Query Migration from Object Oriented World to Semantic World

SOUSSI Nassima, BAHAJ Mohamed

Department of Mathematics & Computer Science, Faculty of Science and Technologies, Hassan 1st University, Settat, Morocco

Abstract — In the last decades, object-oriented approach was able to take a large share of databases market aiming to design and implement structured and reusable software through the composition of independent elements in order to have programs with a high performance. On the other hand, the mass of information stored in the web is increasing day after day with a vertiginous speed, exposing the currently web faced with the problem of creating a bridge so as to facilitate access to data between different applications and systems as well as to look for relevant and exact information wished by users. In addition, all existing approach of rewriting object oriented languages to SPARQL language rely on models transformation process to guarantee this mapping. All the previous reasons has prompted us to write this paper in order to bridge an important gap between these two heterogeneous worlds (object oriented and semantic web world) by proposing the first provably semantics preserving OQL-to-SPARQL translation algorithm for each element of OQL Query (SELECT clause, FROM clause, FILTER constraint, implicit/explicit join and union/intersection SELECT queries).

Keywords — OQL, SPARQL, Semantic Web, Object, OQL To SPARQL.

I. Introduction

The Semantic Web [1] is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation; it’s based on the standards and protocols of the current web (http, URI and XML) and its own standards: The Resource Description Framework RDF [3] dedicated to describe data, the Web Ontology Language OWL [2] for creating structured ontology and the query language SPARQL [4] for querying data from RDF graphs.

Currently, the majority of information systems for companies databases adopt the object-oriented approach regarded as the best data organization paradigm providing the ability to represent complex entities and implement structured software with very high performance, which makes the development of methods and tools for automatic mapping from object oriented world to semantic world a very relevant need. These reasons motivated us to work on this topic so as to elaborate a first conversion query algorithm of OQL to SPARQL that translate each component of OQL SELECT query to its equivalent in SPARQL language.

II. Related Works

Recently, several researches focus on the mapping of data, models, concepts, and queries from the existing data source content to semantic web world. The majority of these researches are interested much more to the relational systems than others; several approaches have been proposed about this mapping direction, such as: RETRO [6] that choose not to physically transform the data but to derive a domain specific relational schema from RDF data and its query mapping transforms an SQL query over the schema into an equivalent SPARQL query executable upon the RDF store. R2RML [7,8] a language for expressing customized mappings from relational databases to RDF datasets presented recently with a novel version which provides a user interface to create and edit mappings interactively even for non-experts. D2RQ/Update [5] is an extension of D2RQ [9] to enable executing SPARQL/Update statements on the mapped data, and to facilitate the creation of a read-write Semantic Web.

Regarding the object-oriented data source, the SPOON approach (Sparql to Object Oriented eNgine) described in [11] propose an automatic mapping between the object-oriented model (ODL) and the correspondent one at the ontological level in order to build a SPARQL endpoint. The paper [12] aims to address query rewriting by means of model transformations. In fact, it allows querying RDF data sources via an object oriented query which is automatically rewritten in SPARQL in order to access RDF data, it also translate SPARQL queries into object oriented queries so as to implement SPARQL endpoints for object oriented applications.

These studies did not propose any query translation solution for rewriting each element of Object Oriented queries into SPARQL queries semantically equivalent but they rely on models transformation process to guarantee this mapping.

III. Query Language Metamodel & Examples

In this section, we describe languages used by our translation approach from object oriented world to semantic web world in order to represent each language with its own metamodel developed from their grammars [14] [15]: the Object Query Language (OQL) for object-oriented databases and a query language for RDF data (SPARQL).

A. OQL Metamodel

The OQL is an object-oriented query language in the Object Data Management Group standard named ODMG; this language provides an easy access to an object databases. Like SQL, the SELECT query which runs on relational tables works with the same syntax and semantics on collections of ODMG objects, which leads to search for an instance of an object rather than looking for a row of data. Several implementations of this standard exist; we quote as examples: HQL [16], JPQL [17], and others.

The metamodel schematized below is limited to SELECT Query in its simple and compound form (Intersect and Union SELECT query). The fig. 1 represents the OQL query of such a type that is composed of five clauses: SelectFromClause, WhereClause, GroupByClause, OrderByClause and HavingClause.
The SelectFromClause representation is given in Fig. 2. This clause is composed of an optional SelectClause (we can omit the SELECT clause in some implementation of OQL language such as HQL) and a mandatory FromClause. A SelectClause contains a PropertyList composed of a list of values or objects resulting from the query; these properties are described as a path that permits to browse the object model. The FromClause allows selecting properties from the object model. This clause is composed of a mandatory ClassReference and an optional ClassJoined; the ClassReference indicates the class name ClassNameDeclaration or collection name CollectionNameDeclaration of selected objects whereas the ClassJoined indicates the set of classes which we want to join.

