is also allowed on arbitrary lists of expressions, and not just on single expressions.

Both groups and aggregates deal with lists of expressions, which evaluate to *v-lists*: lists of values in $\mathsf{T} \cup \{\mathit{error}\}$. In particular, $[\![\mathsf{E}]\!]_{\mu,G} = [[\![E_1]\!]_{\mu,G}, \ldots, [\![E_k]\!]_{\mu,G}]$ for a list of expressions $\mathsf{E} = [E_1, \ldots, E_k]$.

We start our discussion by introducing groups as rst-class citizens of the algebra. Roughly speaking, 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 7. *A* group *is a construct* *Group*(E, P) *with* E *a list of expressions and* P *a pattern. The* evaluation $[\![\ ]\!]_G$ *of over a graph* G *is a partial function from v-lists to multisets of mappings that is de ned for all v-lists Key* $= [\![\mathsf{E}]\!]_{\mu,G}$ *with* $\mu \in [\![P]\!]_G$ *as follows:*

$$[\![\ ]\!]_G(\mathit{Key}) = \{\!\!\{ \mu' \mid \mu' \in [\![P]\!]_G, [\![\mathsf{E}]\!]_{\mu',G} = \mathit{Key} \}\!\!\}.$$

As in SQL, *aggregate functions* in SPARQL 1.1 (e.g., `SUM` in query (Q4)) 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 [14]. 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 a function Flatten, which maps each multiset of v-lists to the multiset of values in $\mathsf{T} \cup \{\mathit{error}\}$ having as base the values in and having card $(v) = \sum_{\lambda \in} (\mathsf{card}\ (\lambda) \times n_{v,\lambda})$ for each such value v, where $n_{v,\lambda}$ is the number of appearances of v in λ.

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

DEFINITION 8. *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* $b \in \mathsf{B}$, $u \in \mathsf{I}$, $\ell \in \mathsf{L}$. *A SPARQL 1.1* aggregate function *is one of the following functions, mapping multisets of v-lists to values in* $\mathsf{T} \cup \{\mathit{error}\}$, *where* = Flatten( ):

{ Count( ) = $\sum_{v \in\ , v \neq \mathit{error}} \mathsf{card}\ (v)$;
{ Sum( ) *is* $\sum_{v \in} (\mathsf{card}\ (v) \times v)$ *if all the values in are numbers, and error otherwise;*
{ Avg( ) *is* 0 *if* Count( ) = 0 *and* Sum( ) / Count( ) *otherwise (in particular, it is error if* Sum( ) = *error);*
{ Min( ) *is* $\prec$*-min value in if* $\neq \emptyset$ *and error otherwise;*
{ Max( ) *is* $\prec$*-max value in if* $\neq \emptyset$ *and error otherwise;*
{ Sample( ) *is some value in if* $\neq \emptyset$ *and error otherwise.*

Finally, CountD, SumD, *and* AvgD *are de ned as their counterparts* Count, Sum, *and* Avg, *but applied to the multiset of v-lists obtained from by removing duplicates.*

Note that *error* does not contribute to Count, but may a ect the results of other functions. Note also that Sample is non-deterministic. We use Id as its synonym whenever, by construction, Flatten( ) consists of a single value (with any cardinality); thus, Id is deterministic.

We now de ne the aggregate construct, which computes a single value for each group by means of aggregate functions.

DEFINITION 9. *An* aggregate A *is a construct of the form* *Aggregate*$(\mathsf{F}, f, \ )$, *for* F *a list of expressions,* f *an aggregate function, and* = *Group*(E, P) *a group. The* evaluation $[\![A]\!]_G$ *of* A *over a graph* G *is the partial function from v-lists to values such that, for each Key in the domain of* $[\![\ ]\!]_G$,

$$[\![A]\!]_G(\mathit{Key}) = f(\{\!\!\{ \ \mid \mu \in [\![\ ]\!]_G(\mathit{Key}), \ = [\![\mathsf{F}]\!]_{\mu,G} \}\!\!\}).$$

Finally, the algebra provides the *AggregateJoin* construct to combine aggregates $A_1, \ldots, A_n$ to form a pattern P. The semantics mandates that $[\![P]\!]_G$ contain a mapping μ_{Key} for each v-list in the domain of all $[\![A_i]\!]_G$; each μ_{Key} de nes variables $?x_i$ to record the values of A_i for that v-list.

