# Subqueries in SPARQL

Renzo Angles[1] and Claudio Gutierrez[2]

[1] Department of Computer Science, Universidad de Talca
[2] Department of Computer Science, Universidad de Chile

**Abstract.** Subqueries are a poweful feature which allows to enforce reuse, composition, rewriting and optimization in a query language.
In this paper we perform a comprehensive study of the incorporation of subqueries into SPARQL. We consider the many possible choices as suggested by the experience of similar languages, as well as features that developers are incorporating and/or experimenting with. Our study includes the discussion in the SPARQL case of common design issues concerning subqueries (i.e., query correlation, scope of datasets, and optimization), show the interplay among them, and study their implications in terms of expressive power.
Based on this study, we present an extension of SPARQL, with syntax and formal semantics, which incorporates all knwon types of subqueries in a modular fashion and preserving the original semantics.

## 1 Introduction

This paper addresses the design issues raised by the introduction of subqueries to the standard RDF query language SPARQL. By a subquery is usually understood a query that is part of another query.

The advantages of having subqueries in a query language are well known [6,4]; among the most important for SPARQL we can mention incorporation of views, reuse of queries, query rewriting and optimization, and facilitating distributed queries [2]. For SPARQL, whose 2008 Recommendation lacks these features, the issue was early raised by the RDF Data Access Working Group in July 2004 with the name of cascadedQueries [1]. Also it has been gradually incorporated into SPARQL engines, like ARQ and Virtuoso. To the best of our knowledge, there are neither formal semantics defined for these types of queries nor a systematic study of their expressive power and relationships with other functionalities present the language.

There are several desirable design considerations when including a new feature to a language. Among the most important that will guide our study here are: *precise semantics*, hopefully a formal one, that avoids case by case analysis and missing corner cases; *compositional semantics*, that is, the meaning of an expression should be the same wherever it appears, and expressions with equal result types should be allowed to appear in the same contexts. This is crucial when designing subqueries; *Modularity*, that is, each functionality should be "basic" (atomic) and hopefully semantically independent of others. This is

particularly important in subquerying, because queries as part of queries bring with them many features, sometimes in an hidden or undesirable manner. Finally, one always want *simplicity*, which in this case amounts to avoid adding new features if already present. Most of the current proposals lack some or all of these desiderata.

In this paper we study the introduction of subqueries to SPARQL following these guidelines. We study the diverse proposals, both theoretical and practical, that have been presented, analyze their basic constructors, and show their interplay and implications. We unify these diverse constructions in an extension of SPARQL that includes all these features –modulo some consistency necessary constraints–, and extend for them the standard semantics of SPARQL. We present the syntax and a formalized semantics.

This global goal is developed as follows in this paper. First, (in Section 2) we present, via examples, an informal introduction to subqueries in SPARQL. We show different approaches existing and introduce informally the problems and advantages of each of them. Then, (in Section 3) we introduce the definition of a formal semantics for subqueries, which is flexible and expressive enough to cover all known cases. We show that it preserves the original one of SPARQL when the constructs are expressible in SPARQL, and extend the original semantics coherently in the other cases. After this, (in Section 4), we discuss one by one the diverse functionalities that subqueries in SPARQL incorporate (explicitly or as side-effect), like creation of new values, projection in patterns, possibility of choosing set semantics for patterns. Our goal here is to isolate the basic constructors playing behind the different proposed extensions. We study their expresiveness and interrelation. Finally, in Section 5 we summarize our findings and suggest next steps in the process of incorporating subqueries into SPARQL.

## 2 Motivating Examples

There are many choices to incorporate subqueries into SPARQL. Two main approaches emerge naturally: to consider those features suggested by the experience of similar languages, notably SQL; and to consider the features that implementors are already incorporating or exploring.

In this work we follow both. On the one hand, subqueries using known constructors taken from standard languages like SQL (subqueries in WHERE and FROM clauses), and on the other, subqueries in places where the developers are considering necessary (filters for example) and also in places where there is still little experience (e.g. in the NAMED FROM clauses).

