SQL2SPARQL

Marius Antal[1], Daniel Anechitei[1]

[1] „Alexandru Ioan Cuza“ University, Faculty of Computer Science,
16, General Berthelot St., 700483, Iaşi
{daniel.anechitei, marius.antal }@info.uaic.ro

Abstract. The large use of World Wide Web applications resulted in big quantities of data which need intelligent software agents to access and present the required information in a rapid and automated manner. The Resource Description Framework is one such standard proposed by the W3C as solution to this problem. Since relational databases are still used and Semantic Web is still new a connection between them is useful and must be done. Regarding SPARQL to SQL translation some solutions exist and are used or proposed by some projects, like Jena, or UltraWrapCompiler. On the other hand, on rewriting from SQL to SPARQL there are many algorithms which have their weaknesses.

Our paper describes an alternative on converting SQL into SPARQL build combining some of the ideas already presented in other works. The solution aims and is focused on transforming a classic SQL query used with rigid but efficient relational database to programmers, but insignificant to machines, into an SPARQL query, one of the many languages which are used to query RDF data stores which have significance to machines. The SQL to SPARQL transformation is done using some dynamic mappings and transformation rules.

Keywords: Query Translation, RDF, Semantic Web, SQL, SPARQL.

1 Introduction

Our research attempts to create a tool to help on transition from relational databases to RDF data stores which are the foundation for Semantic Web. With the growing of RDF stores, and because tools available to RDF data are fewer and less mature than those for RDBMSs, relational models and their tools or technologies used for data management and visualization with their higher level of maturity should be still used until semantic web tools reach a bigger level of reliability than relational databases.

Another motivation for this paper is to help small organizations which can’t afford to spend so much on transition from relational databases to those used by Semantic Web. Related technologies and working methodology with relational databases are

things very popular. Also this tool wants to be a solution for those who move from relational to RDF database and want to jump over the part of accommodating with new syntax.

This paper contains a short description of our work, results, problems met, and some conclusions on SQL2SPARQL problem, which at first sight seemed not so complicated, but it revealed his difficulty after. The research result is this document and a tool written in Java which converts queries from SQL to SPARQL.

Also we created a short demo (a web application) which uses our tool to translate SQL queries into SPARQL. Our converter receives an SQL query and returns the SPARQL translation associated to it. All the translations done within the SQLtoSPARQL module are done at runtime using a combination of algorithms and dynamic mappings described in other researches which want to solve this problem like in [2] or [3] [4].

In order to illustrate the results achieved working at this project we created a demo web application which takes as input from user an SQL query and using our convertor returns an SPARQL query for it. In order check the SPARQL query resulted we also have the option to interrogate Dbpedia endpoint using the query resulted, which can be edited. In the backend the query from the interface is passed to the module which uses Jena for querying Dbpedia.

This paper contains a short description of the tools used for this research, the algorithm used to do the dynamic mapping presented with some case studies (example of some simple queries and some complex ones) , details of our application, some future directions to follow, and conclusions.

2 Short description of application

In order to build our solution we split our work on some directions, the main one SQL to SPARQL translation engine, the other two being, querying DBpedia(in order to check our results), and the web interface, for combining the translation engine and querying Dbpedia module. Our solution is based on technologies and tools like Java, JSF 2.0, Jena and w general overview is presented in Fig 1.

JSF 2.0 – Java Server Faces technology which establishes the standard for building server-side user interfaces and make web application development easier helped us build the interface of our demo application. The application runs on Tomcat application server. Our translation module is built using an open source SqlParser. Just for demo purposes we used Jena to create a module which can be used to test some of our resulted SPARQL queries. Jena is a powerful Java framework for building Semantic Web applications. Being structured as a collection of Java libraries Jena helped us link our application with Dbpedia endpoint.

As developing environment we used Eclipse Indigo (Fig 1). For collaborative working we used a google code svn solution and Tortoise.

