A Robust Optimization Approach of SQL-to-SPARQL Query Rewriting

Ahmed Abatal[1], Mohamed Bahaj[2], Soussi Nassima[3]
Mathematics and Computer Science Department
Hassan I University, Faculty of Sciences and Techniques Settat, Morocco

***Abstract*—In order to ensure the interoperability between semantic web and relational databases, several approaches have been developed to ensure SQL-to-SPARQL query transformation direction, but all these approaches have the same weakness. In fact, they convert directly the input SQL query to its equivalent SPARQL one without any pre-processing phase enabling the optimization of this input query filled by users before starting the conversion process. This weakness has motivated us to add a pretreatment phase aiming to optimize the most important SQL statements which seem to have the biggest impact on the effectiveness of the transformed queries. Our main contribution is to enrich these rewriting systems by adding an optimization layer that integrate a set of simplification rules of Left, Right and Full Outer Join in order to avoid, firstly unnecessary operations during the conversion process, and secondly SPARQL queries with a high complexity due to Optional patterns obtained from outer join in this conversion context.**

Keywords*—SQL-to-SPARQL; outer join optimization; query transformation; SQL simplification; query optimization layer

I. Introduction

In the last decades, the semantic web [10] has emerged as an extension of the classic web aiming to exploit the full web potential by providing a common framework for knowledge to be shared across applications. It is a W3C recommendation that offers an easier way to search, share, reuse and combine information. It allows machines to understand the semantics of data on the web in order to conceive a globally-extended knowledge base that links data from different sources and ensure a better cooperation between computers and people. The semantic web uses numerous technologies to achieve the previous goals: RDF (Resource Description Framework) [4] as a flexible and standard data model for representing information on the web and make it machine readable, OWL (Web Ontology Language) [3] as the famous language of knowledge representation for creating structured ontology and SPARQL query language [5] for querying data from RDF graphs.

However, the majority of web data is stored in relational databases, which motivate the web researchers to develop a set of methods aiming to offer a better interoperability context between the both systems. In this light, some approaches have been made regarding SQL to SPARQL query transformation in order to facilitate data extraction for relational users by querying RDF stores with SQL language, but unfortunately, all these approaches have the same weakness in their proposed systems since they convert directly the input SQL query to its equivalent SPARQL one without any optimization phase enabling the pre-processing of this SQL query before starting the mapping process.

This weakness has motivated us to operate in this topic so as to remedy this gap and establish an intermediate step aiming to improve existing SQL-to-SPARQL mapping approaches by optimizing outer join clauses that leads to generate equivalent SPARQL query with Optional patterns responsible of the high complexity of the output query (SPARQL complexity evaluation PSPACE-hard [9]).

The remainder of this paper is organized as follows: Section II presents the key contribution of some related works. Section III gives some theoretical background related to the current topic. Section IV describes the main rules using by our optimization layer. Section V presents the functional architecture of the conceived layer and the proposed algorithm that aims to simplify all outer join types (left, right and full outer join) before starting the SQL-to-SPARQL conversion process. In Section VI we expose the java application implementing our solution and a comparison results summarizing the executing time of generated SPARQL queries in the optimized and direct mode. Finally, Section VII concludes our work and suggests some future extensions of this topic.

II. Related Works

Since the exponential emergence of semantic web in the last decades, researchers in the four corners of the earth have been interested in the interoperability between the semantic web and relational world, considered as one of the most used database management system until today, without physical transformation of data by elaborating a conversion context of their query languages (SQL and SPARQL). Several researches were particularly interested of SQL-to-SPARQL translation direction such as RETRO, SQL2SPARQL and others.