Our proposal extends the grammar of SPARQL (complete grammar is presented in the Appendix A). In particular, it includes the following features:

- Extends the FROM clause to permit a construct query as either the default graph or a named graph.
- Allows a select-query in the place of a graph pattern.
- Introduces the set-membership operator IN in order to find a value in a set of values returned by a select query.

– Introduces the quantifier operators SOME and ALL which allow, using a scalar comparison operator (e.g. "="), to compare a value with some or all the values returned by a select-query.
– Introduces the operator EXISTS to allow an ask-query inner a filter constraint.

In this section we motivate these choices via examples, and discuss their implications, interplay, and completeness in detail in Section 4.

### 2.1 Subqueries in dataset clauses

This extension is based on the property that construct-queries return RDF graphs, hence, it is natural to allow the inclusion of these type of queries in dataset clauses FROM or FROM NAMED (which currently only accept references to a graph).

These types of subqueries incorporate the possibility to compose queries automatically, and to specify the composition of several queries, and open the door to modularization and optimization of queries. Correlation of variables, that is, variables occurring both in the inner and outer query, enrich even more this construct.

It is interesting to note that as side-effect of this extension, other features are introduced, the most notably being the creation of values (by using ground triples in the template of the construct-query).

*Example 1.* Mails of pairs of people having a co-authorship relation. The first FROM points to a graph. The second one includes an inner query that outputs a graph whose triples represent coauthors.

```
SELECT ?Mail1 ?Mail2
FROM  u
FROM ( CONSTRUCT { ?Aut1 co-author ?Aut2 }
       FROM bib
       WHERE { ?Art bib:has-author ?Aut1 . ?Art bib:has-author ?Aut2 .
               FILTER ( !(?Aut1 = ?Aut2))} )
WHERE { ?Per1 co-author ?Per2 .
        ?Per1 foaf:mbox ?Mail1 . ?Per2 foaf:mbox ?Mail2 }
```

### 2.2 Subqueries as graph patterns

One of the most natural –at least at first sight– form of incorporating subqueries into SPARQL is to allow a query in the place of a graph pattern. There is one necessary restriction, though, that diminishes its power: In order to be compatible with the original semantics of SPARQL graph patterns, the correlation of variables can not go inside these type of subqueries. Consider for example the join of two such subqueries (SELECT ?X ?Z WHERE { ?X ?Y ?Z }) and (SELECT ?Y ?W WHERE { ?X ?Y ?W }). If $?Y$ is visible outside the former (and similarly with $?X$), which pattern is to be evaluated first?

Hence the need to impose the constraint that only the solution variables be visible outside the select-query (i.e., the correlation of variables is not supported). This restriction kills a great part of the expressivess of the subqueries and hence their value. On the other hand, a strong functionality is also introduced: the possibility to eliminate duplicates in patterns. (See Section 4.2, DISTINCT).

The positive aspects of these types of subqueries is the flexibility given por rewriting and optimization purposes.

*Example 2.* Mails of people having no publications. The implementations simple substract from the database of people those having no articles.

```
SELECT ?Mail FROM u
WHERE {
 {?Per foaf:mbox ?Mail} MINUS {( SELECT DISTINCT ?Per
                                 FROM bib
                                 WHERE {?Art bib:has-author ?Per})}}
```

### 2.3 Subqueries in filter constraints

This extension follows the same design philosophy of subqueries in SQL by introducing the operators IN, SOME, ALL, and EXISTS. It results in the following types of filter conditions:

(1) *Set membership condition*: it uses the IN operator to test whether the result of a select-subquery contains a value (Example 3);
(2) *Quantified condition*: it combines a value, a quantifier operator (SOME/ALL), and a scalar comparison operator (e.g., "<") to test whether the comparison condition is satisfied by some (Example 3) or all (Example 4) the data values resulting of a select-subquery.
(3) *Existencial condition*: it uses the EXISTS operator to enclose and verify whether an ask-subquery has at least one solution (Example 5).

Note that the types (1) and (2) demand a select-subquery with a single result variable. The corresponding negation operators, NOT-IN and NOT-EXISTS, can be represented by using the negation of filter conditions (i.e., "!").

