SQL Nested Queries 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. SPARQL currently does not include any form of nested queries. In this paper we present a proposal to incorporate nested queries into SPARQL along the design philosophy of SQL nested queries. We present rewriting algorithms and show that all the proposed nested queries can be expressed in a natural and simple extension of SPARQL syntax.

1 Introduction

One of the most powerful features of a query language is the nesting of queries, that is, the possibility of writing in a single expression a query which uses the output of other queries. The current W3C recommendation of SPARQL [13] does not include any form of nesting, although it has been considered as an issue by the RDF Data Access Working Group, and has been gradually incorporated into SPARQL engines.

For SPARQL, the incorporation of nested queries has several motivations. One of the most important is the *reuse of queries.* Once a query is executed, the user may presumably direct the output to some storage medium, assign an IRI to it and then run a query against that extract. Also, with the right interface, the user might be able to just cut-and-paste a query that was debugged into ad-hoc queries. Hence, this feature would allow to build queries incrementally from separately debugged pieces. Another important motivation, as SPARQL was thought of to work on a distributed environment like the Web, is the notion of *distributed queries.* A SPARQL user will have access to vast and time-varying input RDF graphs, containing huge volumes of data that is not of interest to the user. Hence the query computation can be distributed and only relevant data used to compute the final query. For example, AllegroGraph supports queries with distributed databases. A third and important motivation is *query rewriting.* Complex queries can be structured in a way more intuitive and understandable for the user. Examples of these uses are provided in the paper.

Introducing nesting in a coherent form, that is, providing a clean syntax and semantics, is a delicate task. There are several models and “philosophies”, the most widely known those of SQL and XQuery. Based on the similarities between SQL and SPARQL [1,4], it seems natural to investigate how the SQL nesting model can be introduced into SPARQL. In this paper, we address this challenge by investigating systematically the behavior of SQL nesting operators in the context of SPARQL.

There have been several proposals for introducing nesting in SPARQL. Indeed, it has been considered as an issue by the RDF Data Access Working Group. It was raised on July 2004 and was denominated *cascadedQueries*[1]. Currently, the W3C Working Group of SPARQL is working on new features for the language [9]. Among them, the notion of *Subqueries* (to nest a query within another query) is a required feature. As a possible type of subquery, the working draft of SPARQL 1.1 [7] introduces the notion of *subselect*, that is to allow a SELECT query to be a graph pattern.

Regarding real-life practice, some implementations of SPARQL provide extensions that include support for some types of nested queries. ARQ, the query engine for Jena, supports a type of nested SELECT which uses aggregate functions[2]. Virtuoso has also included some extensions[3] related to nested queries. Among them, an embedded select query in the place of a triple pattern, and filter conditions of the form "exists (<scalar subquery>)". None of these proposals and/or implementations present systematic covering and analysis of these extensions, nor a formal semantics for them. It is important to note that all of them introduce nested queries as a new pattern of the form `(SELECT ....)`. This approach introduces several design decisions that are either non-desirable or not necessary at this level. Minor problems are the creation of values at the pattern level by allowing patterns like `(SELECT ?X  ?Y AS 5 ...)`, and the introduction of projection in patterns by allowing patterns of the form `(SELECT ?X ?Y  WHERE P(?X))` where `?Y` does not appear in the pattern `P`. A more relevant problem is introduced once correlation of variables with subqueries is accepted, a desirable and necessary functionality when dealing with nesting. Either the original SPARQL unorder evaluation strategy of patterns or the standard semantics of correlation would need to be reworked. For example, consider the graph pattern `P(?X) AND (SELECT ?Y WHERE R(?X,?Y))`, where `?X` is a correlated variable. The standard semantics would evaluate *first* `P` and *then* for each value of `?X` the corresponding instance of the `SELECT` pattern. In the proposals presented it is not clear how to deal with this issue. Additionally, we do not consider the orthogonal functionality of composition which in SPARQL would correspond to the discussion of how to nest queries in the FROM and FROM NAMED clauses. Schenk [14] proposes the use of views as parts of a dataset, that is, the inclusion of CONSTRUCT queries in FROM clauses. We do not address this topic in this paper.

