SQL2SPARQL4RDF: Automatic SQL to SPARQL Conversion for RDF Querying

Conference Paper · October 2019

DOI: 10.1145/3372938.3372968

4 authors, including:

Abatal Ahmed
Université Hassan 1er
26 PUBLICATIONS **93** CITATIONS
SEE PROFILE

Mohamed Bahaj
Hassan 1 st university Faculty of sciences and technologies settat morocco
182 PUBLICATIONS **1,411** CITATIONS
SEE PROFILE

SQL to SPARQL Conversion for Direct RDF Querying

Ahmed ABATAL[1], Khadija Alaoui[2], Mohamed Bahaj[4]
Mathematics and Computer Science Department
Hassan I University, Faculty of Sciences and Techniques
Settat, Morocco

Larbi Alaoui[3]
Mathematics and Computer Science Department
International University of Rabat
Sala Al Jadida Morocco

***Abstract*—With the advances in native storage means of RDF data and associated querying capabilities using SPARQL, there is a need to let SQL users benefit from such capabilities for interoperability objectives and without any conversion of the RDF data into relational data. In this sense, this work present SQL2SPARQL4RDF an automatic conversion algorithm of SQL queries into SPARQL queries for querying RDF data, which extends the previously established algorithm with relevant SQL elements such as queries with INSERT, DELETE, GROUP BY and HAVING clauses. SQL users are provided with a relational schema of their RDF data against which they can formulate their SQL queries that are then converted into SPARQL equivalent ones with respect to the provided schema. This avoids the birding of translating instances and data replication and thus saving loading times and guaranteeing fast execution especially in the case of massive amounts of data. In addition, the automatic mapping framework developed by the java programming language, and implement many new mapping functionalities. Furthermore, to test and validate the efficiency of the mapping approach and adding a module for automatic execution and evaluation of the various obtained SPARQL queries on Allegrograph.**

***Keywords*—Resource Description Framework (RDF); Structured Query Language (SQL); Simple Protocol and RDF Query Language (SPARQL); schema mapping; query conversion; Allegrograph**

I. Introduction

The relational database (RDB) systems have been used as a standard for data management for many years and involve the development of various tools. However, in recent years, data management solutions based on the ontology language RDF have proven to be a well-suited alternative to relational databases for the storage and querying RDF data. In the last decade, the use of RDFs has indeed evolved considerably and the amount of RDF data has increased enormously. Since the standardization of RDF and its query language SPARQL ([6], [7], [8]), RDF has found a growing interest for its use in many application domains such as biology [3], health [11], geology [15], smart cities [12], etc. This interest has also been accompanied with the development of various tools for handling RDF data. One of the motivations behind this development is the need for database solutions that are independent of existing relational databases (RDB) technologies in order to handle massive amount of data produced and to tackle the problems related to the limitations of the RDB technologies. To be also noticed, is that due to the simplicity of RDF and its power of presenting data in a machine-readable format many attempts have been done to convert huge amounts of relational data into RDF (e.g., [1], [4]). This was done with the aim to benefit from the opportunities RDF provides for integration purposes and for linking data to make it accessible for the semantic web. The RDF data model allows data to be structured in graphical form, which ensures flexible navigation through the use of well-established graphical algorithms. These characteristics of RDF have encouraged the development of various technologies for managing and querying RDF data.
For all these reasons, the problem that raises itself is how to let RDB users interact with RDF data without any conversion of data into relational data in order to facilitate sharing of RDF data across applications. Solutions to such a problem will also make it possible to take advantage of the multiple RDF dedicated stores and their associated SPARQL capabilities (e.g. Sesame [13], CliqueSquare [16], 4store [17], SOR [18], RDF-3X [19], SHARD [21], ...), avoiding thus the use of relational stores and their associated problems such as lack of adequacy to support dynamic RDF schemas or to support large amounts of RDF data.