In a previous work [2] we studied these types of queries and proved that nested queries using SOME, ALL and IN can be simulated by using nested queries with the EXISTS operator.

It turns out that subqueries in filter expressions preserve faithfully the original semantics of SPARQL. In particular, allow correlation of variables in outer and inner expressions. This is because the filter is evaluted in a per-mapping form: the subquery is evaluated once for each solution mapping of the outer graph pattern (this is called the *nested iteration method* [6]).

*Example 3.* Name of researchers, whithout duplicates, with at least one paper in *some* edition of the ISWC. (The query also works with `IN` replaced by '`= SOME`').

```
SELECT ?Name FROM bib
WHERE { ?Res bib:name ?Name . ?Art bib:author ?Res . ?Art bib:conf ?Conf .
        FILTER (?conf IN ( SELECT ?ConfX
                           WHERE { ?ConfX bib:series "ISWC" }))}
```

*Example 4.* The names of the oldest people. Note that the MAX aggregate operator is not useful here because we are not asking for the highest age.

```
SELECT ?Name FROM foaf
WHERE { ?Per foaf:name ?Name . ?Per foaf:age ?Age .
        FILTER ( ?Age >= ALL ( SELECT ?AgeX
                               WHERE { ?PerX foaf:age ?AgeX }))}
```

*Example 5.* Researchers with at least one paper in *every* edition of the ISWC (this is the typical query of the division operator).

```
SELECT ?Res FROM bib
WHERE { ?Aut name ?Name .
        FILTER !( ASK
                  WHERE { ?conf series "ISWC" .
                          FILTER !( ASK
                                    WHERE { ?Art author ?Aut .
                                            ?Art conf ?Conf   })})}
```

These examples deserve some comments. The IN operator is less expressive than the SOME operator. It is because the former is restricted to equality of values whereas the latter allows any scalar comparison operator. In fact every query using IN can be rewriten to a query using "= SOME".

Subqueries with SOME/ALL operators without correlated variables are well-suited for query composition (i.e., direct copy/paste of queries). In contrast, the use of EXISTS is not good for query composition because it needs correlated variables to make sense. This helps the user to express complex queries but makes the evaluation harder (because the application of the nested iteration method).

An interesting feature of these subqueries is the natural elimination of duplicates. For instance, Example 6 shows the query in Example 3 expressed using DISTINCT. Although the use of DISTINCT here is simple and clear, it must be stressed that the duplicate elimination is expensive.[1] Thus, the use of subqueries could improve and optimize this task in many cases.

*Example 6.* Name of researchers, without duplicates, with at least one paper in some edition of the ISWC (solution using DISTINCT).

```
SELECT DISTINCT ?Name FROM bib
WHERE { ?Res bib:name ?Name . ?Art bib:author ?Res .
        ?Art bib:conf ?conf . ?conf bib:series "ISWC" }
```

SQL includes an aditional type of subquery called *aggregate subquery*. It extends the definition of subqueries type (2) by allowing an aggregate operator (COUNT, AVG, SUM, MIN, MAX) in the select-subquery (see Example 7).

*Example 7.* Name of people whose age is under the average.

```
SELECT ?Name FROM foaf
WHERE { ?Per foaf:name ?Name . ?Per foaf:age ?Age .
        FILTER ( ?Age < ( SELECT AVG(?AgeX)
                          WHERE { ?PerX age ?AgeX }))}
```

[1] The time it takes to sort the solutions, so that duplicates may be eliminated, is often greater than the time it takes to execute the query itself [5].

## 3 Constructors and Semantics for Subqueries

In order to formally study the interplay of the different constructs presented, we need to define the syntax and semantics of them. This is the content of this section.

We depart from the widely accepted formalization of the syntax and semantics given in [7], and present here the corresponding extensions. Additionally we use the following notation: For a query $Q$ we use the notation $Q = (R, F, P)$ to name explicitly its SELECT/CONSTRUCT/ASK clause (the $R$), its dataset clause (the $F$), and the WHERE clause (the graph pattern $P$).

### 3.1 Syntax

As discussed in the previous section, subqueries in SPARQL can be introduced in each of their three main clauses.[2] Formally, we have the following three families of subqueries, and the fragments summarized in Table 1.