An alternative approach to the introduction of `SELECT` as a new type of pattern, is to incorporate nesting as a *filtering device*. An early attempt in this direction is Polleres [12], where it is suggested that boolean SPARQL queries (i.e., queries having ASK query form) can be safely allowed within filter constraints, but the extension is not developed. In this paper we develop this design philosophy to its end based on the design philosophy of SQL nested queries,

[1] `http://www.w3.org/2001/sw/DataAccess/issues#cascadedQueries`

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

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

which restrict the nested queries to a form of filtering in the WHERE condition. This approach allows to have a clean semantics for correlated variables, permits to modularly extend the language, and naturally extend the original SPARQL semantics. We develop this feature through the inclusion of SQL-like nested queries in usual SPARQL filter constraints. In particular, we present the following contributions. First, we present a proposal to incorporate nested queries to SPARQL along the design philosophy of SQL, by presenting the syntax and a formal semantics completely compatibles to the current one, and discuss design features. Second, we show, via illustrative examples, that all SQL facilities are also relevant in SPARQL, and present a set of equivalences and rewriting rules among them. Third, we prove that all classical SQL nesting operators (i.e., IN, SOME/ANY, ALL and EXISTS) can be reduced into one of them (i.e., EXISTS), hence proving that all standard nesting constructs can be expressed with the standard filter part of a SPARQL query.

The paper is organized as follows: Section 2 presents the syntax and the semantics of the extension of SPARQL with nested queries. Section 3 presents examples of nested queries. Section 4 presents the algorithms to rewrite among nested queries. Finally, Section 5 presents some conclusions.

1.1 Nested queries in SQL

SQL is a paradigmatic example of the power of nested queries. In fact, this feature plays and important role in SQL for several reasons [16]. SQL (as defined in the ANSI/ISO SQL-92) allows nesting of query blocks in FROM and WHERE clauses, in any level of nesting. In the FROM clause, a subquery is *imported* and used to conform the set of relations to be queried. A subquery in the WHERE clause can be either aggregate (it returns a single value due to an aggregate operator) or non-aggregate (it returns either a set of values or empty, i.e., a SELECT query).

Let Q_A be an aggregate query, Q_S and Q_* be non-aggregate queries where Q_S returns one-column relation (i.e., it has a single projection predicate), and θ be a scalar comparison operator ($<, \leq, >, \geq =, \neq$). A selection predicate containing nested queries can be of the form:

(1) $\langle$*value* | *attribute*$\rangle$ θ (Q_A) (*value-set comparison predicate*).
(2) $\langle$*value* | *attribute*$\rangle$ IN | NOT-IN (Q_S) (*set-membership predicate*).
(3) $\langle$*value* | *attribute*$\rangle$ θ SOME | ALL(Q_S) (*quantified predicate*).
(4) EXISTS | NOT-EXISTS(Q_*) (*existential predicate*).

For example, consider the relations EMPLOYEES(EMP#,NAME,SAL,DEPT) and DEPARTMENTS(DEPT#,NAME,LOCATION). The following expression shows a SQL nested query Q_1 with aggregate (Q_2) and non-aggregate (Q_3) subqueries.

```
SELECT E.NAME FROM EMPLOYEES E                              (Q1)
WHERE E.SAL >
      (SELECT AVG(F.SAL) FROM EMPLOYEES F                   (Q2)
       WHERE F.DEPT IN
            (SELECT D.DEPT# FROM DEPARTMENTS D              (Q3)
             WHERE D.LOCATION = 'DENVER') );
```

2 Syntax and Semantics of nested queries

In this section we extend the syntax and semantics of SPARQL to support nested queries. This extension, is based on the ideas and syntax of SQL nested queries as presented above. Considering that aggregate operators for SPARQL have not been defined yet, we are only considering non-aggregate queries (i.e., SELECT and ASK queries) as nested queries. Hence, we have not included the value-set comparison predicates of SQL as defined before. The definition of this extension follows the formalization presented in [11].

2.1 Preliminaries: The RDF Model and RDF Datasets

Assume there are pairwise disjoint infinite sets I, B, L (IRIs, blank nodes, and RDF literals respectively). We denote by T the union $I \cup B \cup L$ (RDF *terms*). A tuple $(v_1, v_2, v_3) \in (I \cup B) \times I \times T$ is called an *RDF triple*, where v_1 is the *subject*, v_2 the *predicate*, and v_3 the *object*. An *RDF Graph* [10] (just graph from now on) is a set of RDF triples. Given a graph G, we denote by $\mathrm{term}(G)$ the set of elements of T appearing in G and by $\mathrm{blank}(G)$ the set of blank nodes in G. If G is referred to by an IRI u, then $\mathrm{graph}(u)$ returns the graph available in u, i.e, $G = \mathrm{graph}(u)$.