Fig. 3 describes the WhereClause expression that represents the constraint part of the query. It can be a binary expression (and, or) or an operator expression (\(<\),\(\le\), \(>\),\(\ge\),\(=\)) containing an attribute path and a value.

B. SPARQL Metamodel

The SPARQL is an RDF query language, that is, a semantic query language for databases, able to retrieve and manipulate data stored in Resource Description Framework (RDF) format [13]. The Fig. 4 schematizes the SPARQL metamodel presented the different types for queries. In this paper, we are only interested by SelectQuery.

The SelectQuery as presented in the Fig. 5 is composed of the SelectClause identifies the variables to appear in the results, and the WhereClause consists, in its turn, of GroupGraphPattern represents a set of GraphPattern identifying a various kinds of graph pattern: (a) FilterPattern: used to filter a set of objects using a various criteria and requirements. The filter expressions can be combined through the logical operations so as to form more complex filter, (b) TripleSameSubject: includes a subject and associated properties, (c) UnionGraphPattern: union of patterns, (d) OptionalGraphPattern: optional patterns.

C. Examples

In the examples illustrated in Table I, we consider the two classes quoted bellow that list the Person class having as attributes: matricule, name, age, degree and addr which represents the declaration of Address class in Person class as an attribute; the Address class having as attributes: id, city and state. The OQL queries listed in this example have the types: Simple query (SELECT FROM clause with/without WHERE clause), implicit and explicit join, Union and intersection SELECT queries.

Class Person{
  attribute string matricule;
  attribute string name;
  attribute integer age;
  attribute string degree;
  relationship Address addr;
}
Class Address {
attribute integer id;
attribute string city;
attribute string state;
}

### TABLE I

<table>
<thead>
<tr>
<th>QUERY Type</th>
<th>OQL</th>
<th>SPARQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit Join</td>
<td>SELECT p.name, a.city FROM Person p, Address a WHERE a.state = “MA”</td>
<td>SELECT ?name WHERE { ?who &lt;Person#name&gt; ?name; &lt;Person#addr&gt; ?addr. ?addr &lt;Address#city&gt; ?city; &lt;Address#state&gt; ?state. Filter ( ?state = “MA” ) }</td>
</tr>
</tbody>
</table>

### IV. QUERY MAPPING ALGORITHM

In this section, we will detail our main contribution by describing all procedures used in our query mapping algorithm: ConstructSparqlSelectClause, ConstructTriplePattern, ConstructFilterExpression, ConstructSparqlWhereClause, MappingOQLtoSPARQL and Merge. The fig. 6 schematizes our approach as follow:

Fig. 6. Representation schema of our mapping approach

**A. ConstructSparqlSelectClause Subprocedure**

The ConstructSparqlSelectClause subprocedure takes as input a set of attributes from an OQL SELECT Attributes (OSA) in order to glance through this set and extract the attributes name and add each one to the SPARQL SELECT clause initially blank which is returned at the end by this procedure.

**Input**: OSA  
**Output**: A SPARQL SELECT Clause  
**Begin**  
select = “”  
for each attribute attr ∈ OSA do  
attrName = getAttrName(attri)  
select += “?” + attrName + “ ”  
end for  
return select  
**End Algorithm**

**B. ConstructTriplePattern Subprocedure**

The ConstructTriplePattern subprocedure takes as input the OQL SELECT Attributes, OSA, Class Reference, CR, Class Joined, CJ and Where Clause Attribute, WCA so as to return at the end a set of Triple Patten of SPARQL equivalent query. Firstly, the algorithm stores the OSA in the set A (initially blank) dedicated to contain all query attributes, then it verifies the existence of join in the query by determining its type if it exists; In fact, the explicit join type is checked if the CJ variable is not null, in this case, the algorithm extract the join condition operand in order to add it to the set A, and next it also extract the ClassReference included in the ClassJoined clause in order to add them to the set CR dedicated to contain all Classes References of the query. Similarly, the implicit join type is checked if the number of elements of the set CR is strictly greater than 1, in this case, the join condition operand is added to the set A. If the query contains a where clause, its attribute is added also to the set A. Before adding attributes to the set A, the algorithm checks firstly if these attributes do not already exist in that list.

