Using Heterogeneous Mappings for Rewriting SPARQL Queries

Fernanda Lígia Rodrigues Lopes
Federal Institute of Education, Science and Technology of Ceará
Fortaleza, CE, Brazil
nandaligia@gmail.com

Eveline Russo Sacramento
Department of Informatics
PUC-Rio
Rio de Janeiro, RJ, Brazil
esacramento@inf.puc-rio.br

Bernadette Farias Lóscio
Informatics Center
Federal University of Pernambuco
Recife, PE, Brazil
bfl@cin.ufpe.br

***Abstract*—In this paper we present an approach for rewriting queries between heterogeneous ontologies. In our query rewriting solution, we combine the semantics and expressiveness of SPARQL with notions of logic programming and adopt a rule-based formalism for representing mappings between ontologies. We also deal with some relevant questions, including: structural heterogeneities and pruning of irrelevant parts of a rewritten query.**

***Keywords*—query rewriting; SPARQL; schema mappings; rules.**

I. Introduction

In the Semantic Web, the definition of resources and relationships between them are described by ontologies, which are, in general, independently developed, even for a same domain. Under such an environment, there is often the need to access distinct ontologies without regard to their heterogeneities [1]. In some cases, query answers can be obtained using ontologies different from those initially used to express the query, for example. Thus, in this context, if an application requires querying different ontologies or data sources without formulating a new query for each ontology, it may be necessary to deal with the problem of query rewriting between distinct ontologies. This problem occurs, for example, when a query submitted to a source ontology O_S needs to be reformulated in terms of a target ontology O_T. In this case, we have to answer the following question: how to describe and explore mappings between O_S and O_T in order to rewrite a SPARQL query expressed over O_S into a semantically equivalent query in terms of O_T, considering that the query results must be restructured and presented according to O_S?

In this paper, we address the problem described above and present an approach for rewriting SPARQL queries between heterogeneous ontologies. The main distinguishing features of our proposal are: (i) it can deal with ontologies with distinct structures, through the manipulation of heterogeneous mappings between their concepts; (ii) it implements an approach that combines the semantics and expressiveness of SPARQL with notions of annotated logic programming; (iii) it works with some SPARQL operators that were not addressed in previous works; (iv) it allows the pruning of some nodes of the rewritten query, which do not contribute towards its answering; and (v) it allows the representation of query results in terms of the source ontology. The remainder of this paper is organized as follows. Section II presents our approach for dealing with mappings between heterogeneous ontologies. Section III contains the query rewriting proposal, while Section IV describes implementation issues and the validation of our approach. Section V discusses some related work and finally, Section VI presents our main conclusions.

II. Dealing with Heterogeneous Ontologies

Ontologies have been extensively adopted to provide a common understanding for terms of specific domains. However, since such ontologies can be independently developed, it is common to find ontologies overlapping with each other and also providing distinct views for a same domain. For example, two ontologies can describe similar information using different levels of granularity, concepts or restrictions. In this work, we classify the heterogeneities between ontologies into two main groups, as detailed in the following. To illustrate our explanation, we use excerpts of OWL ontologies describing data about virtual sales of products (Figures 1 and 2) adapted from the *Amazon* virtual store. The namespace prefixes "s:" and "a:" refer to the vocabulary of Sales (Figure 1) and Amazon (Figure 2) ontologies, respectively.

Concept-Based Heterogeneity. Occurs when two ontologies represent the same information through entities of *distinct natures*. By *entity nature*, we mean the type of ontology element, e.g., a class, a datatype property, an object property (relationship) or an instance. For example, two ontologies are named *heterogeneous by concept* if a given information, expressed as a class in the former, is represented as a property in the later, or vice versa. Figure 1 and Figure 2 exemplify this kind of heterogeneity. We may observe that the information about publishers in ontology O_2 is modeled using a datatype property (a:publisher), while in ontology O_1 it is depicted by a relationship (an object property s:publishedBy) and a class (s:Publisher).

