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. The Subqueries functionality is a powerful 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 several possible choices as suggested by the experience of similar languages, as well as features that developers are incorporating and/or experimenting with. Based on this study, we present an extension of SPARQL, with syntax and formal semantics, which incorporates all known types of subqueries in a modular fashion and preserves 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 it is usually understood a query that is nested inside 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.

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; *modularity*, that is, each functionality should be "basic" (atomic) and hopefully semantically independent of others. This is particularly important in subquerying, because queries nested in other queries bring with them many features, sometimes in a hidden or undesirable manner. Finally, one always wants *simplicity*, which in this case amounts to avoiding 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 constructs, and show their interplay and implications. We unify these diverse constructions in an extension of SPARQL that includes all these features –modulo some necessary consistency constraints–, and extend for them the standard semantics of SPARQL. We present the syntax and a formalized semantics.

Our proposal includes the following features: (1) extends the FROM and FROM NAMED clauses by allowing a construct query as input; (2) allows a select-query in the place of a graph pattern; (3) introduces the set-membership operator IN in order to find a value in the sequence of solutions returned by a select-query; (4) 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; and (5) introduces the operator EXISTS to allow an ask-query to be a type of filter constraint.

The paper is organized as follows. In Section 2, we present, via examples, an informal introduction to subqueries in SPARQL. We show different existing approaches and introduce informally the problems and advantages of each of them. In Section 3, we present 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 semantics of SPARQL when the constructs are expressible in SPARQL, and extends it consistently in the other cases. 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. We study their expressiveness and interrelation. In Section 5, we present the works related to our study. Finally, in Section 6 we summarize our findings and suggest future steps in the process of incorporating subqueries into SPARQL.

2 Motivating Examples

In this section we give an informal introduction to subqueries in SPARQL through several motivating examples. In order to describe the basic constructs defined by SPARQL, consider the query expression presented in Example 1.

Example 1. A simple SPARQL query. It returns the names of people whose age is between 18 and 30, plus the names of people having (or not) an email.

```
SELECT DISTINCT ?N
FROM <http://.../people.rdf>
WHERE { {{ ?X name ?N . ?X age ?A } FILTER (?A > 18 && ?A < 30)}
        UNION {{?X name ?N} OPTIONAL {?X email ?E}}   }
```

A SPARQL query consists of three sections. First, a *query form section* which defines the type and result-format of the query: a SELECT query returns a sequence of variable bindings (solutions); an ASK query returns a Boolean value;

and a CONSTRUCT query returns an RDF graph. Second, a *dataset section* which describes the RDF dataset to be queried. This is a sequence of FROM and FROM NAMED clauses indicating the datasets to be queried, including RDF graphs which are identified by and obtained from a given URIs. Third, a *query pattern section* WHERE, which contains an expression representing a graph pattern. A graph pattern is a combination of triple patterns (e.g., `?X name ?N`) and binary operators ".", UNION, OPTIONAL, FILTER, for, respectively, conjunction, disjunction, optional-conjunction, and filtering of graph patterns. Additionally, there are *solution modifiers* which can be used to modify the sequence of solutions obtained from the evaluation (e.g., DISTINCT ensures that the elements in the sequence of solutions are unique). The *evaluation of a SPARQL query* is essentially a matching of the graph pattern against the dataset.

There are many choices to incorporate subqueries into SPARQL[1]. 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 constructs taken from standard languages like SQL (subqueries in WHERE and FROM clauses), and on the other, subqueries in places where the developers are considering it necessary (FILTER constraints for example) and also in places where there is still little experience (e.g. in the NAMED FROM clauses).

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

2.1 SPARQL$_{From}$: Subqueries in dataset clauses

Based on the observation that construct-queries return RDF graphs, a natural extension is to allow the inclusion of such queries in FROM or FROM NAMED clauses (currently accepting only URI references to a graph).

Example 2. Mails of pairs of people having a co-authorship relation. The second FROM clause includes a subquery (lines 3 to 6) that outputs an RDF graph whose triples represent co-authors.

```
1  SELECT ?M1 ?M2
2  FROM  <http://.../people.rdf>
3  FROM ( CONSTRUCT { ?A1 co-author ?A2 }
4         FROM <http://.../biblio.rdf>
5         WHERE { ?Pub author ?A1 . ?Pub author ?A2 . FILTER ( !(?A1 = ?A2))}
6       )
7  WHERE { ?P1 co-author ?P2 . ?P1 mail ?M1 . ?P2 mail ?M2 }
```

[1] In this work we do not consider subqueries in the SELECT and CONSTRUCT clauses because: (1) although the former are allowed in SQL, they are hardly used; (2) there are not use cases reported for the latter.