We define two operations on two graphs G_1 and G_2. The *union* of graphs, denoted $G_1 \cup G_2$, is the set theoretical union of their sets of triples. The *merge* of graphs, denoted $G_1 + G_2$, is the graph $G_1 \cup G_2'$ where G_2' is the graph obtained from G_2 by renaming its blank nodes to avoid clashes with those in G_1.

An *RDF dataset* is a set $D = \{G_0, \langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ where each G_i is a graph and each u_j is an IRI. G_0 is called the *default graph* and each pair $\langle u_i, G_i\rangle$ is called a *named graph*. Every dataset satisfies that: (i) it always contains one default graph, (ii) there may be no named graphs, (iii) each u_j is distinct, and (iv) $\mathrm{blank}(G_i) \cap \mathrm{blank}(G_j) = \emptyset$ for $i \neq j$. Given D, we denote by $\mathrm{term}(D)$ the set of terms occurring in the graphs of D. The default graph of D is denoted $\mathrm{dg}(D)$. For a named graph $\langle u_i, G_i\rangle$ define $\mathrm{name}(G_i)_D = u_i$ and $\mathrm{graph}(u_i)_D = G_i$; otherwise $\mathrm{name}(G_i)_D = \emptyset$ and $\mathrm{graph}(u_i)_D = \emptyset$. We denote by $\mathrm{names}(D)$ the set of IRIs $\{u_1, \ldots, u_n\}$. Although $\mathrm{name}(G_0) = \emptyset$, we sometimes will use g_0 when referring to G_0. Finally, the *active graph* of D is the graph G_i used for querying the dataset.

2.2 Syntax of nested queries

Assume the existence of an infinite set V of variables disjoint from T. Let $\mathrm{var}(\alpha)$ the function which returns the set of variables occurring in the structure α.

A *triple pattern* is a tuple in $(T \cup V) \times (I \cup V) \times (T \cup V)$. A *nested query* is a tuple (R, F, P)[4] where R is a result query form, F is a set –possibly empty– of dataset clauses, and P is a graph pattern. Next we define each component.

[4] In this paper we do not consider the solution modifiers defined in [13].

(1) 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 *result query forms.*

(2) If $u \in I$ and Q_C is a query of the form (CONSTRUCT H, F, P), then the expressions FROM u and FROM NAMED u are *dataset clauses.*

(3) A *filter constraint* is defined recursively as follows:
  - If $?X, ?Y \in V$ and $v \in I \cup L$ then $?X = v$, $?X = ?Y$, and bound($?X$) are (*atomic*) filter constraints[5].
  - If $u \in T$, θ is a scalar comparison operator ($=, \neq, <, <=, >, >=$), and $Q_{?X}$ is a query of the form (SELECT $?X, F, P$), then the expressions (u θ SOME($Q_{?X}$)), (u θ ALL($Q_{?X}$)) and (u IN ($Q_{?X}$)) are filter constraints.
  - If Q_A is a query of the form (ASK, F, P), then the expression EXISTS(Q_A) is a filter constraint.
  - If C_1 and C_2 are filter constraints, then $(\neg C_1)$, $(C_1 \wedge C_2)$, and $(C_1 \vee C_2)$ are (*complex*) filter constraints.

(4) 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 AND P_2), (P_1 OPT P_2), (P_1 UNION P_2), and (P_1 MINUS P_2) are graph patterns.[6]
  - If P is a graph pattern and $u \in I \cup V$ then the expression (u GRAPH P) is a graph pattern.
  - If P is a graph pattern and C is a filter constraint then the expression (P FILTER C) is a graph pattern.

Let $Q = (R, F, P)$ be a query. A query Q' is *nested* in Q if and only if Q' occurs in the graph pattern P, i.e., when Q' is nested in P. 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.

Note that, nested queries in SPARQL have been defined by extending the definition of filter constraints with SQL-like predicates for nesting, specifically by including the IN, SOME, ALL and EXISTS operators. The corresponding opposite operators of nesting can be represented by using the negation of filter constraints, i.e., NOT-IN and NOT-EXISTS are expressed as $(\neg(u\,\mathrm{IN}(Q_{?X})))$ and $(\neg\,\mathrm{EXISTS}(Q_A))$ respectively.