Restriction-Based Heterogeneity. Occurs when restrictions are applied to the source or the target ontology, or even to both ontologies. We deal with two types of restrictions:

(i) *Class restriction* [2]: occurs when a restriction is used to delimit a subset of a specific class. We consider that a restriction is a triple *(prop, op, const)*, where: *prop* is a datatype

property, *op* is a comparison operation and *const* is a constant value assigned to the property *prop*. In this work, we deal with the following comparison operators: equal-to ($=$), greater-than ($>$), greater-than-or-equal ($\geq$), less-than ($<$), less-than-or-equal ($\leq$) and different-from ($\neq$). A built-in predicate is a predicate that allows us expressing the mentioned restrictions. Note that class restriction helps us when at least one of the ontologies being mapped has little or no specialization of some top level classes. Figure 1 and Figure 2 illustrate an example where a class a:Book in ontology O_2 corresponds to a class s:Publication in ontology O_1, whose property s:type has the value "book".

(ii) *Subsumption restriction*: occurs when the same information is represented in different ontologies using distinct levels of granularity with respect to a class hierarchy (subsumption of concepts).

Finally, it is important to mention that the heterogeneities explained before may appear in a combined way.

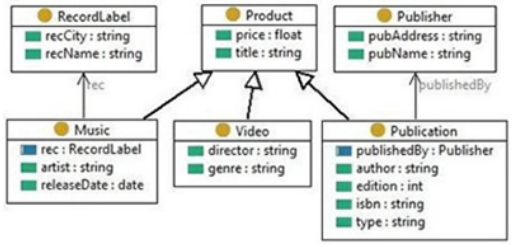

Figure 1. Sales ontology (O_1)

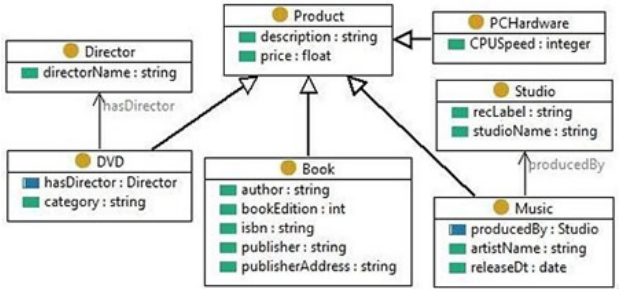

Figure 2. Amazon ontology (O_2)

A. Ontology Mapping Classification

Considering that ontologies need to interoperate with each other, we adopt a formalism for describing ontology mappings, which allows representing the association between ontologies, while deals with the different types of heterogeneities previously described. Our mappings can be:

(i) *Homogeneous or heterogeneous*. A mapping is called homogeneous when it maps elements of a same type, e.g., concepts to concepts, roles to roles and individuals to individuals. By contrast, heterogeneous mappings relate elements of different natures [3], [4]. Note that heterogeneous mappings allow dealing with *concept-based heterogeneities*.

(ii) *Simple or complex*. A mapping is called simple when it specifies a direct relationship between ontology elements of two distinct ontologies. On the other hand, when a mapping involves a complex expression, in at least one side of the mapping, it is called a complex mapping [4]. Class restriction's heterogeneities can be supported by such complex mappings. We point out that both homogeneous and heterogeneous mappings can be simple or complex.

Table I presents some examples of mappings. Note that the left side of a mapping contains concepts of the source ontology, while the right side shows concepts of the target ontology. In our example, we consider that Sales ontology *(O_1)* is the source ontology, while Amazon ontology *(O_2)* is the target ontology. For the sake of simplicity and space, we do not show all mappings that can be obtained between those ontologies, but only the most relevant ones.