These types of subqueries (see Example 2) open the door to composition, modularization and optimization of queries. It is interesting to note that, as a side-effect of this extension, the creation of values is introduced (by using ground triples in the template of the construct-query).

2.2 SPARQL$_{SubS}$: Subqueries as graph patterns

Another form to build SPARQL subqueries, is to allow a query in the place of a graph pattern (See Example 3). This extension allows: expressing operations among select-queries (not currently supported by SPARQL, e.g. union, intersection, etc.); projection of solution variables at the level of graph patterns; and elimination of duplicates in patterns.

Example 3. Mails of people having publications. The subquery returns identifiers (without repetitions) of authors from the bibliographic database.

```
SELECT ?Mail FROM <http://.../people.rdf>
WHERE { {?X mail ?Mail} . {( SELECT DISTINCT ?X
                             FROM <http://.../biblio.rdf>
                             WHERE {?Pub author ?X})} }
```

There is one problem with this type of subquery: the correlation of variables should be constrained artificially. Consider for example the following join of two such subqueries:

```
{ (SELECT ?X WHERE { ?A p ?X }) . (SELECT ?A WHERE { ?A q ?X }) }
```

Note the crossed correlation of variables. How should such pattern be evaluated? The SPARQL semantics indicates that each one should be evaluated independently and then "join" the solutions. But in this example, the correlation of variables is recursive: the second should be evaluated once the values of `?X` are known; similarly with the first and `?A`. A simple solution is to avoid correlation, but this decision limits severely the power of subqueries.

2.3 SPARQL$_{Filter}$: Subqueries in filters 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 constraints:

(1) *Set membership condition*: It uses the IN operator to test whether the result of a select-subquery contains a value (Example 4);
(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 4) or all (Example 5) 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 6).

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

Example 4. Names of researchers, without duplicates, with at least one paper in *some* edition of the ISWC. (The query also works with `IN` replaced by '`= SOME`').

```
SELECT ?Name FROM <http://.../biblio.rdf>
WHERE { ?Aut name ?Name . ?Pub author ?Aut . ?Pub conf ?Conf
        FILTER (?Conf IN ( SELECT ?ConfX FROM <http://.../biblio.rdf>
                           WHERE { ?ConfX series "ISWC" }))}
```

Example 5. 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 <http://.../people.rdf>
WHERE { ?Per name ?Name . ?Per age ?Age .
        FILTER ( ?Age >= ALL ( SELECT ?AgeX FROM <http://.../people.rdf>
                               WHERE { ?PerX age ?AgeX }))}
```

Example 6. Emails of people with at least one paper in *every* edition of the ISWC (this is a typical query solved using the division operator).

```
SELECT ?Email FROM <http://.../people.rdf>
WHERE { ?Per email ?Email .
        FILTER !( ASK FROM <http://.../biblio.rdf>
                  WHERE { ?Conf series "ISWC" .
                          FILTER !( ASK FROM <http://.../biblio.rdf>
                                    WHERE { ?Pub author ?Per .
                                            ?Pub conf ?Conf }) }) }
```

It turns out that this type of subqueries preserves faithfully the original semantics of SPARQL. The evaluation follows a per-mapping approach called the *nested iteration method* [6] (i.e., the subquery is evaluated once for each solution of the graph pattern of the main query).

The above examples deserve some comments. The IN operator is less expressive than the SOME operator, 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 rewritten 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 7 shows the query in Example 4 expressed using DISTINCT. Although the use of DISTINCT here is simple and clear, it must be stressed that the duplicate elimination is expensive.[2] Thus, the use of subqueries could improve and optimize this task in many cases.

Example 7. Name of researchers, without duplicates, with at least one paper in some edition of the ISWC (solution using DISTINCT).