This work provide a solution to this interoperability problem that consists of an extension of the SQL2SPARQL framework established in [10] for RDF querying using SQL without any conversion of RDF data into relational data. The framework provides RDB users with a relational schema that let them query RDF data using SQL. The SQL queries are translated to SPARQL queries that will be executed directly on RDF dataset.The extension give aims at presenting a complete conversion solution with algorithms for the automatic exchange of data with an execution of queries directly on RDF data stored in native systems. Such a solution avoids thus an extra data store and associated problems such as synchronization of data changes between two data stores and loading times of RDF data into relational stores.
The rest of the paper is organized as follows: Section II, he gives an overview of the existing mapping algorithms from SQL to SPARQL. Section III is devoted to the extension SQL2SPARQL4RDF of the SQL2SPARQL framework where various mapping algorithms are presented. Section IV deals with the implementation and tests of the framework for the validation of mapping algorithms. Section V concludes this work.

II. Related Works - SQL2SPARQL Conversion Framework

This section describes the SQL2SPARQL in the work [10] conversion framework and other related works. For the advantages of such a framework over other conversion techniques (e. g. [14],[20]), this work refer to the previous work in [10]. The SQL2SPARQL framework consists in converting SQL queries to SPARQL equivalents, which can be executed on real RDF datasets. SQL queries are formulated against a relational schema that the framework extracts from the RDF data and provides to SQL users.

The steps involved in the relational schema extraction are discussed in the following subsection. Those related to the query conversion according to this schema are detailed in the subsequent subsection.

A. Proposed System Architecture

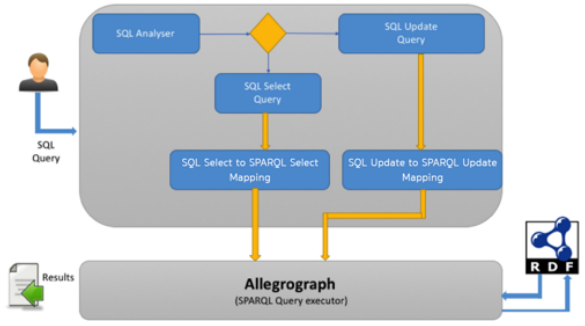

Fig. 1. System architecture

Fig. 1 present the proposed architecture that allows the user to give a complete SQL query and translates to equivalent SPARQL Query, in the first time SQL Analyzer that takes the SQL query as input, then it checks whether it is a SQL Select query or SQL update, after it passes the query to the corresponding component "SQL Select Query" or "SQL Update Query", to break down and extract the necessary elements like the Triple pattern or the attributes and the sending's to the component responsible for the translation of the requests via dedicating algorithms for the conversion then generates the final SAPRQL query to run it on "SPARQL Query Executor" the component that uses Allegrograph [2] and returns the result to User.

B. Relational Schema Extraction

The extraction of the relational schema in the SPARQL2SQL framework is provided by the schemaMapping()-algorithm given in Tables I and II, is based on the vertical partitioning work in[9]. It traverses all triplets and extracts unique predicates that are stored in variables to be used in the other algorithms of the framework. First, the property table method is used to group subjects with common predicates. In each table contains a subject as an attribute to identify the table and a collection of predicates attributes.

In addition, this work does not use the property table method, which allows us to think of a response with many fewer tables. The proposed solution reduces joint conditions and provides a more logical and efficient relational schema that makes it easier for users to query the schema.

TABLE I. Relational Schema Extraction / Part 1

Algorithm: Part 1 - schemaMapping()	
Input:	Bx : list of Abox
Output:	N: hashMap key is "relation name" and value is "predicate name"
1	N = "" {initialize N by null }
2	for i = 1 to Bx.size do
3	T= Bx[i] {extract a triple T from the Bx}
4	r = T.type extract type r from the triple T
5	p = T.predicate
6	extract predicate p from T
7	notExist = true
8	k = 1
9	while ((notExist) AND (j¡=N.size)) do
10	if ((r==N(k).getRelation() AND v==N(k).getPredicate())
11	notExist =false;
12	end if
13	end while
14	if (notExist)
15	N.put (r,v) { predivate p, and relation r to N }
16	end if
17	end for
18	Return N

TABLE II. Relational Schema Extraction / Part 2

Algorithm: Part 2 - schemaMapping()	
Input:	N: hashMap of Part-1
Output:	M : Relational schema
1	For each key v of N
2	Create a table M with an attribute SUBJECT
3	For each value k associated with v in N
4	add an associated attribute k to M
5	return M

C. Query Conversion

The conversion of SQL queries that are issued on the extracted relational schema into equivalent SPARQL queries is the task of the convertSqlQuery()-algorithm given in Table III. This algorithm accepts an SQL query as an input in string format and Map P as the output of the schemaMapping() algorithm, and produces an output in the form of a SPARQL query.