SQL2SPARQL method [2] operates in this light aiming to convert a classic and simple SQL queries into an equivalent SPARQL ones combining the transformation rules presented already in other works so as to realize a dynamic mapping. In addition, RETRO method [6] provides interoperability between relational database and RDF Stores by translating basic and composed SQL query (Union, Intersect and inner join in its simple form) to a semantically equivalent SPARQL one. In the same context, R2D method [8] proposes a mechanism integrating SQL-to-SPARQL translation by converting SQL queries, with pattern matching and aggregation, into the SPARQL equivalent ones. Similarly, the researchers of [1] explain their proposing approach of

interrogating RDF data using SQL queries via an algorithm that convert each clause of SQL queries (simple and complex ones without outer join) into an equivalent SPARQL ones.

Based on the previous analysis, we note that all these approaches have the same and common weaknesses: firstly, they operate just on simple SQL input queries without any consideration of outer join operations. Furthermore, the first work established in SQL-to-SPARQL conversion direction that takes into consideration the outer join in their simple and nested form is presented in [11]. Secondly, all existing solutions convert directly the input SQL query to its equivalent SPARQL one without any pre-processing phase enabling the optimization of outer joins and avoiding unnecessary operations during the conversion process.

To the best of our knowledge, this paper is the first work developed in this topic treating a detailed simplification of the input SQL queries before starting the SQL-to-SPARQL transformation process, more precisely; the simplification of outer joins clauses generating OPTIONAL patterns responsible of the high complexity of SPARQL queries.

III. Preliminaries

This section introduces our work by giving some theoretical background of the different operations used in this optimization approach.

A. SQL Outer Join

In relational terms, the data is distributed among several tables. In fact, we can imagine a table containing a foreign key to another one. In this case, we need to use joins in order to retrieve information from these both tables in a single query and exploit the full power of relational data-bases to achieve results that combine data efficiently from multiple tables. Joins are a powerful construction of SQL language, but they have to be handled carefully, because a small missed join can easily broke down a database server.

Outer join is one of relational join types used via SQL language aiming to regroup data from two or more tables by returning all rows from at least one of these tables indicated in FROM clause. We can distingue three types of outer joins as illustrated in Fig. 1:

- Left Outer Join: returns all rows from the left table and the matched rows from the right table.
- Right Outer Join: returns all rows from the right table and the matched rows from the left table.
- Full Outer Join: return all rows of left and right tables when the join condition is respected.

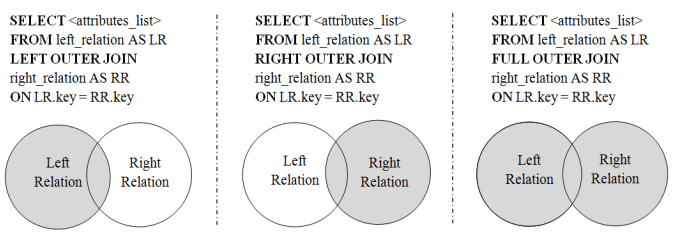

Fig. 1. Descriptive Schema of SQL Outer Joins.

The Left and right outer join are called one-sided outer joins because they preserve only one relation of its joined ones. Whereas, the full outer join is called two-sided outer join [12] because it preserves information from both relations.

As an example, at database level, we suppose that we have two tables: Customer and Order. In fact, we aim to conceive a list of customers with the total amount of their orders by joining these two tables with a left outer join operator so as to return all customers whether they placed any order or not. The final SQL query is conceived as follows:

```
SELECT Customer.fullName, Order.amount
FROM Customer
LEFTOUTERJOIN Order
        ON Customer.cust_id = Order.ord_id
```

B. SPARQL Optional Operator

The main part of SPARQL SELECT query is specified using a graph pattern. In fact, several studies show that 45% of SELECT queries are specified only by graph patterns [7]. They allow users and applications to query RDF data where the entire query pattern must match for there to be a solution. However, RDF has a semi-structured data; this is why SPARQL [5] is able to make queries that allow information to be added to the solution where the information is available, but they do not fail when some part of the query pattern does not match.

In order to realize the previous aim, SPARQL language uses the Option-al operator that combines a pair of graph patterns so as to extend the solution if the patterns matching have been succeeded. Else, the whole query does not fail if the optional pattern match fails.