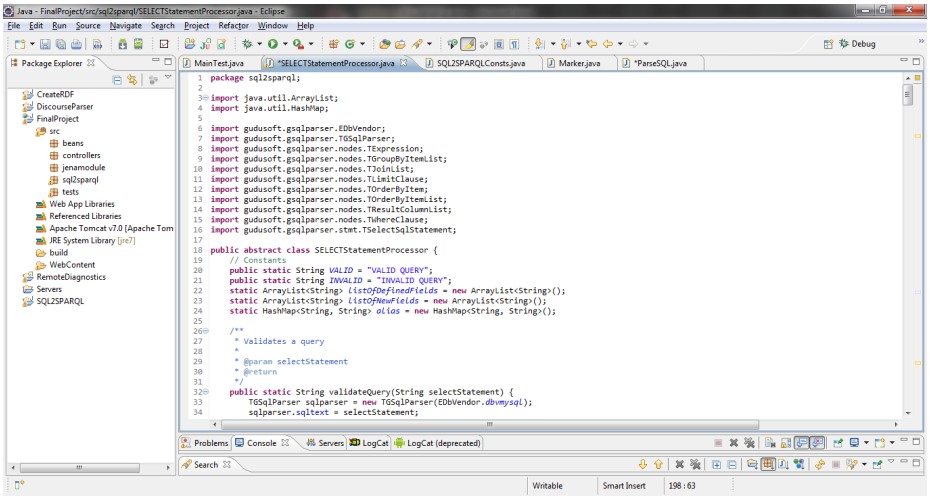

Fig. 1. Eclipse Indigo IDE

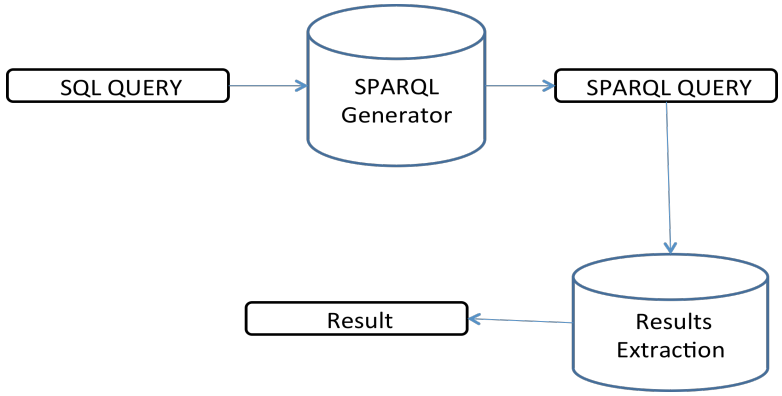

Fig. 2. General overview

3 RELATED WORK Description of existing solutions on SQL to SPARQL

In contrast to the main objective and research subject of our paper, SPARQL to SQL is already used by Jena to query relational databases. It seems that, in order to rebuild all the complex restrictions from an SPARQL query, Jena uses multiple SQL queries. However we didn't manage to get the sql resulted from an SPARQL query. Also UltraWrapCompiler is another open source project which purpose is to translate SPARQL queries into SQL language. Also, related to SPARQL to SQL are many solutions described at theoretically.

In the same area translating a relational database into a semantic form is not trivial, but with some minimal effort, can be done.

Although SQL to SPARQL transformations are poorly referenced, we manage to find some examples of algorithms which were our source of inspiration. Next we will present basic ideas of the works from which we extracted some ideas.

In "A Framework for the Relational Transformation of RDF Data" is provided a relational interface to data stored in the form of RDF triples and, being one of the first tries in this domain. In this paper authors propose a mapping scheme for the translation of RDF Graph structures to an equivalent normalized relational schema. The proposed mapping schema constructs and rules handle a variety of blank nodes and RDF Container objects such as Bags, Sequences, and Alternatives. Based on the RDF-to-RDBMS map file created, they propose a transformation process that presents a normalized, non-generic, domain-specific, virtual relational schema view of the given RDF store. Also authors propose a mechanism to transform any relational SQL queries issued against the virtual relational schema into the SPARQL equivalent, and return the triples data to the end-user in a relational format.

String matching procedures and aggregation facilities have been added to the algorithm in to arrive at the current algorithm. Also they say that they managed to create all of the above in the form of a JDBC interface that can be plugged into existing visualization tools.