The algorithm analyzes the SQL query to extract the SELECT, SQwhereA and SQwhereB clauses, the corresponding SQwhereA has a clause where with join conditions and SQwhereB has a WHERE clause contains boolean conditions, the algorithm affects the NULL value in WHERE clause in the case this clause contains neither joins nor boolean conditions.

TABLE III. ALGORITHM FOR CONVERSION OF SQL QUERY

Algorithm: convertSqlQuery()	
Input	S :SQL query; N: hashMap
Output	SP :SPARQL Query
1	SP =""
2	QR=analyze(S) {analyze SQL query to obtain clauses }
3	Sselect= QR.getSelectClause()
4	SwhereA= QR.getWhereJC()
5	SwhereB= QR.getWhereBE()
6	SPselect="SELECT "
7	SPwhere="WHERE { "
8	T = "" { initialize T (Triple pattern) by NULL }
9	T = ConvSqlSelect(Sselect).getTP()
10	SPselect=ConvSqlSelect(Sselect)
11	SPwhere +=
12	ConvWhereSql(SwhereA, SwhereB,T,N)
13	SP= SPselect+ SPwhere+"}"
14	if QR.type!= null then
15	q1=QR.GetLeftSubSL()
16	q2=QR.getRightSubSL()
17	SP1=queryConvert(q1)
18	SP2=queryConvert(q2)
19	SP=combine(SP1, SP2, QR.type)
20	end if
21	Return SP

The convertSqlQuery() algorithm uses the subalgorithms ConvSqlSelect(), ConvSqlWhere() given in Tables VII and VIII. if the query contains UNION or INTERSECT, this query will be considered as two SQL queries. First of all, each one is converted by the convertSqlQuery() algorithm. The results of these queries are then grouped by the combinatorial algorithm () given in Table IV to give the final SPARQL query.

TABLE IV. ALGORITHM FOR CONVERSION OF A QUERY WITH COMBINATION OF TWO WHERE CLAUSES

Algorithm: Combine()	
Input:	query1 :SPARQL Query ; query2 :SPARQL Query ; type : {type is either INTERSECT, EXCEPT or UNION }
Output:	SP : SPARQL query
1	SP ="" initialize SP by empty
2	sprSelect= query1.getSelectClause()
3	sprWhere=" { ";
4	sprWhere1=" { "+query1. getSparqlWhere()+" }";
5	sprWhere2 = " { "+query2. getSparqlWhere ()+" }";
6	sprWhere +=sprWhere1+type+sprWhere2+" }"
7	SP += sprSelect+spWhere
8	return SP

Tables V and VI example shows a SQL query converting to SPARQL:

TABLE V. EXAMPLE FOR INTERSECT CONVERSION

SQL query:
(SELECT client.name from client) INTERSECT (SELECT supplier.name from supplier)
SPARQL query:

SELECT ?o0
WHERE {
{
?s0 ¡http://uhp.ac.ma/ontology/name¿ ?o0
}
INTERSECT{
?s0 ¡http://uhp.ac.ma/ontology/name¿ ?o0
}
}

TABLE VI. EXAMPLE FOR ORDER BY CONVERSION

SQL query:
SELECT client.name, order.date FROM client, order WHERE client.subject=order.client ORDER BY order.date
SPARQL query:

SELECT ?o0 ?o1
WHERE {
?s1 ¡http://uhp.ac.ma/ontology/client¿ ?s0.
?s0 ¡http://uhp.ac.ma/ontology/name¿ ?o0.
?s1 ¡http://uhp.ac.ma/ontology/date¿ ?o1.
}
ORDER BY ASC(?o1)

Conversion of the select clause

In the first version of the framework [10], the authors developed a "ConvSelectSql ()" algorithm that allows you to convert simple Select clauses into SPARQL Select. Now this framework has improved this algorithm in order to convert a complete Select clause into SPARQL.

The algorithm for the conversion of the select clause of the SQL query is the ConvSqlSelect() algorithm given in Table VII. It takes as input a SELECT clause, and gives as output a SPARQL SELECT clause with a triple pattern list TP, as mentioned in Table VII.