```
SELECT DISTINCT ?Name FROM <http://.../biblio.rdf>
WHERE { ?Aut name ?Name . ?Art author ?Aut .
        ?Art conf ?Conf . ?Conf series "ISWC" }
```

In a previous work [2] we studied these types of queries and proved that subqueries using SOME, ALL and IN can be simulated by using the EXISTS operator. In fact, in this paper we restrict our study to subqueries using the EXISTS operator.

Note 1. On could add - like in SQL - an additional type of subquery called *aggregate subquery*. It would extend the definition of subqueries type (2) by allowing an aggregate operator (e.g., AVG) in the subquery. For example, consider the graph pattern {`?Per age ?Age . FILTER ?Age < (SELECT AVG(?AgeX) ... )`}. Although the inclusion of aggregation does not introduce severe complications, we do not consider them in this paper because aggregate operators have not been standardized yet in SPARQL.

3 Syntax and Semantics for Subqueries

In this section the introduce a proposal of syntax and semantics for subqueries in SPARQL. We will follow the widely accepted formalization of SPARQL given in [8], and present here the corresponding extensions. We refer the reader to such citation for additional details about the following definitions.

We assume the pairwise disjoint, infinite sets I, B, L, and V (IRIs, blank nodes, RDF literals, and variables respectively). An *RDF triple* is a tuple from $I \cup B \times I \times I \cup B \cup L$. An *RDF graph* is a finite set of RDF triples. An *RDF dataset* is a set $\{G_0, \langle u_1, G_1 \rangle, \ldots, \langle u_n, G_n \rangle\}$ where each G_i is an RDF graph and each u_j is an IRI. G_0 is the *default graph* and each pair $\langle u_i, G_i \rangle$ is a *named graph*.

A *triple pattern* is an RDF triple with variables. A *SPARQL query* is a tuple $Q = (R, F, P)$ where R is a query form, F is a set of dataset clauses, and P is a graph pattern. Specifically,

– If $W \subset V$ is a set of variables and H is a set of triple patterns (called a *graph template*) then the expressions SELECT W, CONSTRUCT H, and ASK are *query forms*.

[2] 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](Sec. 6.4.1).

– If $u \in I$ then the clauses FROM u and FROM NAMED u are *dataset clauses.*
– A *graph pattern* is defined recursively as follows. A triple pattern is a graph pattern. If P_1 and P_2 are graph patterns then the expressions $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$ and $(P_1 \text{ UNION } P_2)$ are graph patterns. If P is a graph pattern and C is a filter constraint then the expression $(P \text{ FILTER } C)$ is a graph pattern. A *filter constraint* is a combination of equality predicates, built-in functions, and Boolean operators ('$\neg$', '$\wedge$' and '$\vee$').

3.1 Syntax for Subqueries

We extend the syntax of SPARQL, as defined above, and introduce the following three families of subqueries:

(1) SPARQL$_{From}$: *Subqueries in dataset clauses.* If $u \in I$ and Q_C is a construct-query, then FROM(Q_C) and FROM NAMED $u(Q_C)$ are dataset clauses.
(2) SPARQL$_{SubS}$: *Subqueries as graph patterns.* If Q_S is a select-query, then the expression (Q_S) is a graph pattern.
(3) SPARQL$_{Filter}$: *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.

Flat and nested queries. 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.

3.2 Semantics for Subqueries

In order to give a formal semantics for subqueries, we introduce some basic definitions about SPARQL semantics (for the details we suggest to read [8]).

A *mapping* μ is a partial function $\mu : V \to I \cup B \cup L$. The domain of μ, dom(μ), is the subset of V where μ is defined. The *empty mapping* μ_0 is a mapping such that dom$(\mu_0) = \emptyset$.

Scope of variables. Given a query Q, as in programming languages, a variable 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 graph 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.[3]

[3] 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 then all variables occur bound in P.

Query correlation. Two queries Q and Q' are *correlated* iff Q' is nested in Q, and there is some variable occurring free in both, the graph pattern of Q and in the graph pattern of Q'. Such variables are called *correlated variables.* This notion of correlated queries will be allowed only for subqueries in filter constrains.

Let Q be a query and μ be a mapping. We denote by $\mu(Q)$ the query resulting by replacing each free occurrence of a 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.

Informally, the *answer* for a query $Q = (R, F, P)$ is a function $\text{ans}(Q)$ which 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.

Semantics of SPARQL$_{From}$. Let F be a set of dataset clauses (that is, FROM u, FROM(Q_C), FROM NAMED u, and FROM NAMED(Q_C) clauses). The dataset D obtained from F contains:

(i) A default graph which consists 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 are no such clauses, then the default graph is the 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 $u(Q_C)$.

Semantics of SPARQL$_{SubS}$ *and* SPARQL$_{Filter}$. The evaluation of graph pattern P over a dataset D, denoted $[[P]]_D$ (as defined in [8]), is extended as follows:

– If P is a subSelect graph pattern (R, F, P), then $[[(R, F, P)]]_D = ans(R, F, P)$.
– Let P' be a graph pattern and Q_A be an ask-query. If P is the graph pattern $(P' \text{ FILTER EXISTS}(Q_A))$ then

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

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

4 Design Issues and Expressive Power

In this section we discuss five orthogonal issues susceptible of being incorporated to the language: query composition, creation of new values, projection in graph patterns, elimination of duplicates, and correlation of variables.

Query composition. Composition of queries enforces: (1) reuse of queries, by introducing directly either pieces of text in a query or an URI pointing to such query; (2) rewriting, by allowing distributed evaluation (by pushing the maximum possible information from the WHERE into the FROM clauses); and (3) optimization, by bringing patterns from the FROM into the WHERE clause.

Neither current SPARQL nor SPARQL 1.1 are able to express query composition. In contrast, this feature is naturally included in SPARQL$_{From}$.

Creation of New values. This feature consists in the facility to output 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.

The proposed fragment SPARQL$_{From}$ allows the creation of values. Specifically, the intermediate graph created by a construct subquery can contain new values when its template contains triples with no variables (known as ground or explicit 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. SPARQL does not provide projection in graph patterns, that is, there are no "local" variables at the level of graph patterns (or in other terms, all variables in a pattern are part of the solutions of the pattern). A partial (and dirty) shortcut to solve this issue are blank nodes (e.g., the solution mapping to the pattern $\{\_{:}b_1\ ?X\ ?Y\}$ has only $?X$ and $?Y$ as solution variables). There are two problems with this patch solution: (i) blank nodes are not allowed in predicate positions (although most implementations allow it); (ii) the "variable" $\_{:}b_1$ cannot be constrained with filters.

Note that this feature is naturally supported in SPARQL$_{SubS}$ because one can project variables in a select subquery. 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). Projection in graph 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. The evaluation of a graph pattern in SPARQL has bag semantics by default. This is an important issue considering the complexities that introduces the interplay between bag and set semantics [3].

In SPARQL$_{SubS}$, one could choose bag or set semantics by writing a subSelect `(SELECT DISTINCT * WHERE { ... })`. This functionality adds expressive power as the following example shows.

Assume a database of university people, codified with triples of the form $(u, name, n)$ (u's name is n) and $(u, dept, d)$ (u is attached to department d). 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"}}} }
```

