International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:3, 2015

A Framework for Semantics Preserving SPARQL-to-SQL Translation

N. Soussi, M. Bahaj

***Abstract*—**The enormous amount of information stored on the web increases from one day to the next, exposing the web currently faced with the inevitable difficulties of research pertinent information that users really want. The problem today is not limited to expanding the size of the information highways, but to design a system for intelligent search. The vast majority of this information is stored in relational databases, which in turn represent a backend for managing RDF data of the semantic web. This problem has motivated us to write this paper in order to establish an effective approach to support semantic transformation algorithm for SPARQL queries to SQL queries, more precisely SPARQL SELECT queries; by adopting this method, the relational database can be questioned easily with SPARQL queries maintaining the same performance.

***Keywords*—**RDF, Semantic Web, SPARQL, SPARQL Query Transformation, SQL.

I. Introduction

THE Semantic Web is a mesh of information linked up in such a way as to be easily processable by machines, on a global scale [1]. This is an efficient way of representing data on the World Wide Web. It is considered as a gateway to access the data between different applications and systems.

Semantic annotations of the various heterogeneous resources on the web are represented in RDF, it's a standard Framework to annotate and represent web resources via a powerful data model. SPARQL is officially recommended by the W3C as the most representative language of description for RDF data. This language may initially looks like SQL, but in reality there are many important differences between themselves because the data is graph-based so queries match graph patterns instead SQL's relational matching operations [6].

Our motivation to transform SPARQL queries to SQL queries semantically equivalents is justified by the assurance of a dynamic access to the relational database by the semantic web and querying relational database by SPARQL queries. It facilitates the interoperability of these two heterogeneous systems without physical transformation of data, which contributes to achieve the expected purpose of the Semantic Web namely the improved access to resources interconnected through the web.

The rest of the paper is organized as follows: Section II exposes examples of existing approaches in the translation of SPARQL queries to SQL queries. Section III presents a SPARQL grammar as well as detailed description of the transformation algorithm. Finally, Section IV concludes this paper with a proposition of the future work.

N. Soussi and M. Bahaj are with the Departement of Mathematics and Computer Science, Faculty of Science and Technologies, Hassan 1st University, LITEN Laboratory, Settat, Morocco (e-mail: nassima.soussi@gmail.com, mohamedbahaj@gmail.com).

II. Related Work

In order to ensure a better transformation of SPARQL queries to SQL queries, several approaches have been made to guarantee the preservation of semantics.

The authors in [4] create a computable mapping of SPARQL semantics to SQL semantics by considering only SPARQL SELECT queries. This mapping is based on the decomposition of Basic Graph Pattern in a set of triple patterns conjunctions each of which corresponds to the attributes of a given relationship in the database. The paper [2] presents a formalization of relational algebra based semantics of SPARQL so as to reduce the gap between the Semantic Web and relational database. Based on this result, they have established a generic algorithm for semantic translation of SPARQL to SQL. The authors propose in [5] a translation algorithm of SPARQL to SQL called IC-based; this transformation is based on Integrity Constraint. With the different structure of the tables in the relational databases, the SPARQL query statements are divided into four types which each one has its own translations. The work described in [3] proposes a translation method of SPARQL to SQL that involves the translation of SPARQL queries to a datalog program using the mapping R2RML; A relational algebra is generated from this datalog program in order to design the SQL query by translating relational operators to the corresponding SQL operators. The authors in [7] provide a conversion for SPARQL to SQL described by a converter tool that receives a SPARQL query which is then parsed using the parsing function of Jena Framework and analyzed so as to build a new data structure in order to produce an equivalent SQL query. Aiming to a seamless integration of SPARQL queries with SQL queries, the paper [9] proposes a method that translates a complete SPARQL query into a single SQL that can be directly used as a sub-query by other SQL queries. In particular, the authors propose effective schemes to translate filter expressions into SQL statements.

Our main contribution in this active topic extends RETRO method [8] which explores one direction i.e. to look at RDF through Relational lenses. This paper investigates the reverse direction proposing an efficient and helpful transformation algorithm for SPARQL SELECT query to semantically equivalent SQL SELECT query without physically transforming the data; Our results is obtained by dividing the

SPARQL query to several basics elements (group graph patterns, basic graph patterns, triple patterns, optional group graph patterns and union group graph patterns) in order to carefully analyze and extract its semantic equivalents in SQL.