TABLE I
MAPPINGS BETWEEN SALES ONTOLOGY AND AMAZON ONTOLOGY

#1:	s:Music(x)	$\Leftarrow$	a:Music(x)
#2:	s:Video(x)	$\Leftarrow$	a:DVD(x)
#3:	s:Publication(x)	$\Leftarrow$	a:Book(x)
#4:	s:type(x,k)	$\Leftarrow$	a:Book(x), k = "book"
#5:	s:Product(x)	$\Leftarrow$	a:Book(x); a:DVD(x); a:Music(x)
#6:	s:Publisher(fPublisher(y))	$\Leftarrow$	a:publisher(x,y), a:Book(x)
#7:	s:RecordLabel(fRLabel(y))	$\Leftarrow$	a:producedBy(x,z), a:recLabel(z,y), a:Music(x)
#8:	s:edition(x,y)	$\Leftarrow$	a:bookEdition(x,y), a:Book(x)
#9:	s:author(x,y)	$\Leftarrow$	a:author(x,y), a:Book(x)
#10:	s:price(x,y)	$\Leftarrow$	a:price(x,y), a:Book(x); a:price(x,y), a:DVD(x); a:price(x,y), a:Music(x)
#11:	s:genre(x,y)	$\Leftarrow$	a:category(x,y), a:DVD(x)
#12:	s:director(x,y)	$\Leftarrow$	a:hasDirector(x,z), a:directorName(z,y), a:DVD(x)
#13:	s:pubName(fPublisher(y),y)	$\Leftarrow$	a:publisher(x,y), a:Book(x)
#14:	s:pubAddress(fPublisher(y),z)	$\Leftarrow$	a:publisherAddress(x,z), a:publisher(x,y), a:Book(x)
#15:	s:publishedBy(x,fPublisher(y))	$\Leftarrow$	a:publisher(x,y), a:Book(x)
#16:	s:recName(fRLabel(y),y)	$\Leftarrow$	a:recLabel(z,y), a:Music(x)
#17:	s:rec(x,fRLabel(y))	$\Leftarrow$	a:producedBy(x,z), a:recLabel(z,y), a:Music(x)
#18:	s:title(x,y)	$\Leftarrow$	a:description(x,y), a:Book(x); a:description(x,y), a:DVD(x); a:description(x,y), a:Music(x)

In summary, mappings like #1 and #2 are simple and homogeneous, once a simple expression maps a class to another class. Similarly, we have a property mapping in #8. On the other hand, #6 and #7 are examples of heterogeneous mappings. About the semantics of the presented mappings, we can do some remarks:

- Mapping #5, for example, deals with *subsumption restriction*, once a:Book, a:DVD and a:Music are s:Product (semicolon represents union).
- Mappings #3 and #4 reflect a *class restriction* over the source ontology O_1. It is important to say that two rules are used for this representation, as the proposed formalism enables just atomic elements on the left side.
- Mappings #6, #13, #14 and #15 deal with information restructuration about Books. In O_1, this book information is represented through two classes (s:Publication, s:Publisher) and a relationship (s:publishedBy) , while in

O_2 there is a single class (a:Book). For this reason, mappings with *Skolem Functions* are used to correctly relate the entities, i.e., to return as an object, corresponding to an instance of the class s:Publisher, an information that was originally expressed as a string (datatype property a:publisher).

B. Ontology Mapping Representation

In some previous works, ontology mappings were described using DL (Description Logic)[8], [11] or an extension of DL (DDL - Distributed Description Logic)[3]. Although DL is useful for expressing semantic relationships between ontology concepts, such language has limitations in some important aspects of our work. For example, regarding the object restructuration, DL does not allow the explicit definition of object identifiers (OIDs). To overcome these limitations, we adopt an extended rule-based formalism that is able to represent objects and includes a suitable mechanism for building object identifiers. Hence, we can describe heterogeneous mappings and also properly deal with the restriction-based heterogeneities, when the restriction appears in any one of the two ontologies. Next, we describe how our ontology mappings can be specified according to the adopted formalism.