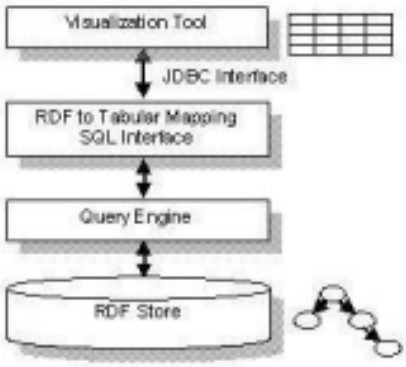

Fig. 3. General architecture of R2D

Worth mentioning that the work presented in "A Framework for the Relational Transformation of RDF Data" focuses on presenting, to the visualization tools, a tabular equivalent of the RDF triples database at the bottom of the above figure, and on an SQL Interface that generates SPARQL versions of SQL queries and passes the same to the Query Engine layer for processing and RDF data retrieval.

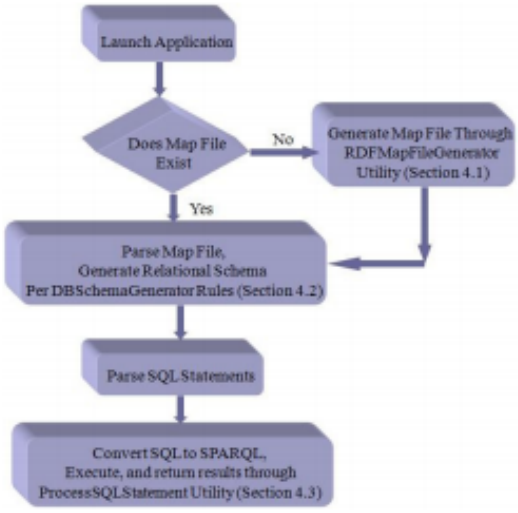

Fig.4. R2D Deployment Sequence

In other existing solutions the basic idea is almost the same. SQL query is parsed, and using some dynamic mappings the SPARQL query is build.

4 Build Prefixes

Prefixes are build using the join list from the SELECT query.

```
  private static String buildPrefixes(TSelectSqlStatement
select) {
    String SPARQLPrefixes = "";

    for (int i = 0; i < select.tables.size(); i++) {
      if
(SQL2SPARQLConsts.NAMESPACES.containsKey(select.tables
          .getElement(i).toString())) {
        SPARQLPrefixes += "PREFIX "
            + select.tables.getElement(i).toString()
            + ": <"
            +
SQL2SPARQLConsts.NAMESPACES.get(select.tables
                .getElement(i).toString()) + ">\n";
      } else {
        SPARQLPrefixes += "PREFIX "
```

```
            + select.tables.getElement(i).toString()
            + ": <"
            + SQL2SPARQLConsts.NAMESPACES
                .get(SQL2SPARQLConsts.DEFAULT_NS) + "/"
            + select.tables.getElement(i).toString() +
">\n";
      }
    }
    return SPARQLPrefixes;

      }
```

5 Process SELECT statement

Build the list of fields which are selected.

```
    for (int i = 0; i < listOfFields.size(); i++) {
      SPARQLSelect += " ?"
          + listOfFields.getElement(i).toString()
              .replaceAll("\\.", "_");

    String ss = listOfFields.getElement(i).toString();
    String sss[] = ss.split("\\.");

    SPARQLWhere += "\n\t?" + sss[0] + " " +
alias.get(sss[0]) + ":"
    + sss[1] + " ?" + ss.replaceAll("\\.", "_") + " .";

  listOfDefinedFields.add(listOfFields.getElement(i).toSt
ring());
    }
```

6 Process WHERE statement

The simplest WHERE statement is of type: (*variable operator value/ variable*). For text values SQL uses single quotes, although, numeric values should not be enclosed in quotes. Each 'variable' will be formatted as "?variable" or if we have an alias it will be "?alias_variable" in the SPARQL query.

WHERE firstName = 'Daniel'
is translated into
FILTER (?firstName = "Daniel")

WHERE age > 18
is translated into
FILTER (?age > 18)

WHERE firstName = lastName
is translated into
FILTER (?firstName = ? lastName)

Also AND & OR operators are used to filter records based on more than one condition. They will be replaced with "&&" and "| |" for SPARQL.