TABLE VII. ALGORITHM FOR CONVERSION OF THE SELECT CLAUSE

Algorithm: ConvSQLSelect()	
Input:	B: list of attributes of an SQL-Select query
Output:	SpSelect : SPARQL Select query ; T : list TP of triple patterns
1	SpSelect =""
2	T =""
3	for j = 1 to B.size do
4	r = B{j}.relation { Retrieve the relation r from attributes A}
5	p = B{j}.attribute { Retrieve the attribute p from A }
6	if p!='subject' then
7	SpSelect += "O" + j
8	tp={ ?Sj P Oj } { A triple pattern is constructed }
9	T.put (r, tp)
10	Else
11	SpSelect += "S " + j
12	tp={ ?Sj rdf:type r } { A triple pattern is constructed }
13	T.put (r, tp)
14	Endif
15	End for
16	Return SpSelect, T

Conversion of the where clause

The ConvSqlWhere() algorithm takes an SQL Where clause and the outputs of schema-mapping() and conSelect-Sql() algorithms to give an equivalent SPARQL clause. The algorithm is given in Table VIII and the addJCtoWhere() Method used by The ConvSqlWhere() algorithm is given in Table IX.

TABLE VIII. ALGORITHM FOR CONVERSION OF THE WHERE CLAUSE

	Algorithm: ConvSqlWhere()
Input:	LJC: List of Join conditions, LBE: List of boolean expressions,T : triple patterns
Output:	SpWhere :SPARQL WHERE clause
1	SpWhere = ""
2	if (LJC.isEmpty() && LBE.isEmpty()) then
3	for each t from T do
4	SpWhere += "?" + t.subject + " " + t.predicate +" ?" + t.object
5	end for
6	else if (! LJC.isEmpty) then
7	for each p from LJC do
8	lOp=p.LeftOperand;
9	rOp=p.RightOperand ;
10	SpWhere = addJCtoWhere(SpWhere,lOp,rOp);
11	end for
12	{if we have a Boolean conditions }
13	else if (! LBE.isEmpty) then
14	for each e from LBE do
15	lOp=e.LeftOperand;
16	t =T.get(lOp.relation)
17	SpWhere += "?" + t.subject + " " + t.predicat + "?" + t.object
18	End for
19	{ add FILTER in SpWhere }
20	SpWhere += "FILTER("
21	for each k from LBE do
22	lOp=k.LeftOperand;
23	rOp=k.RightOperand;
24	t =T.get(lOp.relation)
25	SpWhere += t.object+"" + e.operator + "" + rOp
26	End for
27	SpWhere+=+")"
28	Endif
29	Return SpWhere

TABLE IX. METHOD FOR ADDING JOIN CONDITIONS TO A CONVERTED WHERE-CLAUSE

	Algorithm: addJCtoWhere()
Input:	TP : Triple patterns, lOp: left operand ;rOp: right Operand
Output:	SpWhere : SPARQL WHERE clause
1	tr1 =TP.get(lOp.relation)
2	tr2 =TP.get(rOp.relation)
3	SpWhere += "?"+tr1.subject+""+ tr1.predicat + "?"+tr1.object
4	if lOp.relation = rOp.attribut then
5	SpWhere+="?"+tr1.subject+""+rOp.attribut+" ?"+tr2.subject + "."
5	+ " ?"+tr2.subject +""+rOp.attribut+" ?"+tr1.object+". "
6	elseif (lOp.attribut = rOp.relation) then
7	SpWhere+="?"+tr2.subject+" "+lOp.attribut+" ?" + tr2.subject + "."+
7	" ?"+tr1.subject+""+lOp.attribut+" ?"+tr2.object+". "
8	else
9	SpWhere+="?"+tr1.subject+" "+lOp.attribut+" ?"+tr1.object +
10	"."+ " ?"+tr2.subject +" "+ rOp.attribut+" ?"+tr1.object+". "
11	end if
12	return SpWhere

III. SQL2SPARQL4RDF: EXTENSION OF SQL2SPARQL

This work has improved the SQL2SPARQL framework to convert a complete SQL query of selection including all the clauses (group, classify by ...) and the functions of aggregate (MIN, MAX, AVG, COUNT), / in the second part this extension is able to convert INSERT and DELETE SQL queries into SPARQL INSERT and UPDATE queries, this part introduces pseudo-codes of algorithms