Let F be a set of function symbols, B be a set of atomic concepts and atomic roles, and V be a set of variables. A *constant* is a 0-ary function symbol. The set of *terms* over F and V is recursively defined as follows: (i) each variable v in V is a term; (ii) each constant c in F is a term; (iii) if $t_1, ..., t_n$ are terms, and f is an n-ary function symbol in F, then $f(t_1, ..., t_n)$ is a term. An *atom* over F, B and V is an expression of the form $c(t)$, where c is an atomic concept and t is a term, or of the form $p(t, u)$, where p is an atomic role or a built-in predicate and t and u are terms.

Let O_S and O_T be two ontologies and O_R be a rule language. The set of mappings between O_S and O_T are specified through a set of *mapping rules* of the form $\beta_1(w_1) \Leftarrow \alpha_1(v_1), ..., \alpha_m(v_m)$, where: (i) $\alpha_1(v_1), ..., \alpha_m(v_m)$, called the *body* of the mapping rule, is an atom or an atom conjunction, where $\alpha_i(v_i)$ is an atom whose atomic concept or atomic role occurs in the target ontology O_T, or a built-in operation; (ii) $\beta_1(w_1)$, called the *head* of the mapping, is an atom whose atomic concept or atomic role occurs in the source ontology O_S. This rule-based formalism (a datalog extension) supports *Skolem functions* for the creation of new *object identifiers* of classes in O_S from one or more properties of O_T. So, these mapping rules allow the construction of URIrefs for new objects in O_S as terms of the form $f(t_1, ..., t_n)$, where f is an n-ary function symbol and $t_1, ..., t_n$ is a sequence of terms of O_S. We refer the reader to [5] for further information about our initial strategy to obtain these mappings. It is worth to mention that, to deal with a large ontology, this strategy may be applied to parts of such ontology, in order to incrementally generate the mappings.

III. The Query Rewriting Approach

In this section, we present our approach for rewriting SPARQL queries between heterogeneous ontologies. To help the manipulation of heterogeneous mappings, expressed by rules, our solution was inspired on some notions of annotated logic programming [6]. The input of the query rewriting process is a SPARQL query formulated in terms of a source ontology and a set of mapping rules between the source and the target ontology. In order to rewrite the input query into a semantically equivalent query described in terms of the target ontology, the following activities are executed:

1) Query Normalization. Extracts the basic blocks of construction of a SPARQL query and expresses such query in a homogeneous format, eliminating syntactic divergences and considering implicit rules of precedence and association of operators. This activity turns easy the manipulation of a query and eliminates any ambiguity found during the query analysis.

2) Generation of a SPARQLD tree. In our approach, a SPARQL query is represented by a tree that reflects, through a set of rules and annotations, the semantics of the query's graph pattern. This tree, called SPARQLD, has its construction based on the SLD tree [6], which is a special tree representing possible derivations of a goal ***G***, related to a program ***P***. In PROLOG systems, a SLD tree represents the search and resolution space that reflects a queue of goals built during the computation of ***G***. As a result, it is provided an answer that, in general, consists of an affirmation (or a negation). In this work, we are not interested in receiving an answer, but in generating a set of rules that represents the rewritten query. It is important to say that a SPARQLD tree has its nodes enriched by annotations with the objective of properly representing the semantics of the SPARQL constructors, and the information necessary to the rewriting process. These annotations store, for example, the reference identifier of the node, information about the graph pattern and about the filters (internal and external) applied in such node.

3) Query Rewriting and Results Restructuring. Receives a SPARQLD tree that represents the query and a set of mapping rules. The objective of this activity is to update the SPARQLD tree with the derivations obtained by the rules, so the tree can reflect the rewritten query. Furthermore, the query rewriting algorithm produces two additional artifacts: (i) a set of updates, referring to the format of the query result, which turns possible to present the query answer according to the target ontology; and (ii) a set of subtrees that must be discarded from the rewritten query, as they would return empty or redundant results.