DEFINITION 10. *Let* Sparql_X *extend* Sparql. *The language* Sparql^A_X *extends* Sparql_X *by allowing patterns of the form* *AggregateJoin*$_{\mathsf{x}}(\mathsf{A})$, *with* $\mathsf{x} = [?x_1, \ldots, ?x_n]$ *a list of variables and* $\mathsf{A} = [A_1, \ldots, A_n]$ *a list of aggregates. For a graph* G *and intersection of the domains of all* A_i,

$$[\![\mathit{AggregateJoin}_{\mathsf{x}}(\mathsf{A})]\!]_G = \biguplus_{Key \in} \{\!\!\{ \mu \mid \mu = \{?x_i \mapsto v \mid 1 \leq i \leq n, v = [\![A_i]\!]_G(\mathit{Key}), v \neq \mathit{error}\} \}\!\!\}.$$

We also set $\mathsf{var}(\mathit{AggregateJoin}_{\mathsf{x}}(\mathsf{A})) = \{?x_1, \ldots, ?x_n\}$.

Our query (Q4) translates into the algebra as given next, where we have a single group over departments, and aggregates A_2 and A_3 for counting and summation; an additional aggregate A_1 is required to store the keys of the groups and incorporate them into a pattern using *AggregateJoin*:

$$\mathit{Project}(\{?d, ?n\}, \mathit{Extend}(?n, ?v_2, \mathit{Extend}(?d, ?v_1, P_1))), \text{with}$$
$$P_1 = \mathit{Filter}(1 < ?v_3, \mathit{AggregateJoin}_{[?v_1, ?v_2, ?v_3]}([A_1, A_2, A_3])),$$
$$A_1 = \mathit{Aggregate}([?d], \mathsf{Id}, \mathit{Group}([?d], P_2)),$$
$$A_2 = \mathit{Aggregate}([?s], \mathsf{Sum}, \mathit{Group}([?d], P_2)),$$
$$A_3 = \mathit{Aggregate}([?x], \mathsf{Count}, \mathit{Group}([?d], P_2)),$$
$$P_2 = \mathit{Join}((?x, \mathit{department}, ?d), (?x, \mathit{salary}, ?s)).$$

The 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.

THEOREM 6. *Languages* Sparql^A *and* Sparql^A_{PDE} *have the same expressive power.*

PROOF. We first express $\mathit{Extend}(?x, E, P)$ in Sparql^A_P. For $\mathsf{x} = [?x_1, \dots, ?x_n]$ an enumeration of $\mathsf{var}(P)$ let

$$P_E = \mathit{AggregateJoin}_{[?x, ?x_1, \dots, ?x_n]}([A, A_1, \dots, A_n]), \text{ where}$$
$$A_i = \mathit{Aggregate}([?x_i], \mathsf{Id}, \mathit{Group}(\mathsf{x}, P)), \; 1 \le i \le n,$$
$$A = \mathit{Aggregate}([E], \mathsf{Id}, \mathit{Group}(\mathsf{x}, P)).$$

The evaluation of P_E has the same mappings as the evaluation of $\mathit{Extend}(?x, E, P)$, but all with multiplicities 1. Consider the following pattern, with θ a renaming of $\mathsf{var}(P)$, which is fully equivalent to $\mathit{Extend}(?x, E, P)$:

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

Patterns $\mathit{Distinct}(P)$ can be expressed similarly. Finally, *Project* can be pushed upwards through Sparql operators as in Theorem 1. Thus, it suffices to show that *Project* can be eliminated from $\;= \mathit{Group}(\mathsf{E}, \mathit{Project}(X, P))$ in $\mathit{Aggregate}(\mathsf{F}, f, \;)$. We can do so by replacing $\mathit{Project}(X, P)$ in by $P\theta'$, with θ' a renaming of $\mathsf{var}(P) \setminus X$. □