(1) *Subqueries in dataset clauses.* If $u \in I$ and $Q_C$ is a construct-query then the expressions FROM($Q_C$) and FROM NAMED $u(Q_C)$ are dataset clauses.
(2) *Subqueries as graph patterns.* If $Q_S$ is a select-query then the expression $(Q_S)$ is a graph pattern.
(3) *Subqueries in FILTER constraints.* If $Q_A$ is an ask-query then the expressions EXISTS($Q_A$) and $\neg$ EXISTS($Q_A$) are filter constraints.

| | |
|---|---|
| $\text{SPARQL}_{From}$ | SPARQL + (1) |
| $\text{SPARQL}_{SubS}$ | SPARQL + (2) |
| $\text{SPARQL}_{Filter}$ | SPARQL + (3) |
| $\text{SPARQL}_{Ext}$ | SPARQL + (1) + (2) + (3) |

**Table 1.** Different possible extensions of SPARQL with subqueries. On the left the notation for the language; on the right the features incorporated. SPARQL denotes the SPARQL 1.0 version.

Let $Q = (R, F, P)$ be a query. A query $Q'$ is *nested* in $Q$ if and only if $Q'$ occurs in $Q$ as part of one of the expressions defined above. In such case, $Q$ is known as the *outer query* and $Q'$ is known as the *inner query*. If $Q$ does not contain nested queries then $Q$ is called a *flat* query. This definition supports queries with any level of nesting.

Additionally, two queries $Q$ and $Q'$ are *correlated* if and only if $Q'$ is nested in $Q$, and there is some variable occurring in both the graph pattern of $Q$ and the graph pattern of $Q'$. Such variables are called *correlated variables*. This notion of correlated queries will be allowed only for nested queries in filter constrains.

[2] In this work we will avoid subqueries in the SELECT and CONSTRUCT clauses, because, on one hand, although the former are allowed in SQL, they are hardly used, and on the other, there are use cases reported with subqueries in the CONSTRUCT clause.

### 3.2 Semantics

Although apparently simple and intuitive, the semantics of subqueries has several subleties, the principal one being the scope of correlated variables.

A *mapping* $\mu$ is a partial function $\mu : V \to T$. The domain of $\mu$, dom($\mu$), is the subset of V where $\mu$ is defined. The *empty mapping* $\mu_0$ is a mapping such that $\text{dom}(\mu_0) = \emptyset$ . Given a triple pattern $t$ and a mapping $\mu$, $\mu(t)$ is the triple obtained by replacing the variables in $t$ according to $\mu$.

*Scope of variables in a nested query.* Given a query $Q$, as in programming languages, a variables can occur in (some place of) $Q$ as free or bound. The precise recursive definition is as follows: For a query $Q = (R, F, P)$, an occurrence of a variable $?x$ is *free* in $Q$ iff $?x$ occurs free in the pattern $P$ and does not occur in the clause $R$. For a pattern $P$, an occurrence of $?x$ is free iff either: (1) $P$ is a SPARQL pattern and $?x$ occurs in $P$, or (2) $P$ is the pattern of an ASK-query $Q_A$ in a filter constraint, and $?x$ occurs free in $Q_A$.

*Note:* In this paper we do not consider free occurrences in FROM subqueries, because there is yet little background and no known use cases for them. Note also that if $P$ is a SubSelect pattern, all variables occur bound.[3]

Let $Q = (R, F, P)$ be a query and $\mu$ be a mapping. We denote by $\mu(Q)$ the query resulting by replacing each occurrence of a free variable $?x$ in $Q$ by the constant $\mu(?x)$ (recursively if necessary). Note that the same variable $?x$ could occur free and bound in the same query and we are replacing only the free occurrences.

We need one more notion to be ready to give a formal semantics for subqueries. The *answer* for a query $Q = (R, F, P)$, denoted ans($Q$), is a function which takes a set of mappings $[[P]]_{d(F)}$ obtained from evaluating the pattern $P$ over the dataset $d(F)$, and returns: (i) a sequence of mappings when $Q$ is a select-query; (ii) an RDF graph when $Q$ is a construct-query; and (iii) a boolean value (*true* / *false*) when $Q$ is an ask-query. (For the details of this semantics –which stays unchanged here– we suggest to read [8]).