In the following, we present an example to illustrate our query rewriting approach. Consider that query *Q = "Return the title and the author of books whose price is greater than 30 dollars. Optionally, return information about the publisher of these books. Order the results by title"*, presented in Figure 3, is submitted over the Sales ontology and should be rewritten in terms of the Amazon ontology. In order to obtain the corresponding rewritten query, the set of mappings presented

in Table I are used.

```
SELECT ?tl ?au ?e ?ne ?ee
FROM <Sales.owl>
WHERE{
    {?l s:title ?tl .
    ?l s:type ?b .
    ?l s:author ?au
    OPTIONAL{
        ?e s:pubName ?ne .
        ?e s:pubAddress ?ee.
        ?l s:publishedBy ?e}
    ?l s:price ?p .
    FILTER (?b = 'book' && ?p > 30)}
}ORDER BY ?tl
```

Figure 3. SPARQL query Q

Figure 4 presents the SPARQLD tree built for *Q*, where: nodes 1 (node identifier is repeated due to the OPTIONAL semantics) and 2 represent the original query, while the other nodes are constructed or updated during the query rewriting process; each filter expression in the query is symbolized by a syntax tree that has an identifier. Table II shows the node predicates, which are updated in every call. In this figure, the predicates being resolved are underlined, while the resolved predicates are in italics.

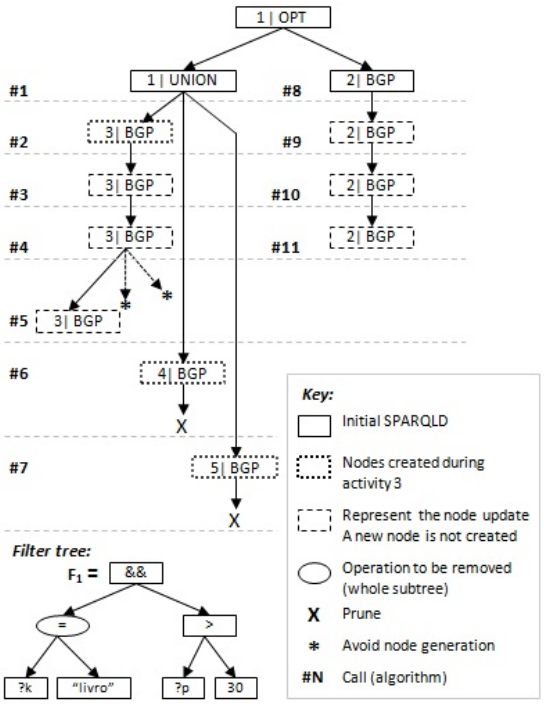

Figure 4. SPARQLD tree for Q

For each leave of the initial SPARQLD, deep-first search is applied. For example, in node 1 (call #1), the first predicate to be rewritten is *s:title(l,tl)*. Using rule #18, this node is expanded to three new nodes (3, 4 and 5), once the mapping expresses a disjunction. Then, in second call (#2), mapping rule #4 is used to resolve the predicate *s:type(l,k)*. We may observe that rule #4 contains a restriction over the source ontology. So, the algorithm verifies if the restriction *k = 'book'* (from such rule) is compatible with the value presented in *Q*. In this case, it is true. However, if instead of *'book'* it was *'magazine'*, for example, the node execution would be discontinued. As indicated by the elliptical node of the Filter tree (Figure 4), the operation's subtree corresponding to such restriction should also be removed. This is done because this restriction does not appear in the target ontology. Through Figure 4, it is possible to see that from call #2 to call #3 there are no new nodes because there are no new disjunctions. However, as showed in Table II, the list of predicates is still updated. In call #3, mapping rule #9 is used to resolve the predicate *s:author(l,au)*.

Next, call #4 uses the rule #10 (*s:price*). This case would generate three new nodes, however just the node referring to book is created, unlike the other two (DVD and Music). It occurs because we have the condition *a:Book* in predicate *a:author*. So, since Book, DVD and Music do not share common instances, the last two nodes may be discarded. On the other hand, if they share instances, then they will be returned by the branch referring to book. Hence, the generation of two new branches is avoided. In #6 and #7, the initiated subtrees are discarded by similar reasons explained in call #4. However, note that in this case the predicate is *s:type(l,b)*. Regarding call #8, the predicate *s:pubName(e,ne)*, uses mapping rule #13. This rule reflects a *concept-based heterogeneity*. Observe that the rule head has a function. This case happens because the target ontology does not have a class representing Publisher. Then, the variable e (corresponding to the object identifier, namely, URI) cannot be obtained. For that reason, such URI should be constructed using the function *fPublisher(y)*, from the value that will be obtained through the query execution.

