SQL2SPARQL

Marius Antal1, Daniel Anechitei1

1„Alexandru Ioan Cuza“ University, Faculty of Computer Science,

16, General Berthelot St., 700483, Iaşi

{daniel.anechitei, marius.antal [}@info.uaic.ro](mailto:%7d@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 “R2D: Bridging the Gap between the Semantic Web and Relational Visualization Tools” or “A Framework for the Relational Transformation of RDF Data” by Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Steven Seida, or "RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation" by Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham.

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.

1. 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. For collaborative working we used a google code svn solution and Tortoise.

**Fig. .** General overview

1. Process Where Clause

The simplest WHERE statement (field1 operator value/field2). For text values SQL uses single quotes. Although, numeric values should not be enclosed in quotes.

*WHERE firstName = ‘Daniel’*

*FILTER (?firstName = “Daniel”)*

*WHERE age > 18*

*FILTER (?age > 18)*

*WHERE firstName = lastName*

*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*

*FILTER (?firstName = “Daniel” && ? age > 18)*

*If(*

*WHERE firstName = ‘Daniel’ AND lastName = ‘Gigi’ OR lastName =’Kent‘*

*FILTER(?firstName = “Daniel” && ? lastName =”Gigi” || lastName =”Kent”)*

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 one is the left side expression of the operator and the second one the right side expression of the operator. For the example above the operator will be

*WHERE firstName = ‘Daniel’ AND (lastName = ‘Gigi’ OR lastName =’Kent‘)*

*FILTER(?firstName ="Daniel" && (?lastName ="Gigi" || ? lastName ="Kent"))*

*WHERE firstName LIKE ‘Daniel’*

*FILTER regex(?firstName,"Daniel" ))*

*WHERE firstName = ‘Daniel’ OR firstName LIKE ‘Daniel’ AND age>=18*

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

While parsing the WHERE statement we also create a list of fields which was not included in SELECT.

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;

}