[3] This rather counterintuitive notion is necessary as was illustrated in Section 2.2. In the SPARQL 1.1 draft the variables in the SELECT clause of a SubSelect query are "free", i.e., are exposed to be captured, like in any other pattern, and those "projected" (i.e. not occurring in the SELECT clause) are "local", i.e. invisible from outside.

This SPARQL 1.1 design decision goes counter-clock wise to the behaviour of variables in a standard SELECT query, whose variables in the SELECT clause can be renamed (consistently with those in the pattern) without harm. This is a desirable design feature in any language which wants to be modular and extensible.

Currently, the only "new" feature a SubSelect pattern has as compared to a standard pattern, are the use of constructors available in the SELECT clause (limit, distinct, etc.) to patterns. We believe that it would be better to sincere this decision, incorporating new constructors instead of changing the semantics of a standard construct like the SELECT. Also, if one would like to compose queries (i.e. include on-the-fly in a program, or "paste" another query) it is desirable to have the FROM clause, which currently is absent.

Hence we need to define recursively the set of mapping $[[P]]_{d(F)}$. First, let us define $data(F)$ as the following dataset:

(i) a default graph consisting of the merge of the graphs referred in clauses "FROM $u$", plus the graphs $\text{ans}(Q_C)$ obtained from clauses "FROM($Q_C$)". If there is no such clauses, then the default graph is an empty graph.
(ii) a named graph $\langle u, \text{graph}(u)\rangle$ for each clause "FROM NAMED $u$".
(iii) a named graph $\langle u, \text{ans}(Q_C)\rangle$ for each clause "FROM NAMED $u(Q_C)$".

Now, the recursive definition for the semantics of patterns is as follows:

1. *Subqueries in graph patterns.* Let $(R, F, P)$ be a subselect graph pattern.

$$[[(R, F, P)]]_D = ans(R, F, P)$$

   Note: In SPARQL 1.1 draft, Select-subqueries do not have datasets and the evaluation can be incorporated in this general framework as: $[[(R, \emptyset, P)]]_D = ans(R, F', P)$, where $F'$ is a FROM clause with $d(F') = D$.
2. *Subqueries in FILTER constraints.* Let $P$ be a graph pattern and $Q_A$ be an ask-query.

$$[[P \,\text{FILTER EXISTS}(Q_A)]]_D = \{\mu \in [[P]]_D : \text{ans}(\mu(Q_A)) \text{ is } \mathit{true}\}$$

*Example 8.* The semantics presented gives for correlated queries the standard semantics of the *nested iteration method* [6], i.e., the inner query is performed once for each solution of the outer query. For example, consider the graph pattern

$$((?X \text{ name } ?N) \,\text{FILTER} \neg \,\text{EXISTS}(\text{ASK}(?X \text{ email } ?E))).$$

Considering that $?Y$ is a correlated variable $?Y$, the method establishes that the graph pattern $(\mu(?Y)$ email $?E)$ must be evaluated over and over again, once for each result mapping $\mu$ of the graph pattern $(?X$ name $?N)$.

## 4 Design Issues and Expressive Power

We are now ready to discuss more formally the incorporation of subqueries into SPARQL. We separate this discussion in two parts: 4.1. The comparison of our approach with respect other proposals; and 4.2. The design decisions triggered by the incorporation of these new features. We add a brief speculative subsection 4.3 to indicate a clean and modular extension to be developed in an ideal world.

### 4.1 Current proposals / Related work

In March 2009, the SPARQL Working Group has started up and is currently working, on defining potential extensions to SPARQL. The last working draft[4] of SPARQL 1.1 presents the following features related to our work:

[4] `http://www.w3.org/TR/2010/WD-sparql11-query-20101014/`

– Graph patterns { P FILTER EXISTS { P' } } which test the presence of P'. Here, P' is basically a graph pattern, therefore there is not a direct notion of subquery.
– Two styles of negation: (i) { P FILTER NOT EXISTS { P' } } which test the absence of a match to the pattern P'; and (iii) { P MINUS { P' } } which remove the solutions of P occurring in P' also. Both can be used to represent negation in SPARQL, however in some cases they are are not equivalent (this assuming a direct transformation).
– SubSelects, which consists in to allow SELECT queries within the graph pattern of another query.