After the combination of all the query attributes in the set A and Classes references in the set CR, it glances through the set A for each Class Reference CR in order to extract for each a attribute its name and the alias for its class; if the CR, alias equal to the alias of the class attribute a, then it formulate the triple pattern of equivalent SPARQL query and adds it to the set TP and removing the attribute a, from the list A so as not to reprocess it in the following iterations. The attributes
that do not satisfy the above condition will be stored in a temporary list so as to add them again to the set A and switch to the next reference class and repeat the same process.

Input : OSA, CR, CJ, WCA
Output : A set of Triple Pattern

Begin
A = ø {Set of all query attributes initially blank}
Stack Atemp = EmptyStack
Boolean simpleJoin = False
A.add(OSA)
if (CJ != ø OR CR.size>1) then
  simpleJoin = True
  if (CJ != ø) then //Explicit Join
    JCA = CJ.getJoinCond().loperand()
    CR.add(CJ.getCR())
  else if (CR.size>1) then //Implicit Join
    JCA = CR1.getClassName().getExternalKey()
  end if
  if (A.ExistInList(JCA) == False) then
    A.add(JCA)
  end if
end if
if (WCA!=NULL AND A.ExistInList(WCA)=False) then
  A.add(WCA)
end if
for i ← 1 to CR.size do
  for j ← 1 to A.size do
    attrClassAlias = getClassAlias(a)
    attrName = getAttrName(a)
    if (CR[i].getClassAlias() == attrClassAlias) then
      tp←{?si<CR[i].getClassName()#attrName> ?attrName}
    end if
    if (simpleJoin==True AND TP[i].object == CRi.getClassName().getFK()) then
      ?si+1 = TP[i].object
    end if
    TP.add(tp)
    A.remove(a)
  end if
  Atemp.add(a)
end if
while (Atemp.size>0) do
  A.add(Atemp.remove())
end while
return TP
End Algorithm

C. ConstructFilterExpression Subprocedure

The ConstructFilterExpression subprocedure takes as input an OQL where condition, WC, so as to extract the left operand, operator and right operand, and formulate at the end the FILTER clause expression of the SPARQL equivalent query.

Input : An OQL where condition, WC
Output : A Filter Expression

Begin
filterExp = "" {A Filter Expression that is initially blank}
if (WC != NULL) then
  leftOp = WC.lOperand
  Op = WC.Operand
  rightOp = WC.rOperand
  attrName = getAttrName(leftOp)
  filterExp = "FILTER("+ ?attrName +Op+ rightOp +")"
end if
return filterExp
End Algorithm

D. ConstructSparqlWhereClause Subprocedure

The ConstructSparqlWhereClause subprocedure takes as input the set of triple pattern TP returned by the ConstructTriplePattern Subprocedure and the Filter Expression FilterExp returned by the ConstructFilterExpression Subprocedure. This algorithm glances through the set of TP to concatenate the triple patterns in order to formulate the SPARQL WHERE clause equivalent. In the case where the two triple patterns have the same subject, the second one will be reduced by removing its subject and adding a comma after the first triple pattern.

Input : TP, FilterExp
Output : A SPARQL WHERE Clause

Begin
where = "" {A SPARQL Where Clause that is initially blank}
for i←1 to TP.size then
  if (i>1 AND TP[i].subject == TP[i-1].subject) then
    TP[i] ← {TP[i].predicate TP[i].object}
  else
    where += TP[i]
  end if
end for
if (isEmpty(FilterExp) == False) then
  where += FilterExp
end if
return where
End Algorithm

E. MappingOQLtoSPARQL Procedure

The MappingOQLtoSPARQL is the main procedure of our algorithm; it takes as input the OQL SELECT query, qin so as to return at the end the SPARQL equivalent query, qout. A conversion tree of OQL query is generated by using the parse function. If the query type is “SimpleQuery”, the conversion tree generates SPARQL SELECT clause, FROM clause contained classes references and WHERE clause if it exists, then the set of triple patterns is constructed from the ConstructTriplePattern, and the FILTER expression from ConstructFilterExpression qualifying as inputs for the ConstructSparqlWhereClause generated the SPARQL
WHERE clause. The SPARQL SELECT clause is generated from
\textit{ConstructSparqlSelectClause}; the results of previous Subprocedures
are concatenated so as to formulate the SPARQL equivalent query. We
proceed with the same manner if the OQL query type is \textit{JoinQuery} except
that the OQL conversion tree will generates the \textit{ClassedJoined} in
addition to \textit{ClassReference} in FROM clause. In cases where the type of
the OQL query is \textit{UnionQuery} or \textit{IntersectQuery}, the conversion
tree generates two OQL SELECT queries q1 and q2 that will be used in
the recursive procedure \textit{MappingOQLtoSPARQL} so as to construct the
SPARQL SELECT query of each one and concatenate them in order to
have an equivalent SPARQL SELECT query.