4.3 Normalisation and Simplification

We now show that features (F2) and (F3) in the aggregate 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 11. *A* Sparql^A *query is in* a-normal form *if each group is of the form* $Group(\mathsf{x}, P)$ *with* x *a list of variables and each aggregate is of the form* $\mathit{Aggregate}([E], f, \;)$ *with* f *different from* CountD, SumD*, and* AvgD*.*

Next we show that a-normalisation is always feasible.

THEOREM 7. *Every* Sparql^A *query admits an equivalent* Sparql^A *query in a-normal form.*

PROOF SKETCH. We first show that the aggregate functions with duplicate elimination can be rewritten using their usual counterparts. Let $f\mathsf{D} \in \{\mathsf{CountD}, \mathsf{SumD}, \mathsf{AvgD}\}$ and $A_1 = \mathit{Aggregate}(\mathsf{F}, f\mathsf{D}, \mathit{Group}(\mathsf{E}, P_1))$ with $\mathsf{F} = [F_1, \dots, F_m]$ and $\mathsf{E} = [E_1, \dots, E_k]$. We can check that A_1 is equivalent to the following aggregate A_1', where $\mathsf{x} = [?x_1, \dots, ?x_m]$ and $\mathsf{y} = [?y_1, \dots, ?y_k]$ are lists of fresh variables, and $\cdot$ denotes list concatenation:

$$A_{?x_i} = \mathit{Aggregate}([F_i], \mathsf{Id}, \mathit{Group}(\mathsf{F} \cdot \mathsf{E}, P_1)), \quad 1 \le i \le m,$$
$$A_{?y_j} = \mathit{Aggregate}([E_j], \mathsf{Id}, \mathit{Group}(\mathsf{F} \cdot \mathsf{E}, P_1)), \quad 1 \le j \le k,$$
$$P_1' = \mathit{AggregateJoin}_{\mathbf{x} \cdot \mathbf{y}}(A_{?x_1}, \dots, A_{?x_m}, A_{?y_1}, \dots, A_{?y_k}),$$
$$A_1' = \mathit{Aggregate}(\mathsf{x}, f, \mathit{Group}(\mathsf{y}, P_1')).$$

Second, we prove that grouping over lists of expressions can be reduced to grouping over lists of variables by exploiting the *Extend* operator. For this, we show that an aggregate $A_2 = \mathit{Aggregate}(\mathsf{F}, f, \mathit{Group}(\mathsf{E}, P_2))$ with $\mathsf{E} = [E_1, \dots, E_m]$ is equivalent to the following aggregate A_2', where $\mathsf{x} = [?x_1, \dots, ?x_m]$ is a list of fresh variables:

$$P_2' = \mathit{Extend}(?x_1, E_1, \dots, \mathit{Extend}(?x_n, E_m, P_2) \dots),$$
$$A_2' = \mathit{Aggregate}(\mathsf{F}, f, \mathit{Group}(\mathsf{x}, P_2')).$$

By Theorem 6, *Extend* in P_2' is inessential as it is expressible using normalised grouping and aggregation constructs.

For the last step, note that lists of expressions in aggregates can be reduced to single expressions by aggregating the expressions in the list; e.g., for Avg, aggregating over the list $[E_1, \dots, E_n]$ is equivalent to aggregating over $(\;^{n}_{i=1} E_i)/n$. Unlike the previous two steps, this step is sensitive to the particular aggregate functions available in SPARQL. □

The normal form in Definition 11 already provides a significant simplification of the algebra. Indeed, features (F2) and (F3) are inconsequential; also the definition of aggregate functions can be made more transparent: not only the functions involving duplicate elimination can be dispensed with, but also the function Flatten is inessential since aggregation is 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 SPARQL 1.1 aggregate algebra closer to its relational counterpart, but can also be exploited to make the mapping from SPARQL 1.1 syntax into the algebra much more direct and transparent. The following definition specifies the aforementioned combined operator.