III. Query Mapping

Before starting the description of transformation algorithm for SPARQL queries to SQL queries, we illustrate this with examples through which we describe an abstract grammar for SPARQL [10] used to build derivation trees of the latter. Then we describe all procedures of mapping queries algorithm: *ExtractSQLFROMClause*, *ExtractSQLWHEREClause*, *ExtractSQLSELECTClause*, *QueryMapping*.

A. Examples

In the examples treated subsequently, we consider the RDF dataset illustrated in Fig. 1 with the schema S:

S = {name(s,o), age(s,o), job(s,o), contry(s,o), website(s,o)}

Table I describes a set of queries using SPARQL and SQL.

TABLE I
Queries Examples Using SQL And SPARQL

Description	SPARQL Query	SQL Query
Cross Product of relations job and contry	SELECT ?s1 ?o1 ?s2 ?o2 WHERE { ?s1 job ?o1 ?s2 contry ?o2 }	SELECT job.s, job.o, contry.s, contry.o FROM job, contry
Find the website of bob	SELECT ?o2 WHERE { ?s1 name ?o1 ?s1 website ?o2 FILTER (?o1 = bob) }	SELECT website.o FROM name, website WHERE name.s = website.s AND name.o = bob
Names of all people with age <30 and job=teacher	SELECT DISTINCT ?s1 ?o1 WHERE { { ?s1 name ?o1 ?s1 age ?o2 FILTER (?o2 < 30) } { ?s1 name ?o1 ?s1 job ?o3 FILTER (job = teacher) } }	SELECT name.s, name.o FROM name, age WHERE age.o < 30 AND name.s = age.s INTERSECT SELECT name.s, name.o FROM name, job WHERE job = teacher AND name.s = job.s
Names of all people who either have a website id or job = web programmer	SELECT ?o1 WHERE { { ?s1 name ?o1 ?s1 website ?o2 } UNION { ?s1 name ?o1 ?s1 job ?o2 FILTER (?o2= webProgrammer) } }	SELECT name.o FROM name, website WHERE name.s = website.s UNION SELECT name.o FROM name, job WHERE name.s = job.s AND job.o = webProgrammer
Find people who work as teachers and not web programmers	SELECT ?s1 WHERE { ?s1 job ?o1 FILTER (?o1 = teacher) } OPTIONAL { ?s1 job ?o2 FILTER (?o1 = webProgrammer) } }	SELECT job.s FROM job WHERE job.o = teacher EXCEPT SELECT job.s FROM job WHERE job.o = webProgrammer

(B1, name, paolo) (B2, name, bob)
(B3, name, adamo) (B4, name, gad)
(B1, age, 34) (B2, age, 29)
(B3, age, 41) (B4, age, 36)
(B2, job, teacher) (B3, job, teacher)
(B3, job, webProgrammer) (B4, job, webProgrammer)
(B1, country, USA) (B2, country, England)
(B3, country, France) (B2, country, Canada)
(B2, website, www.bob.com) (B4, website, www.gad.com)

Fig. 1 The RDF Dataset

B. Abstract Grammar for SPARQL

Let GGP, BGP, Var, VarList, VarName, TP and Op denote group graph pattern, basic graph pattern, variable, variables list, variable name, triple pattern and operator respectively.

Based on the previous examples we can deduce an abstract SPARQL grammar as follows:

SELECTQuery → SimpleQuery | FilterQuery | UnionGGPQuery | SetGGPQuery | OptinalGGPQuery
SimpleQuery → SelectClause 'WHERE{' BGP '}'
FilterQuery → SelectClause 'WHERE{' BGP FILTERClause '}'
UnionGGPQuery → SelectClause 'WHERE{' GGP 'UNION' GGP '}'
SetGGPQuery → SelectClause 'WHERE{' GGP GGP '}'
OptionalGGPQuery → SelectClause 'WHERE{' GGP 'OPTIONAL' GGP '}'
SelectClause → 'SELECT' VarList
VarList → Var (',' Var)*
Var → '?'Varname
BGP → TP ('.' TP)*
GGP → BGP | BGP FILTERClause
TP → Subject Predicate Object
FILTERClause → 'FILTER' Constraint
Constraint → '(' attribut Op value ')'
Op → '<' | '>' | '=' | '>=' | '<=' | '<>'

