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 raisons 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.

DOI: 10.9781/ijimai.2016.373

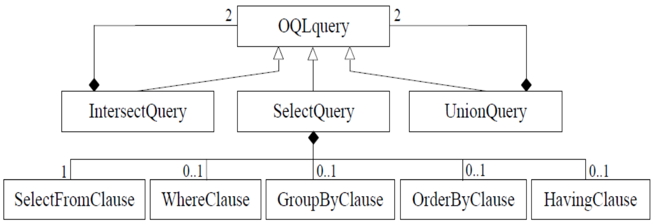


Fig. 1. The OQL Query representation

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.

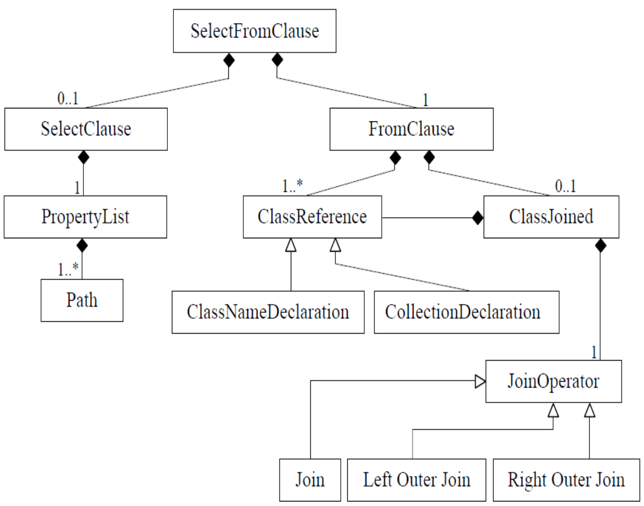


Fig. 2. Representation of the Select FROM Clause

The 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 (<,=<, >, >=,=) containing an attribute path and a value.

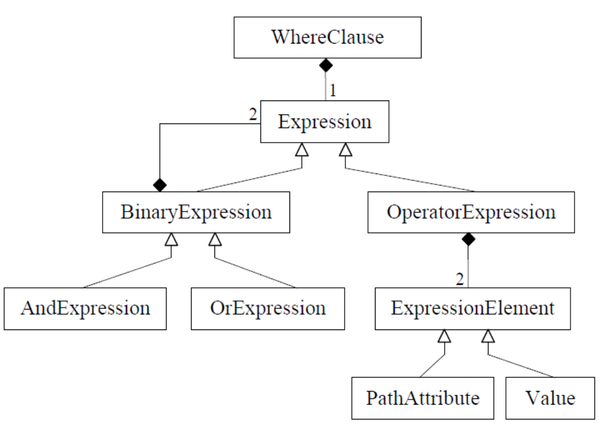


Fig. 3. Clause Representation of the Where Clause

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*.

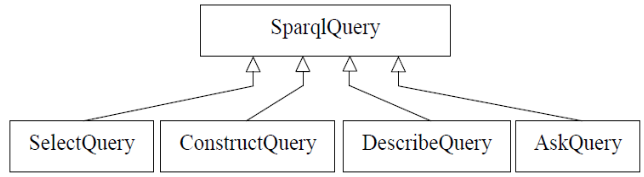


Fig. 4. The different types of Query Operation in the SPARQL metamodel

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.

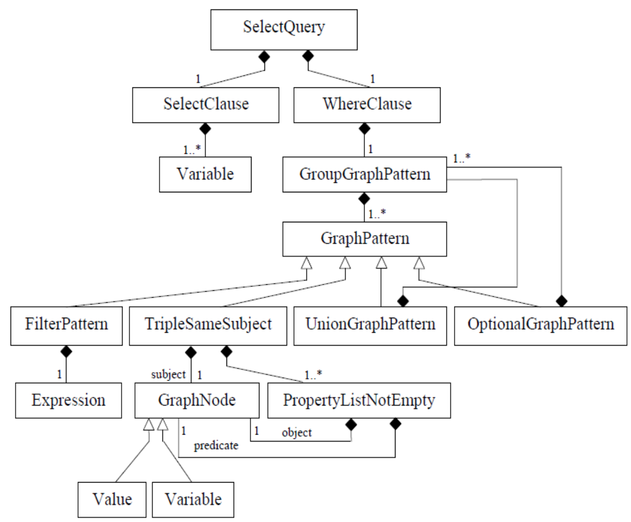


Fig. 5. Representation of the Group Graph Pattern in the SPARQL metamodel

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
QUERIES EXAMPLES USING OQL AND SPARQL