DEFINITION 12. *The language* Sparql^{As} *extends* Sparql *by permitting patterns of the form* $\mathit{GroupAgg}(X, ?z, f, E, P)$*, where* X *is a set of variables,* $?z$ *another variable,* f *an aggregate function,* E *an expression, and* P *a pattern. Given a graph* G *and a mapping* $\mu \in [\![P]\!]_G$*, let*

$$v_\mu = f(\{\!\!\{ v \mid \mu' \in [\![P]\!]_G,\, \mu'|_X = \mu|_X,\, v = [\![E]\!]_{\mu',G} \}\!\!\}),$$

where $\nu|_X$ *is the restriction of* ν *to* X*. Then, the evaluation* $[\![\mathit{GroupAgg}(X, ?z, f, E, P)]\!]_G$ *is the multiset with base set*

$$\{\mu' \mid \mu' = \mu|_X \cup \{?z \mapsto v_\mu\},\, \mu \in [\![P]\!]_G, v_\mu \neq \mathit{error}\} \cup$$
$$\{\mu' \mid \mu' = \mu|_X,\, \mu \in [\![P]\!]_G,\, v_\mu = \mathit{error}\},$$

and multiplicity 1 for each mapping in the base set. We also set $\mathsf{var}(\mathit{GroupAgg}(X, ?z, f, E, P)) = X \cup \{?z\}$*.*

The *GroupAgg* construct is close to the grouping operator in the relational algebra (see [22, Chapter 5]): X represents the set of grouping variables, $?z$ 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.

Query (Q4) can be written in a more natural way as follows (exists is used for succinctness and can be dispensed with):

$\textit{Filter}(\text{exists}(P_2), P_1)$, with
$P_1 = \textit{GroupAgg}(\{?d\}, ?n, \text{Sum}, ?s, P_3)$,
$P_2 = \textit{Filter}(1 < ?v, \textit{GroupAgg}(\{?d\}, ?v, \text{Count}, ?x, P_3))$,
$P_3 = \textit{Join}((?x, \textit{dept}, ?d), (?x, \textit{salary}, ?s))$.

The following theorem establishes that the *GroupAgg* construct captures all grouping and aggregation of SPARQL 1.1.

THEOREM 8. *The languages* Sparql^{As} *and* Sparql^{A} *have the same expressive power.*

PROOF. We first show that every Sparql^{As} pattern $P = \textit{GroupAgg}(\{?x_1, \ldots, ?x_n\}, ?z, f, E, P')$ has an equivalent pattern in Sparql^{A}. We take $\ = \textit{Group}([?x_1, \ldots, ?x_n], P')$, record the values of the grouping variables in each group using aggregates $A_i = \textit{Aggregate}([?x_i], \text{Id}, \ )$, $1 \leq i \leq n$, and capture the value of E using $A = \textit{Aggregate}([E], f, \ )$. Then, P is equivalent to $\textit{AggregateJoin}_{[?x_1, \ldots, ?x_n, ?z]}(A_1, \ldots, A_n, A)$.

For the other direction, we give here a reduction from Sparql^{A} to Sparql^{As}_{P}, the fragment with projection (a reduction to Sparql^{As} is similar but less transparent). Consider a Sparql^{A} pattern $P = \textit{AggregateJoin}_{[?z_1, \ldots, ?z_n]}(A_1, \ldots, A_n)$ in a-normal form with $A_i = \textit{Aggregate}([E_i], f_i, \textit{Group}(\mathsf{x}_i, P_i))$ for $1 \leq i \leq n$. Assume without loss of generality that all x_i are of the same length m since otherwise P evaluates to empty. Let $Y = \{?y_1, \ldots, ?y_m\}$ be fresh variables and, for $1 \leq i \leq n$, let θ_i be a renaming from variables x_i to corresponding variables in Y. We simulate each A_i by pattern $P'_i = \textit{GroupAgg}(Y, ?z_i, f_i, E_i\theta_i, P_i\theta_i)$. We combine these patterns as follows, with ϕ_i, $1 \leq i < n$, renamings of Y to fresh Y_i:

$$P' = \textit{Project}(\{?z_1, \ldots, ?z_n\}, \textit{Filter}(\text{eq}(Y, \phi_1), \ldots, \textit{Filter}(\text{eq}(Y, \phi_{n-1}), \textit{Join}(P'_1\phi_1, \ldots, \textit{Join}(P'_{n-1}\phi_{n-1}, P'_n) \ldots)) \ldots)).$$

We have that P' is equivalent to P. □

5. ANALYTIC AGGREGATE QUERIES

An increasing number of applications of semantic technologies require the analysis of data for effective decision making. In the databases and data warehousing literature, this activity is referred to as OLAP and it involves the execution of complex aggregate queries. In what follows, we exploit our algebra Sparql^{As} to provide a transparent semantics for different types of OLAP queries.

The natural way of thinking about OLAP queries is in terms of a multidimensional data model, which defines measures, such as "sales" in an online store application, and its corresponding dimensions, such as "product", "year", or "country". In this setting, a basic operation consists of aggregating a measure over one or more dimensions (e.g., to determine the total sales per year and product), which can be realised by simply grouping over the relevant dimensions and aggregating over the given measure.

A more complex form of OLAP queries involves aggregating over many subsets of dimensions at the same time. Given k dimensions, a *cube query* aggregates the measure over all of the possible 2^k subsets of these dimensions. The output of the query 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. 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.

Our algebra can be extended with a cube operator in a natural and seamless way as given next.

DEFINITION 13. *For X a set of variables, $?z$ another variable, f an aggregate function, E an expression, and P a pattern, $\textit{Cube}(X, ?z, f, E, P)$ is a pattern with the following semantics, where all is a special value not in* $\mathbf{T}$*:*

$$\biguplus_{Y \subseteq X} \{\!\{ \mu' \mid \mu \in [\![\textit{GroupAgg}(Y, ?z, f, E, P)]\!]_G, \mu' = \mu \cup \{?x \mapsto \textit{all} \mid ?x \in X \setminus Y\} \}\!\}.$$

The special symbol *all* is similar to *error* but has a different semantics. This is in contrast to SQL where NULL values, which carry a different semantics for arithmetic and comparison operators, are used. Rather than a dedicated value *all*, we could have chosen to leave the relevant variables unbound; this, however, would yield counter-intuitive results when further applying operators such as *Join*.

The semantics of *Cube* suggests a straightforward translation to our algebra using *GroupAgg*, *Extend*, and *Union*.

PROPOSITION 3. *Cube is expressible in* Sparql^{As}*.*

PROOF. Given $\textit{Cube}(X, ?z, f, E, P)$, let, for $Y \subseteq X$,

$$P_Y = Extend(?x_1, all, \ldots Extend(?x_n, all, \textit{GroupAgg}(Y, ?z, f, E, P))),$$

where $\{?x_1, \ldots, ?x_n\} = X \setminus Y$. Then $\textit{Cube}(X, ?z, f, E, P)$ is equivalent to $\textit{Union}(P_{Y_1}, \ldots, \textit{Union}(P_{Y_{m-1}}, P_{Y_m}))$ where $Y_1, \ldots, Y_m$ are all the subsets of X. □

We conclude by discussing *window-based* operators, which are heavily used in OLAP queries involving trend analysis over time. 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.