A. Conversion of Select Clauses for Aggregate Functions

The exConvSelectSql () algorithm is the improvement of the convSelectSql () algorithm so that it can detect and extract aggregate functions:

TABLE X. EXTENDED VERSION OF THE SELECT-CLAUSE CONVERSION ALGORITHM

	Algorithm: exConvSqlSelect()
Input:	N :The list of attributes of an SQL Select
Output:	SpSelect: SPARQL Select, TP: triple patterns
1	SpSelect =""
2	TP =""
4	for j = 1 to N.size do
5	r =N{j}.relation extract r from attributes N
6	p=Nj.attribute { extract p from N }
7	if p!='subject' then
8	if isAggregate(N{j})
9	SpSelect += getAggregate(N{j})+"(o" + j+})+")"
10	end if
11	else SpSelect += "o" + j
12	tp={ ?si p oi } { triple pattern generated }
13	TP.put (r, tp)
14	else
15	if isAggregate(N{j})
16	SpSelect += getAggregate(N{j})+"(s" + j+})+")"
17	else SpSelect += "s " + j
18	end if
19	tp={ ?si rdf:type r } {triple pattern generated }
20	TP.put (r, tp)
21	endif
22	endfor
23	return SpSelect, TP

In the exConvSqlSelect() algorithm given in Table X, the isAggregate() function checks each attribute in the select clause contains an aggregate function (MIN, MAX, AVG, ...) if it exists he calls the getAggregate () function before adding the attribute to SPARQL Select to convert to the proper aggregate function.

B. Conversion of GROUP BY and HAVING clauses

The algorithm for the conversion of GROUP BY clause is given in Table XI.

TABLE XI. ALGORITHM FOR CONVERSION OF SQL GROUP BY CLAUSE

	Algorithm: ConvSqlGroupBy()
Input:	G an attribute of SQL-GroupBy query ,TP triple patterns
Output:	SPARQL Group By,TP triple patterns
1	groupBy = ""
2	tp=""
3	if(TP.contains(G.relation) then
4	tp = TP.get(G.relation,G.attrubit)
5	else
6	{ Extract p from A }
7	if G.attributeÊ!='subject' then
8	s = TP.getSubject(G.relation)
9	p = G.attribute
10	o = "o"+TP.getLastObject()+1
11	tp = { ?si p oi } { triple pattern generated }
12	TP.put (r, tp)
13	groupBy = "Group By "+tp.getObject();
14	else
15	s = TP.getSubject(G.relation)
16	o = "o"+TP.getLastObject()+1
17	tp = { ?s rdf:type ?o} { triple pattern generated }
18	TP.put (r, tp)
19	groupBy = "Group By "+tp.getSubject();
20	endif
21	end if
22	return groupBy, TP

.

.

.

Table XII presents the algorithm for converting HAVING clauses.

TABLE XII. ALGORITHM FOR CONVERSION OF SQL HAVING CLAUSE

Algorithm: ConvSqlHaving()	
Input:	H having condition of SQL-Having query ,TP triple patterns
Output:	SPARQL Having
1	having = ""
2	tp=""
3	{extract left and right operand from H}
4	p1 = H.getLeftOperand()
5	p2 = H.getRightOperand()
6	att = p1.getAttrubitFromAgregat();
7	tp = TP.get(att.relation,att.attrubit)
8	agg = H.getAgregat ();
9	if att.atribut="subject"then
10	having = "HAVING(+agg+"("+tp.subject+")" +H.operator+p2
11	else
12	having = " HAVING(+agg+"("+tp.object+")"+H.operator+p2
13	end if
14	return having

C. Conversion of INSERT Queries

Both getTP () and generateInsert () algorithms given in Tables XIII and XIV are needed to convert an INSERT SQL query. The first algorithm traverses an INSERT query to extract all the attributes and their values, to store them in a S hashmap, then the GenrateInsert () algorithm that takes the result of getTP () and generates an equivalent SPARQL insert.