\begin{verbatim}
Input : An OQL SELECT Query, q_{in}
Output : A SPARQL query q_{out}

Begin
q_{out} = "" {A SPARQL query that is initially blank}
Tree = parse(q_{in}) {A parse tree obtained by parsing q_{in}}
q_{in} = SELECT = Tree.getRoot()
q_{in} = FROM_CR = Tree.getClassReference()
q_{in} = FROM_CI = Tree.getClassJoined()
q_{in} = WHERE = Tree.getWhereCond()
q_{out} = SELECT = \textit{"SELECT"} q_{in} \textit{WHERE = "WHERE \{"}
TP = \{ The set of triple patterns is initially empty \}
if (Tree.type == SimpleQuery) then
    TP = \textit{ConstructTriplePattern(q_{in} SELECT, q_{in} FROM_CR, NULL, q_{in} WHERE)}
    FilterExp = \textit{ConstructFilterExpression(q_{in} WHERE)}
    q_{out} += \textit{WHERE}{ {\textit{ConstructSparqlWhereClause(TP, FilterExp)}}}
    q_{out} += \textit{SELECT}{ {\textit{ConstructSparqlSelectClause(q_{in} SELECT)}}}
    q_{out} += \textit{WHERE}{ {\textit{ConstructSparqlWhereClause(TP, FilterExp)}}}
else if (Tree.type == JoinQuery) then
    TP = \textit{ConstructTriplePattern(q_{in} SELECT, q_{in} FROM_CR, q_{in} FROM_CI, q_{in} WHERE)}
    FilterExp = \textit{ConstructFilterExpression(q_{in} WHERE)}
    q_{out} += \textit{WHERE}{ {\textit{ConstructSparqlWhereClause(TP, FilterExp)}}}
    q_{out} += \textit{SELECT}{ {\textit{ConstructSparqlSelectClause(q_{in} SELECT)}}}
    q_{out} += \textit{WHERE}{ {\textit{ConstructSparqlWhereClause(TP, FilterExp)}}}
else if (Tree.type == UnionQuery) then
    q_{1} = Tree.leftSubTree(), q_{2} = Tree.rightSubTree()
    MappingOQLtoSPARQL(q_{1})
    MappingOQLtoSPARQL(q_{2})
    Merge(q_{1} out, q_{2} out, \textit{UNION})
else if (Tree.type == IntersectQuery) then
    q_{1} = Tree.leftSubTree(), q_{2} = Tree.rightSubTree()
    MappingOQLtoSPARQL(q_{1})
    MappingOQLtoSPARQL(q_{2})
    Merge(q_{1} out, q_{2} out, \textit{INTERSECT})
end if
return q_{out}

End Algorithm
\end{verbatim}

V. Conclusion

In summary, the main contribution of this paper in the pertinent topic
of interoperability between object oriented world and relational world
is the elaboration of a query conversion algorithm of the OQL SELECT
queries to SPARQL equivalent queries by translating each element
of OQL query (SELECT clause, FROM clause, FILTER constraint,
imPLICIT/explicit join and union/intersection SELECT queries) to its
equivalent in SPARQL language so as to bridge the gap between this
two world without a physical data transformation.

One obvious extension of our research is to reinforce our algorithm
by supporting more concepts, such as: subqueries, collections,
aggregation and composition.

\begin{verbatim}
References
2014.
SPARQL ”. ACM Transactions on Database Systems (TODS), 34(3), 16.
\end{verbatim}

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.


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 is a full professor in the Department of Mathematics and Computer Sciences from the University Hassan 1st Faculty of Sciences & Technology Settat Morocco. He is co-chairs of IC2INT, International Conference on Software Engineering, Databases and Expert Systems (SEDEXS’12), NASCASE’11. He has published over 80 peer-reviewed papers. His research interests are intelligent systems, ontologies engineering, partial and differential equations, numerical analysis and scientific computing. He is Editor-in-Chief for Australasian Journal of Computer Science and associate editor of Journal of Artificial Intelligence and Journal Software Engineering.