C. Transformation Algorithm

Firstly we dissected and analyzed SPARQL queries by splitting them into two parts: the first part represented by the SELECT clause, it allows to extract the set of attributes constituting the SQL query equivalent via the sub-procedure *ExtractSQLSELECTClause*, the second part contains the WHERE clause that allows to extract relations names from BGP (set of triple patterns) to design the FROM clause of SQL query using the equivalent sub-procedure *ExtractSQLFROMClause* and also joins conditions and booleans conditions constituting the WHERE clause of the SQL query equivalent using the sub-procedure *ExtractSQLWHEREClause*. All these sub-procedures are used in the main procedure *QueryMapping* which takes as input an SPARQL query so as to return at the end an SQL query semantically equivalent.

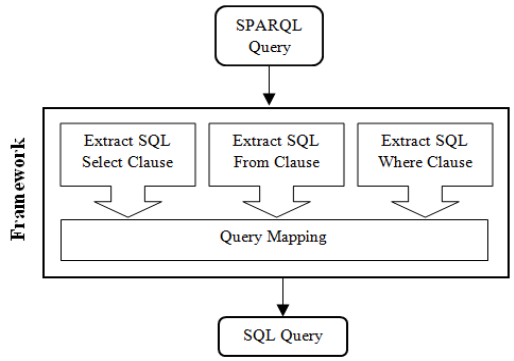

Fig. 2 Our contribution in SPARQL-to-SQL translation

1. ExtractSQLFROMClause Procedure

As regards the procedure *ExtractSQLFROMClause*, it takes as input the set of triple patterns, BGP, that glance through by extracting for each of them its property which represents the name for the SQL relationship and then add it to the variable 'FROM' initially empty. At the end, this procedure returns the FROM clause of the SQL query equivalent.

```
Input: A set of triples patterns, BGP
Output: SQL FROM Clause

   FROM = "" {A SQL FROM Clause initially blank}
R = ø          {A map of relations is initially blank}
   for i ← 1 to BGP.size do
        R.add(BGP[i].predicate)
      FROM += BGP[i].predicate
      if i < BGP.size then
          FROM += ","
      end if
end for
```

Fig. 3 ExtractSQLFROMClause Algorithm

2. ExtractSQLWHEREClause Procedure

The sub-procedure *ExtractSQLWHEREClause* takes as input the set of SQL relations, R, the map of triple patterns, BGP and the FILTER clause constraints, FilterConstraint. This sub-procedure begins by extracting the joins conditions (Attribute Op Attribute), JC, glancing through the map BGP and compare each triple pattern of rank i with the triple pattern of rank j equivalent to i+1. During this process, the algorithm examines cases below to add the joints conditions extracted to the WHERE variable initially empty and separate them with logical AND operator.

- If the subject of triple pattern i equal to the subject of triple pattern j then the join condition is written in the following form:

$$R[i].s = R[j].s$$

- If the subject of triple pattern i equal to the object of triple pattern j then:

$$R[i].s = R[j].o$$

- If the object of triple pattern i equal to the subject of triple pattern j then:

$$R[i].o = R[j].s$$

This sub-procedure also allows extracting booleans conditions (Attribute Op Value) operating on the SPARQL FILTER clause; we cut the constraint of the FILTER clause to extract the attribute, operator and value, and define the attribute type to formulate the boolean condition.

```
Input: R, BGP, FilterConstraint
Output: SQL WHERE Clause

WHERE = ""{A SQL WHERE Clause initially blank}
JC =  ø
for i ←1 to BGP.size do
   for j ← i+1 to BGP.size do
      if  BGP[i].subject = BGP[j].subject  then
         jc = R[i] + ".s =" + R[j] + ".s";     JC.add(jc);
      else if  BGP[i].subject = BGP[j].object  then
         jc = R[i] + ".s =" + R[j] + ".o";    JC.add(jc);
      else if  BGP[i].object = BGP[j].subject  then
         jc = R[i] + ".o =" + R[j] + ".s";    JC.add(jc);
      end if
   end for
end for
for i ← 1 to JC.size do
   WHERE += JC[i]
      if  i < JC.size then
         WHERE += " AND "
      end if
end for
if FilterConstraint.isEmpty() = false do
   if WHERE != "" then
      WHERE += " AND "
   end if
   f1 = FilterConstraint.loperand;
   f2  = FilterConstraint.operator;
   f3 =  FilterConstraint.roperand;
   tp ← BGP.get(f1) { get the TP of the f1 attribute }
   attr_relation = tp.predicate;    attr_name  = f1.getName()
   if (attr_name = "s") then
      WHERE += attr_relation + ".s" + f2  + f3
   else if (attr_name = "o") then
      WHERE += attr_relation + ".o" + f2  + f3
   end if
end if
```