WHERE firstName = 'Daniel' AND age > 18
is translated into
FILTER (?firstName = "Daniel" && ? age > 18)

WHERE firstName = 'Daniel' AND lastName = 'Tom' OR lastName ='Alex'
is translated into
FILTER(?firstName = "Daniel" && ?lastName ="Tom" ||?lastName ="Alex")

By default the logical operators order is: first OR and second AND. But we can use brackets to decide the logical operators order and the parser will split the statement in two sub statements and recursively will create the SPARQL FILTER. The first sub statement is the left side expression of the operator and the second one is the right side expression of the operator. For the example above the first operator will be OR and the left side expression: firstName *= 'Daniel' AND lastName = 'Gigi'* and the right side expression: lastName *='Kent'.*

WHERE firstName = 'Daniel' AND (lastName = 'Tom OR lastName ='Alex')
is translated into
FILTER(?firstName ="Daniel" && (?lastName ="Tom" || ?lastName ="Alex"))

If we use brackets like in the above example, first operator will be AND with left side expression: *firstName = 'Daniel'* and the right side expression: *lastName = 'Tom' OR lastName ='Alex'*. In addition to "=", "<>", ">", "<", ">=", "<=" operators we have support for LIKE operator. In this case we must create a *regex* expression:

WHERE firstName LIKE 'Daniel'
is translated into
FILTER regex(?firstName,"Daniel"))

WHERE firstName = 'Daniel' OR firstName LIKE 'Daniel' AND age>=18

is translated into

FILTER(?firstName ="Daniel" || regex(?firstName,"Daniel") && ?age>=18))

We also have support for SELECT statements which columns are selected from multiple tables. For example:

```
SELECT p.nume , st.id , st.studii , st.grupa FROM persoana p ,
student st , sex s WHERE st.id = p.id and p.sex = s.cod and
s.nume = 'Alex'
```

Is translated into:

```
PREFIX persoana: <http://www.example.com/persoana>
PREFIX student: <http://www.example.com/student>
PREFIX sex: <http://www.example.com/sex>
SELECT ?p_nume ?st_id ?st_studii ?st_grupa
WHERE{
      ?p persoana:nume ?p_nume .
      ?st student:id ?st_id .
      ?st student:studii ?st_studii .
      ?st student:grupa ?st_grupa .
      ?p persoana:id ?p_id .
      ?s sex:cod ?s_cod .
      ?p persoana:sex ?p_sex .
      ?s sex:nume ?s_nume .
      FILTER(?st_id=?p_id && ?p_sex=?s_cod && ?s_nume="Alex")
}
```

While parsing the WHERE statement we also create a list of fields which were not included in SELECT statement. The algorithm is as follows:

```
private static String precessRecursiveWhere(TExpression
condition) {
String filter = "";
if (condition.getOperatorToken() == null) {

TExpression left = condition.getLeftOperand();
TExpression right = condition.getRightOperand();
```

```
String operator =
condition.getComparisonOperator().toString();

if (!listOfDefinedFields.contains(left.toString()))
      listOfNewFields.add(left.toString());

  String strRight = right.toString();
  if (strRight.contains("'")) {
            strRight = strRight.replaceAll("'", "\"");
          } else {
            if (isInt(strRight) == false) {
              if (!listOfDefinedFields.contains(strRight))
                listOfNewFields.add(strRight);

              strRight = "?" + strRight;
            }
          }

String strLeft = left.toString().replaceAll("\\.", "_");

    strRight = strRight.replaceAll("\\.", "_");
      if (!listOfDefinedFields.contains(left.toString()))
            listOfNewFields.add(left.toString());


filter += "?" + strLeft + operator + strRight;
return filter;

} else {
String operator =
condition.getOperatorToken().toString();
if (operator.equalsIgnoreCase("LIKE")) {
String[] lr = condition.toString().split(operator);
            String left = lr[0];
            String right = lr[1];
            left = left.substring(0, left.length() - 1);
            right = right.substring(1, right.length());

if (!listOfDefinedFields.contains(left.toString()))
          listOfNewFields.add(left.toString());

filter += " regex(?" + left.replaceAll("\\.", "_") +
",\"" + right.replaceAll("'", "") + "\") ";
} else {
```

```
TExpression left = condition.getLeftOperand();
TExpression right = condition.getRightOperand();

String recLeft = precessRecursiveWhere(left);
String recRight = precessRecursiveWhere(right);

if (operator.equalsIgnoreCase("or"))
operator = " || ";
else if (operator.equalsIgnoreCase("and"))
operator = " && ";

filter += recLeft + operator + recRight;
  }
 }
return filter;
}
```

7 Process GROUP BY statement

The GROUP BY statement takes the argument from the group by clause if it's found, and translates it into SPARQL.