In comparison with our proposal: a Subselect share the same RDF dataset as their parent query, and FROM and FROM NAMED clauses are not permitted in subselects; there is not a clear and formal definition of the semantics for query correlation.

Two notions of nested queries have been proposed in works that study translations from SPARQL to Datalog. Polleres [9] suggests that boolean SPARQL queries (i.e., queries having ASK query form) can be safely allowed within filter constraints. Additionally, Schenk [11] proposes the use of views as parts of a dataset, that is, the inclusion of CONSTRUCT queries in FROM clauses. Although, in both cases such extension are well supported by their translations from SPARQL to Datalog, they do not include further developments about issues concerning these extensions.

Angles and Gutierrez [2] develop systematically a SQL-like subqueries in filter constraints for SPARQL, showing that they can be reduced to ASK queries in filter constraints.

Regarding real-life practice, implementations are beginning to provide extensions of SPARQL that include support for some types of nested queries. Virtuoso has included extensions[5] related to nested queries. Among them, it allows an embedded select query in the place of a triple pattern; and filter conditions of the form "EXISTS (<scalar subquery>)". The following pattern shows an example of the latter extension:

```
?x foaf:name "Alice" .
FILTER (EXISTS (SELECT *
                WHERE { ?x foaf:knows ?y . ?y foaf:name "John"})).
```

Similarly, ARQ, the query engine for Jena, supports a type of nested SELECT which uses aggregate functions[6]. For example,

```
?x a :Toy . { SELECT ?x ( COUNT(?order) AS ?q )
              { ?x :order ?order } GROUP BY ?x }
```

None of these implementations present systematic covering nor analysis of these extensions. The extension of Virtuoso corresponds to the inclusion of the

[5] http://www.w3.org/2009/sparql/wiki/Extensions_Proposed_By_OpenLink##Nested_Queries
[6] http://jena.sourceforge.net/ARQ/sub-select.html

select-query in a FROM clause. If one does not consider aggregate functions, not present in current SPARQL, the "EXISTS" extension is equivalent to our definition, and the "SELECT" of ARQ can be simulated by our SOME queries.

Additionally, ARQ allows expressions SERVICE <URI> { $P$ } which can be used, inner a graph pattern, to send the sub-pattern $P$ to the SPARQL endpoint named <URI> (basic federated queries). DARQ [10] offers a single interface for querying multiple, distributed SPARQL endpoints and makes query federation transparent to the client.

### 4.2 Design Issues

**Query composition.** Composed queries enforces both, reuse of queries (by introducing directly either pieces of text in a query or an URI pointing to such query), and rewriting, by allowing distributed evaluation (by pushing the maximum possible information from the WHERE into the FROM clauses). Optimization also by bringing patterns from the FROM into the WHERE.

Query composition is naturally included in SPARQL$_{From}$. Neither current SPARQL nor SPARQL 1.1 allow to express it.

**Creation of New values.** This feature consists in outputting atomic values different from those found in the database to be queried.

SPARQL has no mechanism to create new values. In SPARQL 1.1 this functionality is introduced in the SELECT clause by the construct `AS`. For example a discounted price variable can be expressed as: `(?p*(1-?discount) AS ?price)`. By allowing subSelect queries, this functionality is smuggled into graph patterns.

In our proposal, the fragment SPARQL$_{From}$, allows it by means of introducing what the SPARQL Recommendation calls "ground or explicit triples". The intermediate graph created by a construct subquery can contain new values when its template contains ground triples. Therefore we have the following results about the expressive power of SPARQL$_{From}$.

**Lemma 1.** *SPARQL$_{From}$ allows the creation of new values (i.e., values not existing in the database) in a query.*

**Theorem 1.** *SPARQL$_{From}$ is more expressive than SPARQL.*

**Projections in graph patterns.** This is useful to avoid unnecessary clashes of variables (for example in automatic construction of queries, or even when cutting and pasting pieces of code). Note that this feature is naturally supported in SPARQL$_{SubS}$ because one can project in the select sub-query.