Fig. 4 ExtractSQLWHEREClause Algorithm

3. ExtractSQLSELECTClause Procedure

The sub-procedure *ExtractSQLSELECTClause* can extract SQL SELECT clause from the SPARQL SELECT clause; it glances through the set of variables in the SPARQL SELECT clause to verify the type of each (subject or object) in order to conceive its equivalent in SQL language and add it to the SELECT variable which is initially blank.

```
Input: SPARQL SELECT Clause V, BGP
Output: SQL SELECT Clause
SELECT = “”{A SQL SELECT Clause initially blank}
for i ← 1 to V.size do
   tp←BGP.get(V[i])
   if (V[i].name = “s”) then
      SELECT += tp.predicate + “.s”
   else if (V[i].name = “o”) then
      SELECT += tp.predicate + “.o”
   end if
   if (i<V.size) then
      SELECT += “,”
   end if
end for
```

Fig. 5 ExtractSQLSELECTClause Algorithm

```
Input: SPARQL query, qin
Output: SQL query, qout
qout = “”{A SPARQL Query that is initially blank}
tree = parse(qin)
qin^SELECT = tree.getSelectClause()
qout^SELECT = “”
qout^FROM = “”
qout^WHERE = “”
if tree.type = SimpleQuery then
   qin^WHERE = tree.getBGP()
   qout^SELECT +=ExtractSQLSELECTClause(qin^SELECT, qin^WHERE)
   qout^FROM += ExtractSQLFROMClause(qin^WHERE)
   qout^WHERE += ExtractSQLWHEREClause(R, BGP, null)  if qout^WHERE.isEmpty() then
   qout += qout^SELECT + qout^FROM
   else
   qout += qout^SELECT + qout^FROM + qout^WHERE
   end if
else if tree.type = FilterQuery then
   qin^WHERE_BGP = tree.getBGP()
   qin^WHERE_Filter = tree.getFilterClause()
   qout^SELECT +=ExtractSQLSELECTClause(qin^SELECT,qin^WHERE_BGP)
   qout^FROM += ExtractSQLFROMClause(qin^WHERE_BGP)
   qout^WHERE+=ExtractSQLSELECTClause(qin^SELECT, qin^WHERE_BGP, qin^WHERE_Filter)
   qout += qout^SELECT + qout^FROM + qout^WHERE
else if tree.type = UnionGGPQuery then
   lGGP = tree.getLeftGGP()
   rGGP = tree.getRightGGP()
   qin^1 = “SELECT” + qin^SELECT + “WHERE” + lGGP
   qin^2 = “SELECT” + qin^SELECT + “WHERE” + rGGP
   qout+=QueryMapping(qin^1)+“UNION”+QueryMapping(qin^1)
else if tree.type = SetGGPQuery then
   fGGP = tree.getFirstGGP()
   sGGP = tree.getSecondGGP()
   qin^1 = “SELECT” + qin^SELECT + “WHERE” + fGGP
   qin^2 = “SELECT” + qin^SELECT + “WHERE” + sGGP
   qout+=QueryMapping(qin^1)+“INTERSECT”+QueryMapping(qin^2)
else if tree.type = OptionalGGPQuery then
   lGGP = tree.getLeftGGP()
   rGGP = tree.getRightGGP()
   qin^1 = “SELECT” + qin^SELECT + “WHERE” + lGGP
   qin^2 = “SELECT” + qin^SELECT + “WHERE” + rGGP
   qout+=QueryMapping(qin^1)+“EXCEPT” +QueryMapping(qin^2)
end if
```