Query Type	OQL	SPARQL

Simple Query	SELECT p.name, p.age
	FROM Person p
	WHERE p.age<30
		SELECT ?name ?age
		WHERE {
		?who <Person#name> ?name;
		<Person#age> ?age.
		Filter (?age < 30)
		}

Explicit Join	SELECT p.name, a.city
	FROM Person p JOIN Address a
	ON p.addr = a.id
	WHERE a.state = “MA”
		SELECT ?name ?city
		WHERE {
		?who <Person#name> ?name;
		<Person#addr> ?addr.
		?addr <Address#city> ?city;
		<Address#state> ?state.
		Filter (?state = “MA”)
		}

Implicit Join	SELECT p.name, a.city FROM Person p, Address a WHERE a.state = “MA”	
Union Query	SELECT p.name FROM Person p WHERE p.age<20 UNION SELECT p.name FROM Person p JOIN Address a ON p.addr = a.id WHERE a.city = ”Nice”	SELECT ?name WHERE { ?who <Person#name> ?name; <Person#age> ?age. Filter (?age < 20) } UNION { ?who <Person#name> ?name; <Person#addr > ?addr. ?addr <Person#city> ?city Filter (?city = “Nice”) }

Intersect Query	SELECT p.name
	FROM Person p
	WHERE p.age>26
	INTERSECT
	SELECT p.name
	FROM Person p JOIN Address a
	ON p.addr = a.id
	WHERE a.city = ”Paris”
		SELECT ?name
		WHERE
		{
		{ ?who <Person#name> ?name;
		<Person#age> ?age.
		Filter (?age > 26)
		}
		{ ?who <Person#name> ?name;
		<Person#addr> ?addr.
		?addr <Address#city> ?city;
		Filter (?city = “Paris”)
		}
		}

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:

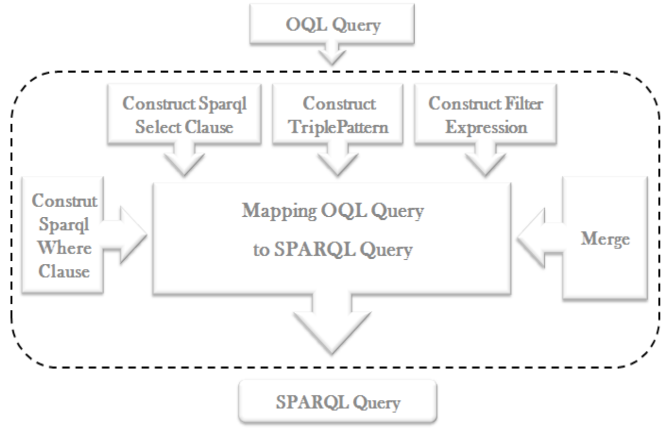


Fig. 6. Representation schema of our mapping approach

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 = “”
{A SPARQL SELECT Clause that is initially blank}
for each attribute attr ∈OSA **do**
    attrName = getAttrName($attr_i$)
    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_i in order to extract for each a_j attribute its name and the alias for its class; if the CR_i alias equal to the alias of the class attribute a_j, then it formulate the triple pattern of equivalent SPARQL query and adds it to the set TP and removing the attribute a_j 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_l = CJ.getJoinCond().loperand()
        CR.add(CJ.getCR())
    else if (CR.size>1) then //Implicit Join
        JCA_l = CR_1.getClassName().getExternalKey()
    end if
    if (A.ExistInList(JCA_l) == False) then
            A.add(JCA_l)
    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_j)
    attrName = getAttrName(a_j)
    if (CR_i.getClassAlias() == attrClassAlias) then
      tp←{?s_i <CR_i.getClassName()#attrName> ?attrName}
      if (simpleJoin==True AND
          tp.object == CRi.getClassName().getFK()) then
      ?s_i+1 = tp.object
      end if
     TP.add(tp)
     A.remove(a_j)
    else
      Atemp.add(a_j)
    end if
  end for
  while (Atemp.size>0) do
      A.add(Atemp.remove())
  end while
end for
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}
     where += ";" + TP[i]
   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, q_{in} so as to return at the end the SPARQL equivalent query, q_{out}. 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 *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 "*JoinQuery*" except that the OQL conversion tree will generates the *ClasseJoined* in addition to *ClassReference* in FROM clause. In cases where the type of the OQL query is "*UnionQuery*" or "*IntersectQuery*", the conversion tree generates two OQL SELECT queries q1 and q2 that will be used in the recursive procedure *MappingOQLtoSPARQL* so as to construct the SPARQL SELECT query of each one and concatenate them in order to have an equivalent SPARQL SELECT query.

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.getSelectClause();
$q_{in}^{FROM\_CR}$ = tree.getClassReference()
$q_{in}^{FROM\_CJ}$ = tree.getClassJoined()
q_{in}^{WHERE} = tree.getWhereCond()
q_{out}^{SELECT} = "SELECT" q_{out}^{WHERE} = "WHERE {"
TP ← ø { The set of triple patterns is initially empty}
if (tree.type == SimpleQuery) **then**
TP = ConstructTriplePattern(q_{in}^{SELECT}, $q_{in}^{FROM\_CR}$, NULL, q_{in}^{WHERE})
FilterExp = ConstructFilterExpression(q_{in}^{WHERE})
q_{out}^{WHERE} += ConstructSparqlWhereClause(TP, FilterExp)
q_{out}^{SELECT} += ConstructSparqlSelectClause(q_{in}^{SELECT})
q_{out} = q_{out}^{SELECT} + q_{out}^{WHERE} +" }"
else if (tree.type == JoinQuery) **then**
TP = ConstructTriplePattern(q_{in}^{SELECT}, $q_{in}^{FROM\_CR}$, $q_{in}^{FROM\_CJ}$, q_{in}^{WHERE})
FilterExp = ConstructFilterExpression(q_{in}^{WHERE})
q_{out}^{WHERE} += ConstructSparqlWhereClause(TP, FilterExp)
q_{out}^{SELECT} += ConstructSparqlSelectClause(q_{in}^{SELECT})
q_{out} = q_{out}^{SELECT} + q_{out}^{WHERE} +" }"
else if (tree.type == UnionQuery) **then**
q_1 = tree.leftSubTree(), q_2 = tree.rightSubTree()
q_1^{out} = MappingOQLtoSPARQL(q_1)
q_2^{out} = MappingOQLtoSPARQL(q_2)
q_{out} = Merge(q_1^{out}, q_2^{out}, "UNION")
else if (tree.type == IntersectQuery) **then**
q_1 = tree.leftSubTree(), q_2 = tree.rightSubTree()
q_1^{out} = MappingOQLtoSPARQL(q_1)
q_2^{out} = MappingOQLtoSPARQL(q_2),
q_{out} = Merge(q_1^{out}, q_2^{out}, "INTERSECT")
end if
return q_{out}
End Algorithm

F. Merge Subprocedure

The *Merge* subprocedure takes as inputs two OQL subqueries and the merge type in order to generate a significant and valid SPARQL query. Firstly, it extracts the SELECT clauses from each subqueries and encapsulate these in S1 and S2, secondly, it extracts and encapsulate the triple patterns of each subqueries in TP1 and TP2. Finally, it extracts and stores the FILTER expressions of each the subqueries in F1 and F2. If the merge type is "UNION" then the q_{out}'s SELECT clause takes one of subqueries SELECT clause, and the q_{out}'s WHERE clause is formulated from the concatenation of the q1's WHERE clause returned by the *ConstructSparqlWhereClause* Subprocedure taking as inputs TP1 and F1 as well as the keyword UNION and the q2's WHERE clause returned also by the *ConstructSparqlWhereClause* Subprocedure taking as inputs TP2 and F2. We proceed with the same manner if the SPARQL query type is "JoinQuery" except that we remove the keyword Union.