The equivalent SPARQL query of the previous SQL one is given below:

```
SELECT ?full_name ?amount
WHERE {
?cust_id :Customer ?full_name.
  OPTIONAL {  ?cust_id :Order ?amount   }}
```

IV. Optimization Rules

In this section, we describe the main rules using by our optimization layer in order to simplify SQL queries with outer join operations (in this paper, we consider all outer join types).

Before starting the optimization process, as mentioned in the previous work [11], the input SQL query (having left outer join else we convert the other join types to a left one) has to respect some semantic rules:

- Condition 1: checking the validity of the join condition. In fact, shared variables between join relations must be bound to the same values.
- Condition 2: ensuring that the left outer join succeed by verifying the columns of the right relation which must in no case be all nulls, else the left outer join is reduced to a simple selection of the left table elements. For example, if this condition is not verified for SELECT * FROM R1 LEFT OUTER JOIN R2 ON (R1.a = R2.a),

our system will reduce it to a simple selection as SELECT * FROM R1.

- Condition 3: Before the evaluation of the main Left Outer Join clause, all containing clauses have succeeded. Hence, the attributes of their right relations must be NOT NULL.

Regarding the optimization process, we have used a set of algebraic equivalence rules based on null-rejected condition so as to simplify, whenever possible, the outer join operation and avoid generating Option-al patterns after the SQL-to-SPARQL conversion process.

In fact, a condition is said null-rejected in attribute set A for an outer join operation if it evaluates to False or Unknown on every tuple in which all attributes in A are null.

We consider the previous tables Customer and Order. For example, in the following query (Eq. 1) that aims to make a list of all customers who lives in Casablanca city:

$$\sigma \text{ City} = \text{‘Casablanca Customer’} \quad (1)$$

The condition on customer’s city (City = ‘Casablanca’) reject nulls on the attribute City and on any superset of City (*sch(Customer)*).

In this study, we use relational operators on a condition C such as a selection (σ_C), left outer join ($=\!\!\triangleright\!\!\triangleleft$), right outer join ($\triangleright\!\!\triangleleft\!\!=$) and full outer join ($=\!\!\triangleright\!\!\triangleleft\!\!=$). The Inner join, returning records that have matching values in both tables, is denoted. The set of attributes referenced by a condition C is called the schema of C, and denoted *sch(c)*.

A. Left Outer Join Simplification

The Left outer join operation can be converted to an inner join one if and only if the WHERE condition is null-rejected on the right relation schema (Eq. 2 in rule 1), else we use the second rule (Eq. 3) to guarantee this equivalence that is presented as a union of the Inner Join and Minus be-tween the both joined relations.

Rule 1: if C_1 is null-rejected on $sch(C_1) \subseteq sch(R_2)$, then:

$$\sigma_{C1}(R_1 =\!\!\triangleright\!\!\triangleleft_{C2} R_2) := \sigma_{C1}(R \triangleright\!\!\triangleleft_{C2} R_2) \quad (2)$$

Rule 2:if the previous condition is not checked, then:

$$R_1 =\!\!\triangleright\!\!\triangleleft_{C2} R_2 := (R_1 \triangleright\!\!\triangleleft_{C2} R_2) \cup (R_1 \setminus R_2) \quad (3)$$

It is not necessary that the condition of null-rejecting is checked just on relation schemas, but also on any set of attributes satisfying the conditions.

B. Right Outer Join Simplification

Regarding the Right Outer Join operation, its simplification (Eq. 4) is obtained by permuting the joined relation so as to have a left outer join expression and continue to use the rules 1 and 2 defined in the previous paragraph.

Rule 3 :

$$\sigma_{C1}(R_1 \triangleright\!\!\triangleleft_{C2} R_2) := \sigma_{C1}(R_2 \triangleright\!\!\triangleleft_{C2} R_1) \quad (4)$$

C. Full Outer Join Simplification

Regarding the Full Outer Join, the test is applied to each side of operation. In fact, if the null-rejected condition is checked on R1 schema then the operation is reduced to a left outer join one (Eq. 5 in rule 4), else if it is checked on R2 schema then we replace the full join with the right outer join (Eq. 6 in rule 5).