SPARQL does not provide projection in patterns, that is, there are no "local" variables in patterns (or in other terms: all variables in a pattern are part of the solutions of the patterns). A partial (and dirty) shortcut to solve this issue are blank nodes (e.g., the solution mapping to the pattern $\{\_{:}b_1\ ?X\ ?Y\}$ have only $?X$ and $?Y$ as solution variables). There are two problems with this patch solution: blank nodes are not allowed in the predicate positions (although most implementations allow it), and cannot constraint the "variable" $\_{:}b_1$ with filters.

Projection in patterns is the type of feature that does not increase the expressive power of the language, but is very useful for programmers.

**DISTINCT in graph patterns.** This is a delicate issue. It is well known the complexities that introduces the interplay between bag and set semantics [3].

The evaluation of a graph pattern $P$ in SPARQL has bag semantics by default. But in SPARQL$_{SubS}$, one could choose bag or set semantics by writing a subselect `(SELECT DISTINCT * WHERE P)`. Is this desirable? This functionality adds expressive power as the following example shows.

*Example 9.* Assume a database of university people, codified with triples of the form $(u, name, n)$ (name of individual $u$) and $(u, dept, d)$ (department to whom $u$ is attached to). The query "list names of people in either the CS or the Math department" is the following in SPARQL 1.1:

```
SELECT ?N
WHERE {?U  name ?N . { SELECT DISTINCT *
                       WHERE {?U dept CS} UNION {?U dept Math}}}
```

Note that this query has no equivalent one in current SPARQL because one has to project in the SELECT before eliminating duplicates with the DISTINCT. Hence, one could get either more or less names than existing selected individuals. (For example, if there are three Peters in both departments).

**Theorem 2.** *SPARQL$_{SubS}$ is more expressive than SPARQL.*

**Variable Correlation in SPARQL$_{SubS}$.** Correlation of variables is a basic and useful feature of subqueries (as we shown in the examples of Section 2). In SPARQL$_{SubS}$ and SPARQL 1.1, only variables projected by the select subquery are visible to operations outside the subquery.

Allowing correlated variables (for example, through projected variables in a pattern to be visible from outside), would bring an undesirable design: the need to introduce an order to the evaluation of patterns, thus changing its current SPARQL semantics. In fact, to make consistent the design, subSelects should be evaluated at the end. But still in this case we could find troubles like the one signaled in subsection 2.2.

This problem is solved, for example in SQL, by giving the subqueries the function of filters. Filters are evaluated last, and they do not allow crossing of correlated variables because they have a different status than standard predicates (graph patterns in SPARQL). Note that the extension of SPARQL with subqueries in filter constraints and in dataset clauses follow this philosophy, hence do not need any artificial constraint on visibility of variables.

### 4.3 A clean and modular extension

From the previous results, it turns out that if one would like to incorporate all current features for subqueries proposed for SPARQL, but have a modular and

clean language (thus, in the unreal world where one could forget the unavoidable compromises of standardization Committees and Working Groups), this language would be as follows:

1. The extension SPARL$_{From}$
2. The extension SPARQL$_{Filter}$
3. An operator DISTINCT for patterns, which would eliminate duplicate in the multiset of mappings resulting from the evaluation of a pattern. For example, a syntax like {DISTINCT $P$} with the obvious semantics.
4. An operator of projection for patterns, which would hide some of its variables. For example, a syntax like { OUT(?$X$,?$Y$) $P$ } would indicate that the only variables visible from outside in the pattern $P$ are ?$X$ and ?$Y$.

This extension would encompass the SPARQL$_{SubS}$ fragment, hence being equivalent to SPARQL$_{Ext}$ (our extension that embodies all current proposals), but avoiding the noise that bring on subSelect queries.

## 5 Conclusions

In this paper we presented a comprehensive discussion of all possible ways of introducing subqueries intro SPARQL, while preserving the original semantics and spirit.