We have defined two explicit restrictions about inner queries. On the one hand, filter expression using IN, ALL and SOME are restricted to use SELECT queries with a single projection-variable. On the other hand, filter expressions using EXISTS are restricted to use ASK queries. The latter condition has been included for simplicity. In practice, EXISTS filters could have queries having any result query form (e.g. SELECT), because the EXISTS condition does not really use the results of the inner query at all.

[5] For a complete list of atomic filter constraints see the SPARQL specification [13]

[6] The MINUS operator is not defined in the SPARQL specification, however it can be simulated by a combination of the OPT and FILTER operators [1].

Similar to SQL, the extension presents two features inherent to query nesting. First, the language allows queries with any level of nesting. However, it is well-known that queries with more than two levels of nesting are not recommended in practice because makes the query more difficult to read, understand, maintain and increases the execution time [16]. Second, variables from an outer query block can be accessed inside a nested query block. Such variables, called *correlated variables*, perform as *outer references* from the inner query to the outer query. A subquery containing correlated variables is called a *correlated subquery.*

2.3 Semantics of nested queries

A *mapping* μ is a partial function $\mu : V \to T$. The domain of μ, $\text{dom}(\mu)$, is the subset of V where μ is defined. The *empty mapping* μ_0 is a mapping such that $\text{dom}(\mu_0) = \emptyset$. Given a triple pattern t and a mapping μ such that $\text{var}(t) \subseteq \text{dom}(\mu)$, $\mu(t)$ is the triple obtained by replacing the variables in t according to μ. Abusing notation, for a query Q, we denote by $\mu(Q)$ the query resulting from replacing variables in Q according to μ.

Two mappings μ_1 and μ_2 are *compatible* when for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it satisfies that $\mu_1(?X) = \mu_2(?X)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping. The operations of *join*, *union*, *difference* and *left outer-join* between two sets of mappings Ω_1 and Ω_2 are defined as follows:

- $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are compatible}\}$
- $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$
- $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{ for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$
- $\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

The *answer* for a query $Q = (R, F, P)$, denoted $\text{ans}(Q)$, is a function which returns: (i) a set of mappings when R is a SELECT query; (ii) an RDF graph when R is a CONSTRUCT query; and (iii) a boolean value (*true / false*) when R is an ASK query. We will use this informal definition of $\text{ans}(\cdot)$ to define the semantics for the components of a query.

(1) *Semantics of result query forms.* Let μ be a mapping and R be a result query form. The result of R given μ, denoted $\text{result}(R, \mu)$, is defined as follows:
   - If R is SELECT W then $\text{result}(R, \mu)$ is the restriction of μ to W, that is the mapping denoted $\mu_{|W}$ such that $\text{dom}(\mu_{|W}) = \text{dom}(\mu) \cap W$ and $\mu_{|W}(?X) = \mu(?X)$ for every $?X \in \text{dom}(\mu_{|W})$.
   - If R is CONSTRUCT H then $\text{result}(R, \mu)$ is the set of RDF triples (i.e. an RDF graph) $\{\mu(t) \mid t \in H \text{ and } \mu(t) \subset (I \cup B) \times I \times T\}$.
   - If R is ASK then $\text{result}(R, \mu)$ is *false* if $\mu = \emptyset$ and *true* otherwise.

(2) *Semantics of dataset clauses.* Let F be a set of dataset clauses. The dataset resulting from F, denoted $\text{dataset}(F)$, contains:
   (i) a default graph consisting of the merge of the graphs referred in clauses FROM u. If there is no FROM u, then the default graph is an empty graph $G_0 = \emptyset$; and

(ii) a named graph $\langle u, \text{graph}(u)\rangle$ for each dataset clause "FROM NAMED u".