TABLE II
PREDICATES OF THE SPARQLD TREE FOR Q

#	idRef	Predicates
-	1	q1(au,e,ee,k,l,ne,p,tl)
1	1	s:title(l,tl), s:type(l,b), s:author(l,au), s:price(l,p)
2	3	s:type(l,b), s:author(l,au), s:price(l,p), *a:description(l,tl)*, *a:Book(l)*
3	3	s:author(l,au), s:price(l,p), *a:description(l,tl)*,*a:Book(l)*
4	3	s:price(l,p), *a:description(l,tl)*, *a:Book(l)*, *a:author(l,au)*
5	3	*a:description(l,tl)*, *a:Book(l)*, *a:author(l,au)*, *a:price(l,p)*
6	4	s:type(l,b), s:author(l,au), s:price(l,p), *a:description(l,tl)*, *a:DVD(l)*
7	5	s:type(l,b), s:author(l,au), s:price(l,p), *a:description(l,tl)*, *a:Music(l)*
8	2	s:pubName(e,ne), s:pubAddress(e,ee), s:publishedBy(l,e)
9	2	s:pubAddress(e,ee), s:publishedBy(l,e), *a:publisher(_p13,ne)*, *a:Book(_p13)*
10	2	s:publishedBy(l,e), *a:publisher(_p13,ne)*, *a:Book(_p13)*, *a:publisherAddress(_p14,ee)*
11	2	*a:publisher(l,ne)*, *a:Book(l)*, *a:publisherAddress(l,ee)*

The remainder of the rewriting process occurs in a similar way. Besides its output, the rewriting algorithm may produce the query answer in a tabular form, as well as RDF triples (to this case, CONSTRUCT may be used). Figure 5 shows the rewritten query for *Q*. To obtain this query through the tree exhibited in Figure 4, such tree should be traversed,

reconstructing the operators and filters and removing discarded branches, filters or any unnecessary information. Moreover, the functions for building the URIs are created and associated to their respective variables. We emphasize that these functions could be implemented in a procedure outside the query.

```
SELECT ?tl ?au ?e ?ne ?ee
FROM <Amazon.owl>
WHERE{
   {?l a:description ?tl .
   ?l rdf:type a:Book .
   ?l a:autor ?au .
   ?l a:price ?p
   OPTIONAL{
      ?l a:publisher ?ne .
      ?l rdf:type a:Book .
      ?l a:publisherAddress ?ee}
   FILTER (?p > 30)}
   LET (?e := IRI(fn:concat(a:,?ne)))
}ORDER BY ?tl
```

Figure 5. Rewritten query for Q

IV. Implementation Issues and Validation

To validate our approach, we developed a tool, named *SQuOL*, and we performed some experiments. *SQuOL* has been implemented in Java and Jena API[1] was used to manipulate both ontologies and SPARQL queries. More specifically, queries were manipulated through ARQ API[2]. Both the mapping model and ontology mappings were described in XML. *SQuOL* also offers a user-friendly interface where users can easily formulate and rewrite queries, and manipulate ontology mappings. Concerning the experiments, we considered the following:

- Five (5) OWL ontologies, divided in two groups: sales and education. The first set was based on the ontologies adopted by [5], while the second one represents ontologies available on the Web.
- A set of mappings between the mentioned ontologies. In each group, for every pair of ontologies, each one was tested as both source and target ontology, as the mappings are unidirectional.
- A set of twenty six (26) queries was chosen considering: the main SPARQL constructs, different types of heterogeneous mappings and different types of queries (conjunctive queries, disjunctive queries and queries with negation, in particular, negation as failure).