DEFINITION 14. *Given a pattern P, an expression F_θ over $\text{var}(P) \cup \text{var}(P\theta)$ for a renaming θ, a variable $?z \notin \text{var}(P)$, an aggregate function f, and an expression E over $\text{var}(P)$, the construct $\textit{AggWindow}(F_\theta, ?z, f, E, P)$ is a pattern. For a graph G and mapping μ, let*

$$v_\mu = f(\{\!\{ v \mid \mu' \in [\![P]\!]_G, [\![F_\theta]\!]_{\mu \cup \mu'\theta, G} = \textit{true}, v = [\![E]\!]_{\mu', G} \}\!\}).$$

Then the semantics of AggWindow is as follows:

$$[\![\textit{AggWindow}(F_\theta, ?z, f, E, P)]\!]_G = \{\!\{ \mu' \mid \mu \in [\![P]\!]_G, \mu' = \mu \cup \{?z \mapsto v_\mu\}, v_\mu \neq \textit{error} \}\!\} \uplus \{\!\{ \mu \mid \mu \in [\![P]\!]_G, v_\mu = \textit{error} \}\!\}.$$

Note that the expression F_θ specifies a window (i.e., 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.

This operator can also be expressed in our algebra.

PROPOSITION 4. *AggWindow is expressible in* Sparql^{As}*.*

Proof Sketch. Any pattern *AggWindow*$(F_\theta, ?z, f, E, P)$ is equivalent to the pattern

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

where θ' is another renaming of $\mathsf{var}(P)$ to fresh variables and P' = *GroupAgg*$(\mathsf{var}(P), ?z, f, E, \textit{Filter}(F_\theta, \textit{Join}(P, P\theta)))$.

Note that $[\![P']\!]_G$ and $[\![\textit{AggWindow}(F_\theta, ?z, f, E, P)]\!]_G$ coincide when interpreted as sets; the additional transformations are applied to P' to obtain the correct multiplicities. □

6. ADDITIONAL CONSIDERATIONS

We have made several simplifying assumptions that made us deviate from the standard. First, we have omitted the non-deterministic aggregate function GroupConcat; it can be treated similarly to the other aggregate functions. Second, we have considered only expressions already available in SPARQL, whereas SPARQL 1.1 defines a richer language for expressions. Third, we have assumed that patterns do not contain blank nodes. Finally, we have assumed that the result of a query is a multiset of mappings, where the standard defines it as a list. The purpose of this section is to discuss how the last three assumptions affect our results.

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 15. *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 mapping μ and graph G the value $[\![\mathsf{if}(E_1, E_2, E_3)]\!]_{\mu,G}$ is $[\![E_2]\!]_{\mu,G}$ if $[\![E_1]\!]_{\mu,G}$ = true, it is $[\![E_3]\!]_{\mu,G}$ if $[\![E_1]\!]_{\mu,G}$ = false, and error otherwise.*

If $\mathsf{E} = [E_1, \dots, E_n]$ is a list of expressions then $\mathsf{coalesce}(\mathsf{E})$ *is an expression with the following semantics: for μ and G the value $[\![\mathsf{coalesce}(\mathsf{E})]\!]_{\mu,G}$ is error if $[\![E_i]\!]_{\mu,G}$ = error for each $1 \leq i \leq n$, and $[\![E_j]\!]_{\mu,G}$ otherwise, for the smallest j with $[\![E_j]\!]_{\mu,G} \neq$ error.*

We next show that these expressions can be rewritten in terms of Sparql expressions, and hence their introduction is immaterial to our results in this paper.

Proposition 5. *Expressions with* if *and* coalesce *are expressible in both* Sparql *and* Sparql^{As}.

Proof. For Sparql, by Lemma 1, it suffices to eliminate expressions with if and coalesce in *Filter*-subpatterns. This is achieved for if by exhaustively applying the following rule:

$$\textit{Filter}(E[\mathsf{if}(E_1, E_2, E_3)], P) \rightsquigarrow$$
$$\textit{Union}(\textit{Union}(\textit{Filter}(E_1 \wedge E[E_2], P),$$
$$\textit{Filter}(\neg E_1 \wedge E[E_3], P)),$$
$$\textit{Filter}(E[E_1], \textit{SetMinus}(P, \textit{Filter}(E_1 \vee \neg E_1, P)))).$$