(3) *Semantics of filter constraints.* Let μ be a mapping and C be a filter constraint. We say that μ satisfies C, denoted $\mu \models C$, if:
  - C is $?X = v$, $?X \in \text{dom}(\mu)$, and $\mu(?X) = v$;
  - C is $?X = ?Y$, $?X \in \text{dom}(\mu)$, $?Y \in \text{dom}(\mu)$, and $\mu(?X) = \mu(?Y)$;
  - C is bound$(?X)$ and $?X \in \text{dom}(\mu)$;
  - C is $(\neg C_1)$ and it is not the case that $\mu \models C_1$;
  - C is $(C_1 \vee C_2)$ and $\mu \models C_1$ or $\mu \models C_2$;
  - C is $(C_1 \wedge C_2)$, $\mu \models C_1$ and $\mu \models C_2$.
  - C is $(u\ \theta\ \text{SOME}(Q_{?X}))$ and there exists a mapping $\mu' \in \text{ans}(\mu(Q_{?X}))$ satisfying that either $u\ \theta\ \mu'(?X)$ when $u \in I \cup L$ or $\mu(u)\ \theta\ \mu'(?X)$ when $u \in V$.
  - C is $(u\ \theta\ \text{ALL}(Q_{?X}))$ and for every mapping $\mu' \in \text{ans}(\mu(Q_{?X}))$ it holds that either $u\ \theta\ \mu'(?X)$ when $u \in I \cup L$ or $\mu(u)\ \theta\ \mu'(?X)$ when $u \in V$.
  - C is $(u\ \text{IN}\ (Q_{?X}))$ and there exists a mapping $\mu' \in \text{ans}(\mu(Q_{?X}))$ satisfying that either $u\ \theta\ \mu'(?X)$ when $u \in I \cup L$ or $\mu(u)\ \theta\ \mu'(?X)$ when $u \in V$.
  - C is EXISTS(Q_A) and ans$(\mu(Q_A))$ is *true*.

(4) *Semantics of graph patterns.* The *evaluation* of a graph pattern P over a dataset D with active graph G, denoted $[\![\cdot]\!]^D_G$, is defined recursively as follows:
  - P is a triple pattern then $[\![P]\!]^D_G = \{\mu \mid \text{dom}(\mu) = \text{var}(P) \text{ and } \mu(P) \subseteq G\}$
  - $[\![(P_1 \text{ AND } P_2)]\!]^D_G = [\![P_1]\!]^D_G \bowtie [\![P_2]\!]^D_G$.
  - $[\![(P_1 \text{ OPT } P_2)]\!]^D_G = [\![P_1]\!]^D_G ⟕ [\![P_2]\!]^D_G$.
  - $[\![(P_1 \text{ UNION } P_2)]\!]^D_G = [\![P_1]\!]^D_G \cup [\![P_2]\!]^D_G$.
  - $[\![(P_1 \text{ MINUS } P_2)]\!]^D_G = [\![P_1]\!]^D_G \setminus [\![P_2]\!]^D_G$.
  - If $u \in I$ then $[\![(u \text{ GRAPH } P_1)]\!]^D_G = [\![P_1]\!]^D_{\text{graph}(u)_D}$.
  - If $?X \in V$ and $\mu_{?X \to v}$ is a mapping such that $\text{dom}(\mu) = \{?X\}$ and $\mu(?X) = v$, then
    $[\![(?X \text{ GRAPH } P_1)]\!]^D_G = \bigcup_{v\ \in\ \text{names}(D)}([\![P_1]\!]^D_{\text{graph}(v)_D} \bowtie \{\mu_{?X \to v}\})$.
  - $[\![(P_1 \text{ FILTER } C)]\!]^D_G = \{\mu \mid \mu \in [\![P_1]\!]^D_G \text{ and } \mu \models C\}$
  - If P is a SELECT query Q_S then $[\![P]\!]^D_G = \text{ans}(Q_S)$.

Definition 1 (Answer for a query). *Let $Q = (R, F, P)$ be a query, D be the dataset obtained from F, and G be the default graph of D. The* answer *to Q, denoted* ans(Q)*, is defined as follows:*

- *if R is* SELECT W *then* $\text{ans}(Q) = \{\text{result}(R, \mu) \mid \mu \in [\![P]\!]^D_G\}$.
- *if R is* CONSTRUCT H *and* blank(H) *is the set of blank nodes appearing in H, then* $\text{ans}(Q) = \{\beta_i(\text{result}(R, \mu_i)) \mid \mu_i \in [\![P]\!]^D_G\}$ *where* $\beta_i : \text{blank}(H) \to (B \setminus \text{blank}(H)$ *is a blank renaming function satisfying that for each pair of mappings* $\mu_j, \mu_k \in [\![P]\!]^D_G$, $\text{range}(\beta_j) \cap \text{range}(\beta_k) = \emptyset$.
- *if R is* ASK *then* $\text{ans}(Q) = \text{false}$ *when* $[\![P]\!]^D_G = \emptyset$ *(i.e., there exists no mapping* $\mu \in [\![P]\!]^D_G$*) and* $\text{ans}(Q) = \text{true}$ *otherwise.*

