TOWARD AUTOMATIC GENERATION OF SPARQL RESULT SET VISUALIZATIONS

A Use Case in Service Monitoring

Marcello Leida[1], Du Xiaofeng[2], Paul Taylor[3] and Basim Majeed[1]
[1]*EBTIC (Etisalat BT Innovation Center), Khalifa University, P.O. Box 127788, Abu Dhabi, U.A.E.*
[2]*School of Computer Science, University of Birmingham, Birmingham, U.K.*
[3]*BT Innovate & Design, Adastral Park, Martlesham Heath, Ipswich, U.K.*

Keywords: Data visualization, Ontologies, RDF, SPARQL, Rules, Logic reasoning, Service monitoring.

Abstract: The problem of representing RDF data using charts, dashboards, maps and so on has become pressing, in particular to prove the value of the Semantic Web to enhance the analysis of business data. State of the art solutions focus on mapping query results to a specific chart type or view and then manually writing the procedure that creates the final dashboard, but whenever a different visualization model is required, the mapping process needs to be repeated. In this paper we propose a semi-automatic approach that generates various charts from SPARQL queries over data represented as RDF graphs, we introduce and describe the generic approach and present a use case scenario in the context of service monitoring.

1 INTRODUCTION

The idea of the web as a huge knowledge base, composed of highly interconnected graphs, where arcs and nodes have their well-defined, machine understandable semantics is indeed extremely appealing and the growing interest around projects like *Linked Open Data (LOD)*[1] confirms this trend.

Enterprises have sensed the potential improvement in their Business Intelligence (BI) capabilities that the information in the Web could leverage: the use of virtually infinite information in the web that can be coupled with internal resources in order to improve the quality of analytical and reporting tools. The key challenge is how to present the information in a format that is understandable, and at an appropriate level of granularity to identify benefits and to inform strategic, tactical and operational decisions. Put simply, presenting the information in the manner that makes it the most useful for the target audience.

Ideally, a BI system should be able to take visualization requests and then generate appropriate dashboards on demand.

In this paper, we propose an ontology-based, semantic-aware, data annotation and visualization method to produce charts, diagrams and graphs from a SPARQL query. As use case we apply our method to produce dynamic dashboards concerned with Service Level Agreements (SLAs). Additional information is added to the query and to the visualization libraries in order to automatically or semi-automatically (in case more than one choice is possible) associate the appropriate visualizations to query results.

The paper is structured as follows: Section 2 presents the related work on the field of SPARQL query visualization. Section 3 is the main section of the paper, where our approach is presented. Section 4 presents an application of our technique to visualizing Service Monitoring information. The paper concludes with Section 5 where final considerations are presented together with the future directions.

2 RELATED WORK

Dynamic visualization of information is a very broad problem: we discuss the problem in the field of BI, however our approach is suitable for the visualization of SPARQL queries in any context.

Many solutions based on graphs, trees, tree maps, crop circles and similar are presented in detail in a survey (Katifori et al., 2007) which extensively describes implementations and different visualization

[1]http://linkeddata.org/

Leida M., Xiaofeng D., Taylor P. and Majeed B..
TOWARD AUTOMATIC GENERATION OF SPARQL RESULT SET VISUALIZATIONS - A Use Case in Service Monitoring.
DOI: 10.5220/0003487701810186
In *Proceedings of the International Conference on e-Business* (ICE-B-2011), pages 181-186
ISBN: 978-989-8425-70-6

techniques used to visualize RDF graphs. The problem with these techniques is that they are oriented towards a structural visualization of the graph and they have no practical utility in a BI environment, where the user expects to work with dashboards, charts and tables. As proposed in (Leida et al., 2010), there is a need for more sophisticated methods for visualizing SPARQL query result sets, in order to fully utilize the advantages of RDF data representation, especially its flexibility.

The only recent approach, which follows the research direction indicated by (Leida et al., 2010) is SPARQL Web Page (SWP) (Knublauch, 2010a). SWP is an RDF-based framework used to describe user interfaces for rendering semantic web data. It links RDF resources, extracted using SPARQL queries, with user interface descriptions that can be rendered as HTML. Together with its extension library *UISPIN Charts* (Knublauch, 2010b), SWP allows developers to bind the results of a SPARQL query into a set of a charts from Google Visualization API[2]. SWP is an important step toward the user-friendly presentation of Semantic Web data, especially in the field of SPARQL queries, where there is little previous work. However one of the major limitations of SWP is that requires specific code to be written to generate and display the desired chart. There is not an automatic or semi-automatic procedure to infer the most suitable chart from the query.