Similarly, we replace $\textit{Filter}(E[\mathsf{coalesce}([E_1, E_2])], P)$ with

$$\textit{Union}(\textit{Filter}((E_1 \vee \neg E_1) \wedge E[E_1], P),$$
$$\textit{Filter}(E[E_2], \textit{SetMinus}(P, \textit{Filter}(E_1 \vee \neg E_1, P)))),$$

and coalesce over longer lists can be treated analogously. For Sparql^{As} we need to eliminate if and coalesce from *GroupAgg*-patterns as well, which can be done similarly. □

Blank Nodes SPARQL triple patterns may contain blank nodes in subject and object position, which we disallowed in Section 2. Furthermore, SPARQL introduces BGPs | sets of triple patterns | as a separate construct, which we did not consider. Roughly speaking, blank nodes in BGPs 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 projection: given a BGP P having variables X and blank nodes B, we have that $[\![P]\!]_G = [\![\textit{Project}(X, P\theta)]\!]_G$, where θ is a renaming of B to fresh variables. As shown in Section 3, projection in patterns does not add expressive power to Sparql, and hence neither does allowing BGPs and blank nodes. 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 2 can be easily adapted.

Lists of Solutions We have so far treated the semantics of patterns and queries uniformly in terms of multisets, which simplifies the algebra by avoiding numerous type conversions. The standard, however, defines query evaluation in terms of lists (ordered sequences of mappings). The standard also defines solution modifiers for queries, 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 of mappings, let be the set of all lists of mappings that coincide with when disregarding the order of elements. Furthermore, let h be the function that translates queries from any our algebra Sparql_X into the normative one by adding the necessary type conversions between multisets and lists. Then for every Sparql_X query Q and graph G, we have $[\![Q]\!]_G =$ if and only if $[\![h(Q)]\!]_G^{std} \in$, where $[\![\cdot]\!]^{std}$ 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 the ones using the non-deterministic aggregate function Sample), whereas the evaluation to a list of solutions is non-deterministic even for very simple queries.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented a first in-depth analysis of the SPARQL 1.1 subquery and aggregate 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.

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 property paths, entailment regimes, and query federation. Furthermore, there have been proposals for an extension of SPARQL with stream reasoning and event processing features [5, 10, 13] as well as with analytical queries [19]; it would be interesting to study the connections between these languages and the SPARQL 1.1 normative algebra.

8. ACKNOWLEDGMENTS

Work supported by the Royal Society and the EPSRC projects MaSI[3], Score!, DBOnto, and ED3.

9. REFERENCES

[1] A. Abello, O. Romero, T. B. Pedersen, R. B. Llavori, V. Nebot, M. J. A. Cabo, and A. Simitsis. Using semantic web technologies for exploratory OLAP: A survey. *IEEE TKDE*, 27(2):571{588, 2015.

[2] S. Ahmetaj, W. Fischl, R. Pichler, M. Simkus, and S. Skritek. Towards reconciling SPARQL and certain answers. In *WWW*, pages 23{33, 2015.

[3] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *ISWC*, pages 114{129, 2008.

[4] R. Angles and C. Gutierrez. Subqueries in SPARQL. In *AMW*, 2011.

[5] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: A uni ed language for event processing and stream reasoning. In *WWW*, pages 635{644, 2011.

[6] M. Arenas, S. Conca, and J. Perez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW*, pages 629{638, 2012.

[7] M. Arenas, G. Gottlob, and A. Pieris. Expressive languages for querying the semantic web. In *PODS*, pages 14{26, 2014.

[8] M. Arenas and J. Perez. Querying semantic web data with SPARQL. In *PODS*, pages 305{316, 2011.

[9] E. A. Azirani, F. Goasdoue, I. Manolescu, and A. Roatis. E cient OLAP operations for RDF analytics. In *ICDE Workshops*, pages 71{76, 2015.

[10] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. C-SPARQL: a continuous query language for RDF data streams. *Int. J. Semantic Comput.*, 4(1):3{25, 2010.