Considering that the DISTINCT will be evaluated before the projection of `?N`, the query could return either more or less names than existing selected individuals (for example, if there are three Peters in both departments). This query has no equivalent one in current SPARQL because one has to project in the SELECT before eliminating duplicates with the DISTINCT.

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

Variable correlation. Correlation of variables is a basic and useful feature of subqueries (as 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 into 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 Section 2.2.

This problem is solved, for example in SQL, by giving the subqueries the function of filters. Filters are evaluated last. 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.

5 Current proposals / Related work

The current working draft[4] of SPARQL 1.1 presents the following features related to our work:

- Graph patterns { P FILTER EXISTS { P' } } where P' is basically a graph pattern to test. Therefore there is not a direct notion of subquery.
- Two styles of negation: (i) { P FILTER NOT EXISTS { P' } } which tests the absence of a match to the pattern P'; and (ii) { P MINUS { P' } } which removes the solutions of P occurring also in P'. Both can be used to represent negation in SPARQL, however in some cases they are not equivalent.
- SubSelects, which consist in allowing a select-query within the graph pattern of another query.

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

Differently to our proposal, in SPARQL 1.1 FROM and FROM NAMED clauses are not permitted in subSelects, hence a subSelect is forced to share the same RDF dataset with its parent query. Moreover, it has no formal definition for the semantics of query correlation.

Two notions of subquery have been proposed in works that study translations from SPARQL to Datalog. Polleres [9] suggests that Boolean 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 extensions are well supported by their translations from SPARQL to Datalog, they do not include further developments on issues arising from these extensions.

Regarding real-life practice, implementations are beginning to provide extensions of SPARQL that include support for some types of subqueries. Virtuoso includes extensions[5] related to subqueries. Among them, it allows an embedded select query in the place of a triple pattern; and filter constraints of the form "EXISTS (<scalar subquery>)". Similarly, ARQ , the query engine for Jena, supports a type of subSelect which uses aggregate functions[6]. For example, consider the pattern `{ ?x a :Toy . { SELECT ?x ( COUNT(?order) AS ?q) ...}}`.

Additionally, ARQ allows expressions SERVICE <URI> { P }, which can be used to send and execute the graph pattern P into 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.

The extension of Virtuoso corresponds to the inclusion of the 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.

None of these implementations present systematic covering nor analysis of these extensions.

6 Conclusions

In this paper we presented a comprehensive discussion of several ways of introducing subqueries into SPARQL, while preserving the original semantics and spirit. The arguments presented in Section 2, show that it is completely feasible to incorporate all the flexibility of subqueries 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.

In this direction we presented in Section 3 the syntax and semantics of a complete proposal of adding subqueries and composition to SPARQL in all reasonable forms. Our extension can be done while preserving the original semantics and adding negligible overhead over the current SPARQL.

[5] `http://www.w3.org/2009/sparql/wiki/Extensions_Proposed_By_OpenLink#Nested_Queries`

[6] http://jena.sourceforge.net/ARQ/sub-select.html

We presented (Section 4) expressiveness results of these extensions, analyzed their interplay and their implications, particularly regarding unexpected side-effects produced by the incorporation of certain features.

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. 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, current implementations of SPARQL can be modularly extended to include these new features.

Acknowledgements. R. Angles was supported by Fondecyt Project No. 11100364. C. Gutierrez was supported by Fondecyt Project No. 1090565.

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.