Fig. 6 QueryMapping Algorithm

4. QueryMapping Procedure

The main procedure QueryMapping takes as input an SPARQL query and returns an equivalent SQL query. A conversion tree of SPARQL query is generated by using the parse function. If the query type is 'Simple Query' then the conversion tree generates SPARQL SELECT clause and WHERE clause that contains only a set of triple patterns BGP; Then the sub-procedures *ExtractSQLSELECTClause* and *ExtractSQLFROMClause* are called as well as *ExtractSQLWHEREClause* with FILTER constraint null so as to conceive the equivalent SQL query via concatenating the results of previous functions. We proceed in the same manner if the SPARQL query type is 'Filter Query' except that the WHERE clause contains in addition to BGP, the FILTER clause that leads to the consideration of FILTER constraint in the sub-procedure *ExtractSQLWHEREClause* in order to design the output SQL query. In cases where the type of the SPARQL query is UnionGGPQuery, SetGGPQuery or OptionalGGPQuery, the conversion tree generates the SELECT clause and both GGP encapsulated in the WHERE clause so as to design with each of them an SPARQL sub-query that will be used in the recursive procedure QueryMapping in order to have at the end an SQL query semantically equivalent.

IV. Conclusion

In summary, the main contribution of this paper in the interoperability between RDF Stores and RDBMS as discussed above is the elaboration of a translation Framework for SPARQL SELECT query to SQL query by providing a direct mapping algorithm that decomposes the SPARQL query in order to interpret it and deduce its equivalent in the semantic of SQL language.

One obvious extension of our research regarding SPARQL to SQL converter will be an improvement of the SPARQL grammar so as to enhance and reinforce our algorithm to support more queries types (construct, describe and ask queries). Another promising direction for future work is to provide a Framework for SPARQL to XQuery translation in order to bridge the gap between semantic web and XML world.

References

[1] Http://infomesh.net/2001/swintro/.
[2] A. Chebotko, L. Shiyong, and F. Fotouhi, “Semantics preserving sparql-to-sql translation”, Data Knowl. Eng, 68(10):973–1000, October 2009.
[3] M. Rodriguez-Muro, M. Rezk, J. Hardi, M. Slusnys, T. Bagosi and D. Calvanese, “Efficient SPARQL-to-SQL translation using R2RML Mapping”, KRDB Research Centre, Free University of Bozen-Bolzano, 2013.
[4] E. Prud'hommeaux, and A. Bertails, “A Mapping of SPARQL Onto Conventional SQL”, World Wide Web Consortium (W3C), 2008.
[5] X. Cui, D. Ouyang, Y. Ye, X. Wang., “Translation of Sparql to SQL Based on Integrity Constraint”, Journal of Computational Information Systems 7:2 394-402, 2011.
[6] “A Comprehensive Comparative study of SPARQL and SQL”, Vipin Kumar. N, Archana P. Kumar, Kumar Abhishek. (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (4), 2011, 1706-1710.

[7] K. Bajda-Pawlikowski, "Querying RDF data stored in DBMS: SPARQL to SQL Conversion", Technical Report TR-1409, Yale Computer Science Department, USA.
[8] J. Rachapalli, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham, "RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation", The University of Texas at Dallas, 800 West Campbell Road, Richardson, TX 75080-3021, USA, 2009.
[9] J. Lu, F. Cao, L. Ma, Y. Yu, and Y. Pan, "An Effective SPARQL Support over Relational Databases", Semantic Web Ontologies and Databases, pp 57-76, 2007.
[10] http://www.w3.org/TR/rdf-sparql-query/#sparqlGrammar.

N. Soussi was born in 1991, in Khouribga, Morocco. She got her special higher studies degree in software engineering from National School of Applied Sciences in 2014. She is now a Phd student in the Department of Mathematics and computer sciences, Faculty of Sciences & Technology of Settat, Hassan 1st University, Settat, Morocco. Her area of interest includes web ontologies and semantic web.

M. Bahaj was born in 1964, in Ouezzane, Morocco. He got his PhD in Applied Mathematics, from University of Pau, France, in 1993. He is now a full Professor in Department of Mathematics and Computer Sciences from the University Hassan 1st Faculty of Sciences & Technology Settat Morocco. He is co-chairs of International Conference on Software Engineering, Databases and Expert Systems (SEDEXS'12), NASCASE'11. He has published over 60 peer-reviewed papers. His research interests are intelligent systems, Ontologies Engineering, Partial and differential equations, Numerical Analysis and scientific computing.