Rule 4: if C_1 is null-rejected on $sch(C_1) \subseteq sch(R_1)$, then:

$$\sigma_{C1}(R_1 =\!\!\triangleright\!\!\triangleleft_{C2} R_2) := \sigma_{C1}(R_1 \triangleright\!\!\triangleleft_{C2} R_2) \quad (5)$$

Rule 5: if C_1 is null-rejected on $sch(C_1) \subseteq sch(R_2)$,

$$\sigma_{C1}(R_1 =\!\!\triangleright\!\!\triangleleft_{C2} R_2) := \sigma_{C1}(R_1 \triangleright\!\!\triangleleft_{C2} R_2) \quad (6)$$

After performing the simplification of two-sided outer join to one-sided outer join, we process with the same manner as left and right ones.

V. Strategy Overview

A. Functional Architecture

In order to avoid direct transformation of input SQL queries to the equivalent SPARQL ones in existing conversion systems, we have proposed to add an optimization layer to these mapping systems in order to bridge the previous gap by conceiving a functional architecture, schematized in Fig. 2 and composed of five components: Query Analyser & Corrector, Is Null Rejected, Full Outer Join Simplifier, Right to Left Outer Join Converter and Outer Join Optimizer.

B. Optimization Algorithms

Query Analyser & Corrector: This step is very helpful especially in the case when the SQL queries were built based on user input that can be scanned and analyzed in order to correct syntactic errors, if they exist, before starting the optimization process.

Outer Join Optimizer: This component is considered as the main one in our proposed architecture. In fact, it operates on an SQL query filled by users and having an outer join clause in order to return the optimized SQL query at the end of Algorithm 1.

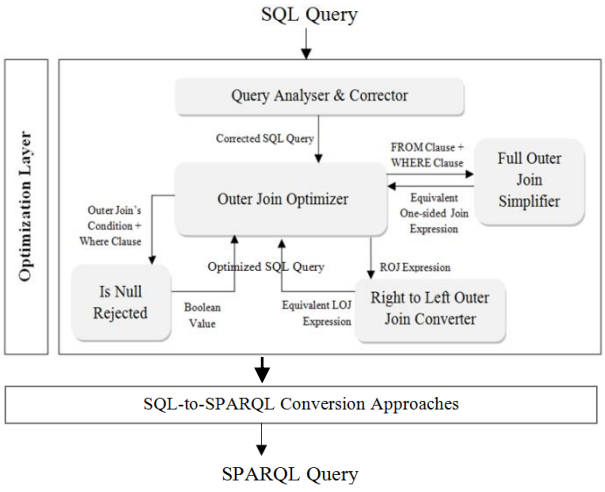

Fig. 2. Functional Architecture of our Optimization Layer.

Firstly, our algorithm extracts each clause of SQL query separately by parsing it to a binary tree, and then we check if the FROM clause contains full outer join expression then we call FullOuterJoinSimplifier in order to replace it by a one-sided join whenever possible. Else if the FROM clause contains a right outer join expression then we use the Right2LeftOJConverter component aiming to convert this expression to a Left outer join one before starting the optimization process using the rules defined in the previous section.

Secondly, we check if the where clause contains a null-rejected condition using the sub component *IsNullRejected* described subsequently, then we replace the one-sided outer join operator with the inner join one; Else we modify the FROM clause before returning the output SQL query.