TABLE XIII. EXTRACTION OF PREDICATES AND VALUES FROM SQL QUERIES

Algorithm: getTP()	
Input	Input: sql : SQL Insert Query
Output	Output: S : Map of predicates and values
1	A= sql.getAttributes() // return all attributes from INSERT
2	V= sql.getValues() //return values from INSERT
3	S={ };
4	for j = 1 to A.size do
5	S.put(A.value(j), V.value(j));
6	end for
7	return S

TABLE XIV. INSERT CONVERSION

Algorithm: generateInsert()	
ínput	A : Map of predicates and values
Output	SPARQL Insert Query
1	sparql= "INSERT DATA \n" ;
2	for i =1 to A.size do
3	sparql += V.value(0) +" "+V.getValue(i) +" "+A.getValue(i)+". \n";
4	end for
5	return sparql

D. Conversion of DELETE Queries

The conDeleteSql () algorithm given in Table XV was called to convert a SQL DELETE to SPARQL DELETE, which analyzes the SQL deletion request to extract the BE Boolean conditions, and generate the triple patterns TP.

TABLE XV. CONVERSION ALGORITHM FOR DELETE QUERIES

Algorithm: ConvDeleteSql()	
Input:	SQL : an sql delete query
Output:	SPARQL DELETE query
1	sparql = "DELETE { /n "where = " WHERE { /n"
2	BL= sql.getBE() // get Boolean conditions
3	TP = sql.getTP() // generate TP from SQL query
4	if (BL.isEmpty()) then
5	for tp in TP do
6	where += "?" + tp.subject + " " + tp.predicate +" ?" + tp.object
7	end for
8	{verification of booleans conditions }
9	else if (! BL.isEmpty) then
10	for e in BL do
11	p=e.getLeftOperand();
12	tp =TP.get(p.relation)
13	sparql += "?" + tp.subject + " " + tp.predicat +"?" + tp.object
14	where += "?" + tp.subject + " " + tp.predicat +"?" + tp.object
15	end for
16	{ add FILTER in query }
17	where += "FILTER("
18	for e in BL do
19	p1=e.getLeftOperand();
20	p2=e.getRightOperand();
21	tp =TP.get(p1.relation)
22	where += tp.object+"" + e.operator + "" + p2
23	endfor
24	sparql+="}"
25	where+=")"
26	endif
27	where+="}"
28	Return sparql + where

IV. IMPLEMENTATION AND TESTS

To verify the efficiency of the conversion framework, this work was implemented using java programming language and jena [5] with connection to Allegrograph[2] for data storage and query processing.

AllegroGraph [7] is a database and application framework for building semantic applications. It can store data in the form of triples, query these triples via SPARQL.

Fig. 2 shows a screenshot of the API execution platform for the case of the conversion of previously given SQL query with ORDER BY into an equivalent SPARQL query.

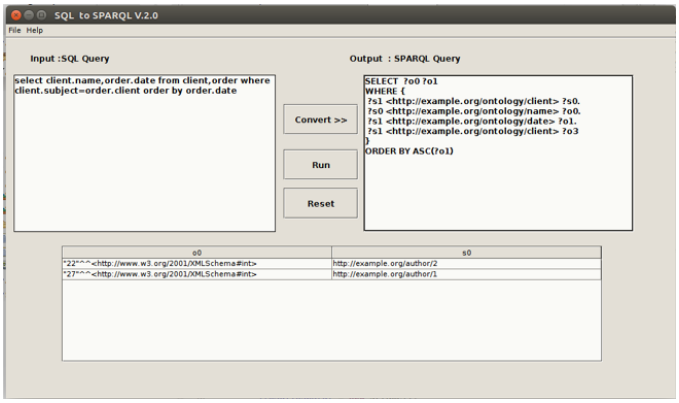

Fig. 2. Implementation of SQL2SPARQL4RDF Framework

The API also allows the execution on Allegrograph of the SPARQL query that is obtained through the conversion algorithm and the presentation of the execution result.