8 Process ORDER BY statement

The ORDER BY being very simple is almost similar to GROUP BY.

9 Process LIMIT statement

The LIMIT statement was also easy to translate. If it's found in SQL SELECT statement we add a similar simple line in SPARQL version of the query.

10 Design of the application

The design of our demo application is very simple and efficient. Build in JSF, uses a text area component for the SQL to be translated and also another text area for the output SPARQL resulted query (in this way the sparql query can be corrected if the translation is not completely correct). Also another text area component is used to display the results of the sparql translation.

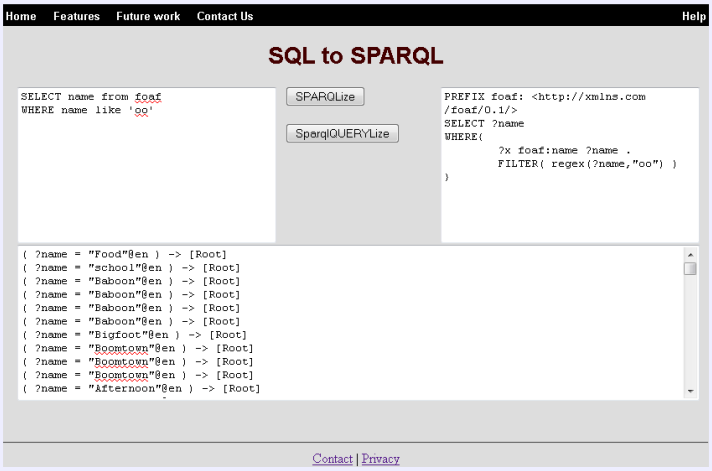

Fig.5. SQL to SPARQL interface

11 Conclusions and Further Work

In the present work we combined SQL parsing with SQL to SPARQL translation and also build an SPARQL test point for queries on *dbpedia*. Our contribution is the algorithm which makes the transformation from an SQL query into a SPARQL query. We done this rewriting using some dynamic mappings, based on input query. Despite the fact that there are not so much examples and works related to SQL to SPARQL transformation we managed to create a tool which parses and translates from basic queries to queries with a medium level of complexity. In future work we should expose a web service to our tool, and also treat the remaining cases of queries.

References

1. Brian McBride (2010) An Introduction to RDF and the Jena RDF API. http://jena.sourceforge.net/tutorial/RDF_API/
2. R2D: Bridging the Gap between the Semantic Web and Relational Visualization Tools
3. Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham: A Framework for the Relational Transformation of RDF Data
4. Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham: RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation
5. Bizer, C., Cyganiak, R., Garbers, J., and Maresch, O. The D2RQ Platform. http://www4.wiwiss.fu-berlin.de/bizer/d2rq/
6. RDF Vocabulary Description Language 1.0: RDF Schema. Feb, 2004. http://www.w3.org/TR/rdf-schema/
7. Tauberer, J. What is RDF. July, 2006. http://www.xml.com/pub/a/2001/01/24/rdf.html
8. Muys, A.: Building an Enterprise-Scale Database for RDF Data. http://www.netymon.com/papers/muysa06buildforrdf.pdf(2006)
9. Bizer, C., and Cyganiak, R. D2RQ – Lessons Learned. Position Paper. W3C Workshop on RDF Access to Relational Databases, 2007.
10. Bizer, C., and Seaborne, A. D2RQ – Treating Non-RDF Databases as Virtual RDF Graphs. 3rd International Semantic Web
11. Teswanich, W., Chittayasothorn, S.: A Transformation of RDF Documents and Schemas to Relational Databases. In: IEEE
12. PacificRim Conferences on Communications, Computers, and Signal Processing, pp. 38-41(2007)
13. Hendler, J.: RDF Due Diligence. http://civicactions.com/blog/rdf_due_diligence. (2006)