Algorithm 1: Outer Join Optimizer

Input: SQL query with outer join clause(s), q_{in}
Output: Optimized SQL query, q_{out}
Begin
Tree SQLtree ← parse(q_{in})
q_{in}^{SELECT} ← SQLtree.getSelectClause()
q_{in}^{FROM} ← SQLtree.getFromClause()
q_{in}^{WHERE} ← SQLtree.getWhereClause()
Operator ← q_{in}^{FROM}.getJoinOperator()
if(IsFullOuterJoin(Operator) = True) **then**
/*Replace the FOJ by a one sided OJ whenever possible*/
q_{in}^{FROM}← FullOuterJoinSimplifier(q_{in}^{FROM}, q_{in}^{WHERE})
End if
If (IsRightOuterJoin(Operator) = True) **then**
q_{in}^{FROM}← Right2LeftOuterJoinconverter(q_{in}^{FROM})
End if
/*we check if the where clause contains a null rejected condition*/
R_R ← q_{in}^{FROM}.getRightRelation()
/* LAR_R : List of the right relation attributes*/
LAR_R ← parseList(R_R)
If (IsNullRejected(q_{in}^{WHERE}, LAR_R) = True) **then**
/*Replace the one-sided outer join operator with the inner join*/
q_{in}^{FROM}.setJoinOperator('INNER JOIN')
Else
R_L ← q_{in}^{FROM}.getLeftRelation()
R_R ← q_{in}^{FROM}.getRightRelation()
q_{in}^{FROM} ← q_{in}^{FROM} + 'UNION' + R_L + 'MINUS' + R_R
End if
q_{out} ← q_{in}^{SELECT} + q_{in}^{FROM} + q_{in}^{WHERE}
Returnq_{out}
End Algorithm

Full Outer Join Simplifier This is the main component in our system as presented in the Algorithm 2. In fact, it takes as input an SQL query qin containing an outer This sub component takes as input the Full outer join expression and the where clause of the SQL query in order to return the equivalent one-sided join expression. Firstly, we extract the left and right relations of the input join expression and then we parse them to a list of attributes for an ulterior use. In the next step, we check if the null-rejected condition is checked on the left relation schema then the operation is reduced to a left outer join one (Rule 4), else if it is verified on the right relation schema then we replace the full join with the right outer join (Rule 5).

Algorithm 2: Full outer join simplifier

Input: Full outer join expression (Exp_{in}), Where conditions (WhereClause)
Output: Equivalent one-sided join expression, Exp_{out}
Begin
R_L ← Exp_{in}.getLeftRelation()
R_R ← Exp_{in}.getRightRelation()
OnCond ← Exp_{in}.getOnCondition()
$schR_R$ ← parseList(R_R)
$schR_L$ ← parseList(R_L)
If (IsNullRejected(WhereClause, $schR_R$) = True) **then**
Exp_{LOJ} ← R_R + "Right Outer Join" + R_L + OnCond
Elseif(IsNullRejected(WhereClause, $schR_L$) = True)**then**
Exp_{LOJ} ← R_R + "Left Outer Join" + R_L + OnCond
End if
Return Exp_{LOJ}
End Algorithm

Is Null Rejected Takes as input the where clause and the attribute list of one joined relation in order to test if this clause contains a null-rejected condition or not on the given schema relation (the second input) and return a Boolean value. Algorithm 3 presents below the detailed instructions to realize this goal.

Algorithm 3: Is Null Rejected

Input: Where condition (WhereClause), Set of relation attribute ($AttrList_R$)
Output: Response as aBoolean value
Begin
int i ← 0
List WCL ← parseList(WhereClause)
while (i < WCL.size()){
cond ← WCL[i]
attribute ← cond.getAttribute()
if (attribute.isIncluded($AttrList_R$) = True) **then**
If (cond is like a joined tables attribute with 'IS NOT NULL' condition
OR
cond is evaluated to False or Unknown for the generated null tuples) **then**
Return True
End if
end if
i ← i +1
End while
Return False
End Algorithm

Right to Left Outer Join Converter the main aim of this sub-component is to convert a ROJ expression to an equivalent LOJ one applying this swapping ruleRelation_L Right Outer Join $\text{Relation}_R \Leftrightarrow$ Relation_R Left Outer Join Relation_L as presented in Algorithm 4 presented below.