V. Conclusion

This paper present an extension of SQL2SPARQL framework already developed in the previous work for efficiently querying RDF documents using SQL queries expressed on the basis of a relational schema model without any conversion of extensions. With this framework, users who use SQL language no longer need to store data in a relational database, just store them in RDF data according to a schema mapping and SQL processors to execute their queries. They can also query large amounts of RDF data that are not possible to store in relational databases. They can also query a mass of RDF data that cannot be stored in relational databases. The considered extension dealt with the problems of converting SQL constructs related to INSERT, DELETE, GROUP BY and HAVING clauses. An associated java platform is also developed with a module for tests on the Allegrograph triplestore to prove the efficiency of this established conversion algorithms. In the future work, intend to optimize the conversion algorithms to apply them in web services to provide an API to make the mapping of SQL queries to equivalent SPARQL queries very easy to use in other platforms.

Acknowledgment

The authors would like to express thiere very great appreciation to there supervisors for thier valuable and constructive suggestions during the planning and development of this research work. Thier willingness to give thier time so generously has been very much appreciated. The authors would also like to extend thiere thanks to the colleagues of the laboratory of the mathematics and computer science department of Faculty of Sciences and Techniques, Hassan I University of settat, for their help and support.

References

[1] R2RML: RDB to RDF Mapping Language: https://www.w3.org/TR/r2rml/.

[2] Allegrograph: http://www.franz.com.

[3] Bioportal, http://sparql.bioontology.org/.

[4] Chhaya, P. et al.: Using D2RQ and Ontop to publish relational database as Linked Data. In: ICUFN. pp. 694–698 IEEE (2016).

[5] Kruti Jani, Dr. V.M. Chavda A Study on Semantic Web Framework: JENA and Protégé Indian Journal of Applied Research, Vol.IV, Issue. I - Jan 2014

[6] F. Alam, S. Ali, M. A. Khan, S. Khusro, A. Rauf, "A Comparative Study of RDF and Topic Maps Development Tools and APIs", BUJICT Journal, Volume 7, Issue 1, December 2014, pp. 1-12.

[7] W3C: SPARQL Protocol for RDF. http://www.w3.org/TR/rdf-sparql-protocol/.

[8] W3C: SPARQL Query Language for RDF. http://www.w3.org//TR/rdf-sparql-query/.

[9] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In VLDB, pages 411–422, 2007.

[10] L. Alaoui, A. Abatal, K. Alaoui, M. Bahaj and I. Cherti, (2015). SQL to SPARQL mapping for RDF querying based on a new Efficient Schema Conversion Technique. International Journal of Engineering Research and Technology, 4(10). IJERT.

[11] S. Anand and A. Verma, (2010). Development of Ontology for Smart Hospital and Implementation using UML and RDF. IJCSI Int. J. of Computer Science Issues, Vol. 7, Issue 5.

[12] P. Bellini and P. Nesi (2018). Performance assessment of RDF graph databases for smart city services. Journal of Visual Languages and Computing 45, 24–38.

[13] J. Broekstra, A. Kampman and F. van Harmelen (2002). Sesame: a generic architecture for storing and querying RDF and RDF schema. In: ISWC, pp. 54–68.

[14] E. I. Chong, S. Das, G. Eadon and J. Srinivasan (2005). An efficient SQL based RDF querying scheme. In: VLDB, pp. 1216–1227.

[15] G. Garbis, K. Kyzirakos and M. Koubarakis (2013). Geographica: a benchmark for geospatial RDF stores. In Proceedings of the 12th International Semantic Web Conference, 343–359.

[16] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. A. Quiané-Ruiz and S. Zampetakis (apr 2015). CliqueSquare: Flat plans for massively parallel RDF queries. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 771–782.

[17] S. Harris, N. Lamb and N. Shadbolt (2009). 4store: The design and implementation of a clustered RDF store. In Proceedings of the 5th Int. Workshop on Scalable Semantic Web Knowledge Base Systems, 16.

[18] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan and Y. Yu (2007). SOR: A practical system for ontology storage, reasoning and search. In: Proceedings of VLDB, pp. 1402–1405.

[19] T. Neumann and G. Weikum (2010). The RDF-3X engine for scalable management of RDF data. The VLDB Journal, 19(1):91–113.

[20] J. Rachapalli, V. Khadilkar, M. Kantarcioglu and B. Thuraisingham (2011). RETRO: a framework for semantics preserving SQL-to-SPARQL translation. In: EvoDyn Workshop.

[21] [21] K. Rohloff and R. E. Schantz 2010). High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In ACM Programming Support Innovations for Emerging Distributed Applications, 2010