For this we presented in Section 3 the syntax and semantics of a complete proposal of adding subqueries and composition to SPARQL in almost all possible forms. The arguments presented show that is is complete feasible, adding little overhead over the current specification of SPARQL and in particular without changing the original semantics, to incorporate all the flexibility of subqueries of SQL (that has been systematically tested by users and developers), to include all syntaxes currently in use, plus some new constructs that would add facilities to the users.

We presented expressiveness results of these extensions, analyzed their interplay and their implications, particularly regarding unexpected side-effects produced by the incorporation of some features.

Based on this analysis, we defined a global extension (expressed in Table 1 as SPARQL$_{Ext}$) which incorporates all these extensions –with some restrictions necessary to preserve the original semantics of SPARQL– and gave it a complete syntax (Appendix A) and formal semantics.

To be complete, the discussion should include at some point the incorporation of subqueries in the aggregate constructs that are planned for SPARQL 1.1. Again, we strongly suggest to follow the SQL-philosophy in this regard. We have showed that it can be incorporated to SPARQL with little noise, in a perfectly coherent manner, without altering the original semantics of SPARQL, and adding few syntactic construct with a clear semantics. In this regard, implementations of SPARQL can be modularly extended to include these new features.

## References


1. W3C RDF Data Access Working Group - Issues List. http://www.w3.org/2001/sw/DataAccess/issues.
2. R. Angles and C. Gutierrez. SQL Nested Queries in SPARQL. In *Proc. of the IV Alberto Mendelzon Workshop on Foundations of Data Management*, 2010.
3. S. Cohen. Equivalence of queries combining set and bag-set semantics. In *Proc. of the 25th Symp. on Principles of DB Systems (PODS)*, pages 70–79. ACM, 2006.
4. R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *Proceedings of the 1987 Int. Conf. on Management of data (SIGMOD)*, pages 23–33, New York, NY, USA, 1987. ACM Press.
5. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems - The Complete Book*. Prentice Hall, 2002.
6. W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.
7. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, number 4273 in LNCS, pages 30–43. Springer-Verlag, 2006.
8. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.
9. A. Polleres. From SPARQL to Rules (and back). In *Proceedings of the 16th International World Wide Web Conference (WWW)*, pages 787–796. ACM, 2007.
10. B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *Proc. of the 5th European SW Conf. (ESWC)*, volume 5021 of *LNCS*, 2008.
11. S. Schenk. A SPARQL Semantics Based on Datalog. In *30th Annual German Conf. on Advances in AI (KI)*, volume 4667 of *LNCS*, pages 160–174. Springer, 2007.


## A Proposed Grammar for subqueries in SPARQL

Expressions added to the original grammar of SPARQL occurs in **bold**.

| | | |
|---|---|---|
| DatasetClause | ::= | “FROM” ( DefaultGraphClause \| NamedGraphClause ) |
| DefaultGraphClause | ::= | SourceSelector \| **( “(” ConstructQuery “)” )** |
| NamedGraphClause | ::= | “NAMED” SourceSelector **( “(” ConstructQuery “)” )?** |
| GraphPatternNotTriples | ::= | OptionalGraphPattern \| GroupOrUnionGraphPattern \| GraphGraphPattern \| **SubSelect** |
| **SubSelect** | ::= | **“(” SelectQuery “)”** |
| ValueLogical | ::= | RelationalExpression \| **SubqueryExpression** |
| **SubqueryExpression** | ::= | **QuantifiedExpr** \| **ExistentialExpr** \| **SetMembershipExpr** |
| **QuantifiedExpr** | ::= | VarOrTerm **ComparisonOp ( “SOME”** \| **“ALL”) SingleVarSelect** |
| **ComparisonOp** | ::= | “ = ” \| “! = ” \| “ < ” \| “ > ” \| “ <= ” \| “ >= ” |
| **SetMembershipExpr** | ::= | VarOrTerm **“IN” “(” SingleVarSelect “)”** |
| **ExistentialExpr** | ::= | **“EXISTS”** “(” AskQuery “)” |
| **SingleVarSelect** | ::= | “SELECT” **Var** DatasetClause* WhereClause |