Algorithm 4: Right to Left Outer Join Converter

Input: ROJ expression, ROJexp
Output: Equivalent LOJ expression, LOJexp
Begin
LeftRelation = ROJexp.getLeftExpression()
RightRelation = ROJexp.getRightExpression()
OnCond ← ROJexp.getOnCondition()
LOJexp ← RightRelation + 'Left Outer Join' + LeftRelation + OnCond
Return LOJexp
End Algorithm

VI. Implementation

In order to improve the effectiveness of our proposed approach, we have developed a java application offering to relational users an efficient tool to query RDF stores with optimized SPARQL queries. The experiments were carried out on a PC with 8 GB RAM, intel Xeon X7460 2.7 GHz.

We present below some examples of SQL queries supported by our system and its equivalent SPARQL ones obtained by a direct conversion and optimized one. In the example illustrated in Fig. 3, we have operated on an SQL query with left outer join aiming to select researchers having supervisors. Thus, the equivalent SPARQL query is composed obviously of an Optional pattern that corresponds to the left join in the input query. However, if we analyze deeply the input query, we note that it contains a null rejected condition (Supervisor.sup_id IS NOT NULL) in where clause that leads to transform left outer join operation to an inner join one (Rule 1). Consequently, the SPARQL equivalent query conceived from the optimized SQL one is composed by a simple graph pattern instead of an optional pattern responsible of the high complexity of SPARQL queries (Fig. 4).

In order to compare our optimized approach with others converting SQL queries to SPARQL ones without any preprocessing phase, we have used JENA framework and an RDF file with 5 million triples to execute a set of SPARQL queries generated by our application via the direct and optimized way, and then we have compared the execution time of each one. The comparison results are summarized in Fig. 5, which ensure the effectiveness of our proposed work that generates optimized SPARQL queries as shown in the diagram.

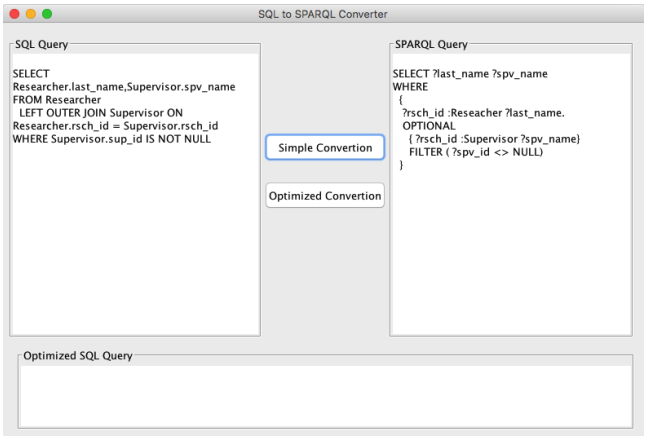

Fig. 3. Direct Conversion Example of SQL Query with Left Outer Join.

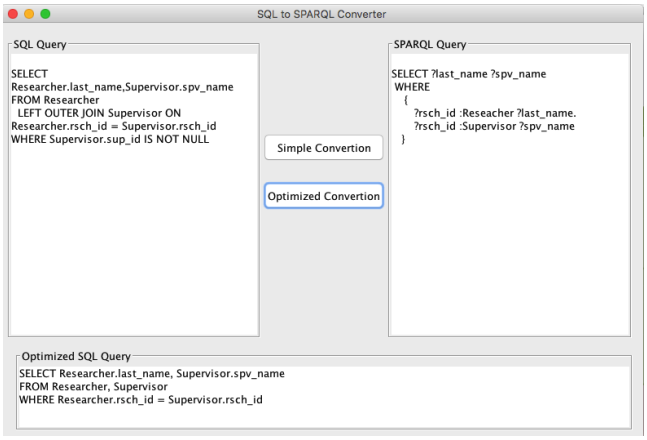

Fig. 4. Optimized Conversion Example of SQL Query with Left Outer Join.