The semantics for correlated queries as defined above follows the *nested iteration method* [8], i.e., the inner query is performed once for each solution of the outer query (it is because the results of the inner query are correlated with each individual solution of the outer query). This procedure is attained by replacing variables in the inner query with the corresponding values given by the current mapping of the outer query (e.g., by applying $\mu(Q_{?X})$). For example, consider the graph pattern

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

The method establishes that the graph pattern $(?Y$ email $?E))$ (i.e., the subquery) is evaluated over and over again, once for each result mapping of the OPTIONAL graph pattern (i.e., the outer query).

We have identified two issued related to the use of correlated variables. First, loss of correlation due to unbounded variables. Consider that P is the the OPTIONAL graph pattern in the above example. If μ is a mapping in $[\![P]\!]$ such that $\mu(?X) = a$, $\mu(?N) = b$ and $\mu(?Y)$ is unbounded (i.e., there was no solution for the OPTIONAL part), then there is no value to replace the variable $?Y$ in the inner query, and consequently there exists no correlation. This loss of correlation results in an undesirable evaluation because, when the inner query has at least one solution, the filter condition is true and the mapping is accepted as a solution. Clearly, it is not what the query intuitively means such that the evaluation of the inner graph pattern depends directly on the evaluation of the outer graph pattern. This problem, produced by correlated variables that could be evaluated to unbounded, is intrinsic to the language because the semantics of the UNION and OPTIONAL operators (i.e., they can generate unbounded variables). Hence, we restrict our study by avoiding graph patterns of this type.

Another issue concerns the use of correlated variables in the projection part of a nested SELECT query. Consider the graph pattern

$$((?X \text{ p } ?Y)\,\text{FILTER } ?Y = \text{SOME } (\text{SELECT } ?Y \,\text{WHERE } (?Z \text{ q } ?Y))).$$

Note that, the use of variable $?Y$ as a projected-variable in the nested SELECT query, generates ambiguity about its scope. In fact, it is not clear whether $?Y$ must be considered local to the inner query or it occurs as correlated with the outer query. To minimize the possibility of confusion, the scope of a variable will be interpreted using the nearest result query form possible (i.e., the nearest SELECT). Hence, variable $?Y$ is local in the inner query of the example.

3 Examples of nested queries in SPARQL

Let G_1, G_2 be two RDF graphs identified by IRIs `foaf` and `bib` respectively. G_1 contains personal information using the FOAF vocabulary [7]. G_2 contains bibliographic information using the bibTex Vocabulary [8]. Consider the following examples of nested queries.

[7] `http://xmlns.com/foaf/spec/`

[8] `http://zeitkunst.org/bibtex/0.1/`

Example 1. The oldest people.

```
SELECT ?Per1 FROM foaf
WHERE ((?Per1 foaf:age ?Age1)
          FILTER (¬(?Age1 < SOME ( SELECT ?Age2 FROM foaf
                                    WHERE (?Per2 foaf:age ?Age2)))))
```

Example 2. The youngest people.

```
SELECT ?Per1 FROM foaf
WHERE ((?Per1 foaf:age ?Age1)
          FILTER ( ?Age1 ≤ ALL ( SELECT ?Age2 FROM foaf
                                  WHERE (?Per2 foaf:age ?Age2))))
```

Example 3. Mails of people being part of at least one group.

```
SELECT ?Mail FROM foaf
WHERE ((?Per foaf:mbox ?Mail)
          FILTER (?Per IN ( SELECT ?Mem FROM foaf
                             WHERE (?Mem foaf:member ?Group))))
```

Example 4. Mails of people having at least one publication.

```
SELECT ?Mail FROM foaf
WHERE ((?Per foaf:mbox ?Mail)
          FILTER ( EXISTS (ASK FROM bib
                             WHERE (?Art bib:has-author ?Per)))))
```

The above examples deserve several comments. IN expressions are less expressive than SOME expressions because the former are restricted to equality of values, whereas the latter allows all scalar comparison operators. Nested queries with SOME / ALL operators without correlated variables are better for query composition, i.e., simple and direct copy/paste of queries. The use of EXISTS is not adequate for distributed queries 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).