Input: q1, q2, mergeType
Output: A SPARQL query qout
Begin
qout = "SELECT"
S1=q1.ExtractSelectClause();
S2=q2.ExtractSelectClause();
TP1= q1.ConstructTriplePatterns();
TP2= q2.ConstructTriplePatterns();
F1 = q1.ExtractFilter();
F2 = q2.ExtractFilter();
if (mergeType = "UNION") **then**
qout += ConstructSparqlSelectClause(S1)
qout+="WHERE{ {" + ConstructSparqlWhereClause(TP1, F1) + "} UNION {" + ConstructSparqlWhereClause(TP2, F2) + "} }"
else if (mergeType = "INTERSECT") **then**
qout += ConstructSparqlSelectClause(S1)
qout += "WHERE{ {" + ConstructSparqlWhereClause(TP1, F1) + "}{" + ConstructSparqlWhereClause(TP2, F2) + "} }"
end if
return qout
End Algorithm

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.

References

[1] T. Berners-Lee, J. Hendler and O. Lassila, "The Semantic Web, Scientific American", Mai 2001.

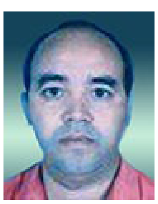
[2] S. Bechhofer, "OWL: Web ontology language". In Encyclopedia of Database Systems (pp. 2008-2009). Springer US, 2009.

[3] World Wide Web Consortium. RDF 1.1 Concepts and Abstract Syntax. 2014.

[4] J. Pérez, M. Arenas & C. Gutierrez, "Semantics and complexity of SPARQL ". ACM Transactions on Database Systems (TODS), 34(3), 16,

2009
[5] V. Eisenberg and Y. Kanza, “D2RQ/update: updating relational data via virtual RDF”, In Proceedings of the 21st international conference companion on World Wide Web (pp. 497-498), 2012.
[6] 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.
[7] K. Sengupta, P. Haase, M. Schmidt, and P. Hitzler, “Editing R2RML mappings made easy”, 2013.
[8] S. Das, S. Sundara, and R. Cyganiak, “R2RML: RDB to RDF Mapping Language”. Working draft, W3C, 2011.
[9] C. Bizer, A. Seaborne, “D2RQ: treating non-RDF databases as virtual RDF graphs”, In: International Semantic Web Conference ISWC (posters), 2004.
[10] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller, “Triplify--Light-Weight Linked Data Publication from Relational Databases”, In Proceedings of the 18th International World Wide Web Conference, 2009.
[11] W. Corno, F. Corcoglioniti , I. Celino and E. Della Valle, “Exposing Heterogeneous Data Sources as SPARQL Endpoints through an Object-Oriented Abstraction”, In: Asian Semantic Web Conference (ASWC 2008), pp. 434–448, 2008.
[12] G. Hillairet, F. Bertrand, & J. Y. Lafaye, “Rewriting queries by means of model transformations from SPARQL to OQL and vice-versa”, In Theory and Practice of Model Transformations (pp. 116-131), Springer Berlin Heidelberg, 2009.
[13] SPARQL. (2015, avril 27). Wikipedia. Retrieved from https://fr.wikipedia.org/wiki/SPARQL. Last visited January 2016.
[14] OQL Grammar. (2015, September 23). Retrieved from https://wiki.openitop.org/doku.php?id=2_2_0:oql:oql_grammar. Last visited January 2016.
[15] E. Prud’hommeaux, S. Harris, Garlik, and A. Seaborne. SPARQL 1.1 Query Language. (2013, 21 March) . Retrieved from http://www.w3.org/TR/sparql11-query/#sparqlGrammar. Last visited december 2015.
[16] S. Guruzu, and G. Mak, “HQL and JPA Query Language”. Hibernate Recipes: A Problem-Solution Approach, 155-166, 2010.
[17] J. Juneau, “The Query API and JPQL”. In Java EE 7 Recipes (pp. 447-470). Apress, 2013.

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.