[11] C. Buil Aranda, M. Arenas, O. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.*, 18(1):1{17, 2013.

[12] C. Buil Aranda, A. Polleres, and J. Umbrich. Strategies for executing federated queries in SPARQL1.1. In *ISWC*, pages 390{405, 2014.

[13] J. Calbimonte, H. Jeung, O. Corcho, and K. Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Semantic Web Inf. Syst.*, 8(1):43{63, 2012.

[14] S. Cohen. Containment of aggregate queries. *SIGMOD Record*, 34(1):77{85, 2005.

[15] S. Cohen. Equivalence of queries combining set and bag-set semantics. In *PODS*, pages 70{79, 2006.

[16] S. Cohen. Equivalence of queries that are sensitive to multiplicities. *VLDB J.*, 18(3):765{785, 2009.

[17] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM TODS*, 31(2):672{715, 2006.

[18] S. Cohen, W. Nutt, and Y. Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 54(2), 2007.

[19] D. Colazzo, F. Goasdoue, I. Manolescu, and A. Roatis. RDF analytics: lenses over semantic graphs. In *WWW*, pages 467{478, 2014.

[20] R. Cyganiak and D. Reynolds (Editors). The RDF data cube vocabulary. W3C recommendation, W3C, Jan. 2014.

[21] L. Etcheverry and A. A. Vaisman. Enhancing OLAP analysis with web cubes. In *ESWC*, pages 469{483, 2012.

[22] H. Garc a-Molina, J. D. Ullman, and J. Widom. *Database Dystems: The Complete Book*. Pearson Education, 2nd edition, 2009.

[23] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, Mar. 2013.

[24] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *J. ACM*, 48(4):880{907, 2001.

[25] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimanyi. Processing aggregate queries in a federation of SPARQL endpoints. In *ESWC*, pages 269{285, 2015.

[26] R. Kontchakov, M. Rezk, M. Rodriguez-Muro, G. Xiao, and M. Zakharyaschev. Answering SPARQL queries over databases under OWL 2 QL entailment regime. In *ISWC*, pages 552{567, 2014.

[27] E. V. Kostylev and B. Cuenca Grau. On the semantics of SPARQL queries with optional matching under entailment regimes. In *ISWC*, pages 374{389, 2014.

[28] E. V. Kostylev, J. L. Reutter, M. Romero Orth, and D. Vrgoc. SPARQL with Property Paths. In *ISWC*, pages 3{18, 2015.

[29] E. V. Kostylev, J. L. Reutter, and M. Ugarte. CONSTRUCT queries in SPARQL. In *ICDT*, pages 212{229, 2015.

[30] A. Letelier, J. Perez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. *ACM TODS*, 38(4), 2013.

[31] L. Libkin. Logics with counting and local properties. *ACM TOCL*, 1(1):33{59, 2000.

[32] L. Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379{404, 2003.

[33] K. Losemann and W. Martens. The complexity of evaluating path expressions in SPARQL. In *PODS*, pages 101{112, 2012.

[34] J. Perez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.

[35] A. Polleres. From SPARQL to rules (and back). In *WWW*, pages 787{796, 2007.

[36] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, Jan. 2008.

[37] P. Schäuble and B. Wüthrich. On the expressive power of query languages. *ACM TOIS*, 12(1):69{91, 1994.

[38] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4{33, 2010.

[39] N. Schweikardt. Arithmetic, rst-order logic, and counting quanti ers. *ACM TOCL*, 6(3):634{671, 2005.

[40] X. Zhang and J. Van den Bussche. On the primitivity of operators in SPARQL. *Inf. Process. Lett.*, 114(9):480{485, 2014.

[41] X. Zhang and J. Van den Bussche. On the power of SPARQL in expressing navigational queries. *Comput. J.*, 58(11):2841{2851, 2015.

[42] P. Zhao, X. Li, D. Xin, and J. Han. Graph cube: on warehousing and OLAP multidimensional networks. In *SIGMOD*, pages 853{864, 2011.