4 Equivalences among nested queries

In this section we present transformations among nested queries. We will show that all types of nested queries can be simulated by filter conditions with the EXISTS operator. Several equivalences presented in this section are well know in SQL [3].

4.1 Normalization

In order to simplify the transformations, we will avoid complex filter constraints, i.e., expressions of the form $C_1 \wedge C_2$ and $C_1 \vee C_2$ where C_1 and C_2 are filter constraints. This assumption is supported by the following lemma.

Lemma 1. *Every graph pattern having complex filter constraints can be transformed in a graph pattern without complex filter constraints* [9].

Proof. Let P be a graph pattern and C, C_1, C_2 be filter constraints. The lemma is supported by the following equivalences:

$$(P\,\mathrm{FILTER}(C_1 \wedge C_2)) \equiv ((P\,\mathrm{FILTER}\,C_1)\,\mathrm{FILTER}\,C_2) \tag{1}$$

$$(P\,\mathrm{FILTER}(C_1 \vee C_2)) \equiv ((P\,\mathrm{FILTER}\,C_1)\,\mathrm{UNION}(P\,\mathrm{FILTER}\,C_2)) \tag{2}$$

$$(P\,\mathrm{FILTER}(\neg C)) \equiv (P\,\mathrm{MINUS}(P\,\mathrm{FILTER}\,C)) \tag{3}$$

Is not hard to see that the equivalences holds.

4.2 Transformations

Consider the following definition of query equivalence.

Definition 2 (Equivalence of queries). *Two graph patterns P_1 and P_2 are* equivalent*, denoted $P_1 \equiv P_2$, if and only if $[\![P_1]\!]^D_G = [\![P_2]\!]^D_G$ for every RDF dataset D with active graph G. Additionally, given two queries $Q = (R, F, P)$ and $Q' = (R, F, P')$, we say that Q and Q' are equivalent, denoted $Q \equiv Q'$, if and only if $P \equiv P'$.*

Next we will define transformations among several types of nested queries. Based on transformations defined in Section 4.1, we assume that queries do not contain complex filter constraints.

Proposition 1 (Transforming IN queries). *Let P be a pattern of the form $(P_1\,\mathrm{FILTER}(u\ \mathrm{IN}\ \{Q_2\}))$ where $u \in T$. Then, P is equivalent to expression:*

$$(P_1\,\mathrm{FILTER}(u = \mathrm{SOME}\{Q_2\})) \tag{4}$$

Proposition 2 (Transforming SOME queries). *Let P be a pattern of the form $(P_1\,\mathrm{FILTER}(u\ \theta\ \mathrm{SOME}(Q_2)))$ where $u \in T$, $Q_2 = (\mathrm{SELECT}\ ?X_2, F_2, P_2)$, and $\hat{\theta}$ is the inverse operator to θ. Then, P is equivalent to the following expressions:*

$$(P_1\,\mathrm{FILTER}(\neg(u\ \hat{\theta}\ \mathrm{ALL}(Q_2)))) \tag{5}$$

$$(P_1\,\mathrm{FILTER}\,\mathrm{EXISTS}(\mathrm{ASK}, F_2, (P_2\,\mathrm{FILTER}(u\ \theta\ ?X_2)))) \tag{6}$$

Proposition 3 (Transforming ALL queries). *Let P be a pattern of the form $(P_1\,\mathrm{FILTER}(u\ \theta\ \mathrm{ALL}(Q_2)))$ where $u \in T$, $Q_2 = (\mathrm{SELECT}\ ?X_2, F_2, P_2)$, and $\hat{\theta}$ is the inverse operator to θ. Then, P is equivalent to the following expressions:*

$$(P_1\,\mathrm{FILTER}(\neg(u\ \hat{\theta}\ \mathrm{SOME}(Q_2)))) \tag{7}$$

$$(P_1\,\mathrm{FILTER}(\neg\,\mathrm{EXISTS}(\mathrm{ASK}, F_2, (P_2\,\mathrm{FILTER}(u\ \hat{\theta}\ ?X_2))))) \tag{8}$$

[9] Lemma 1 is true under set semantics. The inclusion of bag semantics, as defined for SPARQL, introduces complexity issues which are not discussed here.