In all experiments, the queries were correctly rewritten and the irrelevant parts of the output query were pruned as expected. The obtained results shown that *SQuOL* is able to correctly handle with heterogeneous and complex mappings during the rewriting of SPARQL queries.

V. Related Work

In this section, we briefly present some of the research literature related to our work. In [7], the authors focus on accessing relational data using *DL-Lite*$_A$ ontologies. They expand a query using ontology terminological components (*TBox*) before applying the obtained mappings. In our work, the query expansion phase occurs during mappings generation. This way, we avoid expansions each time a query is submitted to an ontology and we prevent unnecessary expansions of class hierarchies. Fernandes [8] proposes a semantic query reformulation process in order to obtain approximate query answers instead of exact ones. However, such queries, expressed in DL or SPARQL, can be constructed just over classes and cannot be freely formulated, as they are based on existing templates. The approaches presented in [9] and [11] directly deal with SPARQL queries. However, they do not treat filter expressions in the queries and only consider homogenous mappings expressed in RDF ([9]) and DL ([11]). In [10] the authors propose a method to deal with native query rewriting in SPARQL. However, they do not address heterogeneity questions mentioned in our work.

[1]http://jena.sourceforge.net/

[2]http://jena.sourceforge.net/ARQ/app_api.html

VI. Conclusion

In this work, we presented an approach for query rewriting between heterogeneous ontologies. One of the main distinguishing issues of our solution is that it can deal with ontologies with distinct structures through the manipulation of heterogeneous mappings between their concepts. We also presented the formalism for representing such mappings and we described how they could be used during the query rewriting process. As a future work, we plan to develop new experiments considering larger ontologies and known datasets.

References

[1] Y. Li and J. Heflin, "Using Reformulation Trees to Optimize Queries over Distributed Heterogeneous Sources", Proc. International Semantic Web Conference (ISWC 2010), Springer, pp.502-517.

[2] R. Parundekar, C. Knoblock and J. Ambite, "Linking and Building Ontologies of Linked Data", Proc. International Semantic Web Conference (ISWC 2010), Springer, pp.598-614.

[3] C. Ghidini and L. Serafini, "Reconciling concepts and relations in heterogeneous ontologies", Proc. 3rd European Semantic Web Conference (ESWC 2006), Springer, pp.50-64.

[4] R. De Virgilio, F. Giunchiglia and L. Tanca, Semantic Web Information Management: A Model-Based Perspective, 1st ed. Springer-Verlag New York Inc, 2010.

[5] E. Sacramento, V. Vidal, J. Macêdo, B. Lóscio, F. Lopes and M. Casanova, "Towards automatic generation of application ontologies", Journal of Information and Data Management (JIDM), SBC, 2010, pp.535-551.

[6] J. Lloyd, Foundations of Logic Programming. Berlin: Springer-Verlag, 1987.

[7] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro. "Ontologies and databases: The DL-lite approach", Reasoning Web, Springer, 2009, pp.255-356.

[8] D. Fernandes, Using Semantics to Enhance Query Reformulation in Dynamic Distributed Environments. PhD thesis, Federal University of Pernambuco, Brazil, 2009.

[9] G. Correndo, M. Salvadores, I. Millard, H. Glaser and N. Shadbolt, "Sparql query rewriting for implementing data integration over linked data", In Proceedings of the International Workshop on Data Semantics, 13th EDBT, 2010, pp.1-11.

[10] W. Le, S. Duan, A. Kementsietsidis, F. Li and M. Wang, "Rewriting queries on SPARQL views", Proc. 20th International Conference on World Wide Web, ACM, 2011, pp.655-664.

[11] K. Makris, N. Bikakis, N. Gioldasis and S. Christodoulakis, "SPARQL–RW: Transparent Query Access over Mapped RDF Data Sources", Proc. EDBT Demo Session, Springer, 2012, pp.1108-1117.