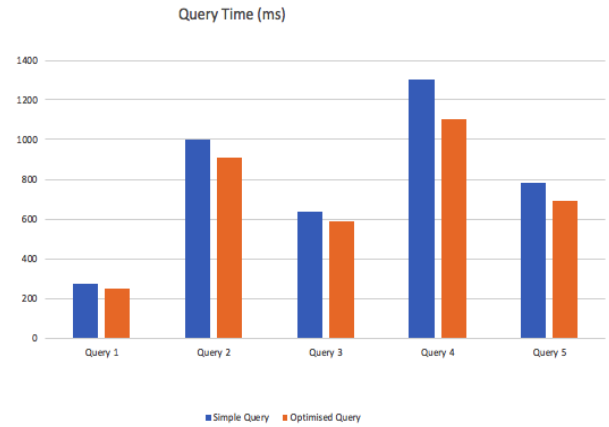

Fig. 5. Diagram of SPARQL Queries Execution Time via the Simple and Optimized Conversion Mode.

VII. CONCLUSION

In this paper, we have contributed in the enhancement of existing SQL-to-SPARQL conversion approaches by adding an optimizer and pretreatment layer to these systems in order to simplify all outer join types (left, right and full) in SQL queries aiming to avoid the generation of optional pattern(s), whenever possible, in the output SPARQL queries responsible of its high complexity. In addition, we have implemented our algorithm in order to improve its performance on a real data and make our strategy easily and effortless exploitable by the target audience.

The major limitation of our approach that is based only on null rejected properties of join condition so as to simplify SQL queries and avoid the generation of optional pattern(s), whenever possible, in the output SPARQL queries responsible of its high complexity.

In the future work, an obvious extension of the approach is to add more simplification rules to our Optimizer component and to integrate this framework into the relational database management system in order to offer relational users a direct and optimized extension to the semantic world.

REFERENCES

[1] Alaoui, L., Abatal, A., Alaoui, K., Bahaj, M., & Cherti, I. (2015, July), "SQL to SPARQL Mapping for RDF querying based on a new Efficient Schema Conversion Technique", International Journal of Engineering Research and Technology, Vol. 4, No. 10.

[2] Antal, M., Anechitei D., & Cuza, A. I. (2012), "SQL2SPARQL".

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

[4] Cyganiak, R., Wood, D., Lanthaler, M., Klyne, G., Carroll, J. J., & McBride, B. (2014), "RDF 1.1 concepts and abstract syntax", W3C recommendation, 25(02).

[5] Harris, S., Seaborne, A., & Prud'hommeaux, E. (2013), "SPARQL 1.1 query langu-age", W3C Recommendation, 21.

[6] Rachapalli, J., Khadilkar, V., Kantarcioglu, M., & Thuraisingham, B. (2011), "RETRO: a framework for semantics preserving SQL-to-SPARQL translation", The University of Texas at Dallas, 800, 75080-3021.

[7] Picalausa, F., & Vansummeren, S. (2011, June), "What are real SPARQL queries like? ", Proceedings of the International Workshop on Semantic Web Information Manage-ment, p. 7, ACM.

[8] Ramanujam, S., Gupta, A., Khan, L., Seida, S., & Thuraisingham, B. (2009), "In Seman-tic Computing", IEEE International Conference, pp. 303-311.

[9] Schmidt, M., Meier, M., & Lausen, G. (2010, March). "Foundations of SPARQL query optimization", In Proceedings of the 13th International Conference on Database Theo-ry ACM, pp. 4-33.

[10] Shadbolt, N., Berners-Lee, T., & Hall, W. (2006), "The semantic web revisited", IEEE intelligent systems, 21(3), 96-101.

[11] Soussi, N., & Bahaj, M. (2017), "Semantics preserving SQL-to-SPARQL query trans-lation for Nested Right and Left Outer Join", Journal of Applied Research and Tech-nology.

[12] Pérez, J., Arenas, M., & Gutierrez, C. (2006, November), "Semantics and Complexity of SPARQL", In International semantic web conference , Springer, Berlin, Heidelberg, pp. 30-43.