For example, the following queries show the application of transformations (7) and (8) to the query of Example 2.

Example 5. The youngest people (using the SOME operator).

```
SELECT ?Per1 FROM foaf
WHERE ((?Per1 foaf:age ?Age1)
            FILTER ( ¬(?Age1 > SOME (SELECT ?Age2 FROM foaf
                                      WHERE (?Per1 foaf:age ?Age2)))))
```

Example 6. The youngest people (using the EXISTS operator).

```
SELECT ?Per1 FROM foaf
WHERE ((?Per1 foaf:age ?Age1)
            FILTER (¬ EXISTS (ASK FROM foaf
                                  WHERE ((?Per2 foaf:age ?Age2)
                                            FILTER (?Age1 > ?Age2)))))))
```

From the transformations defined above we can present the following result.

Theorem 1. *Nested queries using SOME, ALL and IN can be simulated by using nested queries with the EXISTS operator.*

5 Conclusions

We have studied how to extend SPARQL to support nesting along the design philosophy of SQL. We showed that there is a simple syntax and semantics for such extensions in SPARQL. We have shown that incorporating ASK queries in FILTERS (through the EXISTS operator) gives the full power and flexibility of SQL nesting, allowing additionally to extend the semantics of SPARQL in a clean and modular form. The proposal presented here permits a simple and direct implementation of nested queries as known in the relational world (and hence by known translation results) in the SPARQL world.

Future work. An interesting problem studied in the database literature is the efficient implementation of nested queries, where a well known approach is the development of algorithms which transform nested queries into equivalent non-nested queries which can be processed more efficiently by query-processing subsystems [8,5]. On this line, most results are concentrated on aggregate subqueries; optimization of non-aggregate subqueries has some limitations, specially for queries with multiple subqueries and null values [2].

Although decorrelation often results in cheaper non-nested plans, decorrelation is not always applicable, and even if applicable may not be the best choice in all situations since decorrelation carries a materialization overhead [15,6]. In this direction, the issue of efficient methods of processing nested queries is one of the main problems to be addressed in future works on this topic.

Acknowledgments. C. Gutierrez was supported by FONDECYT projects No. 1070348 and No. 1090565. The authors wish to thank the reviewers for their comments.

References

1. R. Angles and C. Gutierrez. The Expressive Power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, number 5318 in LNCS, pages 114–129, 2008.
2. B. Cao and A. Badia. A nested relational approach to processing SQL subqueries. In *Proc. of the 2005 ACM SIGMOD international conference on Management of data*, pages 191–202, New York, NY, USA, 2005. ACM Press.
3. S. Ceri and G. Gottlob. Translating SQL into relational algebra: optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985.
4. R. Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Labs, 2005.
5. R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 23–33, New York, NY, USA, 1987. ACM Press.
6. R. Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In *Proc. of the 31st Int. Conf. on Very large Data Bases (VLDB)*, pages 481–492. VLDB Endowment, 2005.
7. S. Harris and A. Seaborne. SPARQL 1.1 Query. W3C Working Draft. http://www.w3.org/TR/2009/WD-sparql11-query-20091022/, October 22 2009.
8. W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.
9. K. Kjernsmo and A. Passant. SPARQL New Features and Rationale. W3C Working Draft. http://www.w3.org/TR/2009/WD-sparql-features-20090702/, July 2 2009.
10. G. Klyne and J. Carroll. Resource Description Framework (RDF) Concepts and Abstract Syntax. http://www.w3.org/TR/2004/REC-115-concepts-20040210/, February 2004.
11. 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.
12. A. Polleres. From SPARQL to Rules (and back). In *Proceedings of the 16th International World Wide Web Conference (WWW)*, pages 787–796. ACM, 2007.
13. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation 15 January. http://www.w3.org/TR/2008/REC-115-sparql-query-20080115/, 2008.
14. S. Schenk. A SPARQL Semantics Based on Datalog. In *30th Annual German Conference on Advances in Artificial Intelligence (KI)*, volume 4667 of *LNCS*, pages 160–174. Springer, 2007.
15. P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proc. of the 12th Int. Conf. on Data Engineering (ICDE)*, pages 450–458. IEEE Computer Society, 1996.
16. P. Weinberg, J. Groff, and A. Oppel. *SQL, The Complete Reference.* McGraw-Hill, 2010.