In Section 3 we present a system where for each SPARQL query, a set of suitable charts are generated automatically for the user to choose from.

3 AUTOMATIC APPROACH TO VISUALIZATION OF SPARQL QUERY RESULTS

In this section, we propose a method that is able to automatically infer the type of the results returned by a query in order to associate it with an appropriate visualization method for that specific result set.

The overall process is illustrated in Figure 1: the user, by interacting with the dashboard will generate a SPARQL query that will be analyzed and semantically annotated with information related to the types of the variables in the result set. This information is then used in an inference process which exploits these annotations in order to select a visualization method that suits the given result set. Once the visualization method has been selected, a server-side (application dependent) process will use the information generated in the previous steps (SPARQL result set and the visualization method) in order to create the final chart for the user.

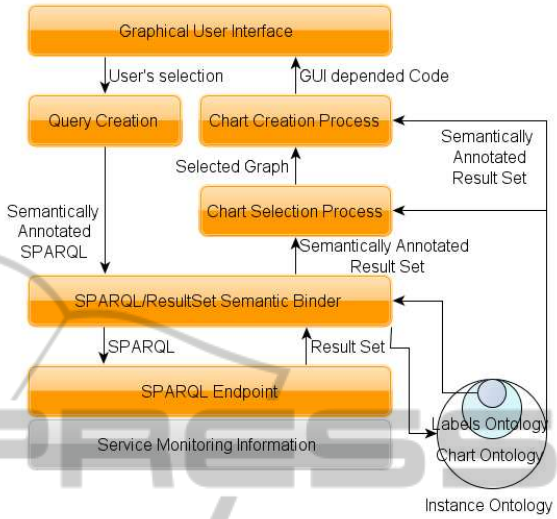

Figure 1: The proposed Chart Generation process flow.

3.1 Label Ontology

In order to automatically infer the type of the elements returned by a SPARQL query we need to provide a semantic description the result set returned by the query. This semantic description is defined as an ontology (the Label Ontology in Figure 1), which provides a semantically rich description of the variables in the query. The Label Ontology is a hierarchical model that has a top concept called `Label`, which identifies the most generic type of variable. The other concepts extend and specialize the `Label` concept: therefore we have `String`, `Data`, `Number`, `Currency`, `Country`, `Location` concepts.

In the fully automated case a matching function is used to analyze the result set and associate the variables with the appropriate concept from the Label Ontology. For example, if a regular expression was associated with each Label type, the matching function could link the variables that match these regular expressions with the related Label type.

3.2 Chart Ontology

After the matching process we describe now the Chart Ontology, which defines the core of our approach. It contains a concept `ResultSet` with a `{1:n}` relation (called `elements`) with the concept `Label` (and all its sub concepts) from the Label Ontology.

The concept `ResultSet` is linked with the concept `VisualizationMethod` from the Chart Ontol-

[2]http://code.google.com/apis/charttools/index.html

ogy, which has several sub-concepts defining different types of charts: `BarChart`, `PieChart`, `LineChart`, `Scatter`, `Map`, `TimeLine`; with attributes that define specific parameters for the visualization library used (Google Visualization API, Yahoo UI Charts [3] or Exibit[4]).

The main element of this ontology is the `visualizationCode` property associated with each `VisualizationMethod`. This property is extremely important because it contains the code template that our system will use to generate the final chart. The content of this property depends on the chosen library; as for example, in case of Google Visualization API, the value of `visualizationCode` for a vertical bar chart will be the template of a JavaScript function as shown in Table 1.

```
google.load('visualization', '1',
{packages: ['corechart']});
    function drawVisualization() {
  var data = new google.visualization.DataTable();
  var raw_data = [%{Start_Repeat}%[%{Label1.Value}%,
    %{Number.Value}%]%{End_Repeat}%];
  var label= [%{Label2.Value}%];
  %{Start_Repeat}%
  data.addColumn(%{Label1.Type}%,%{Label1.Value}%);
  %{End_Repeat}%
  for (var i = 0; i  < raw_data.length; ++i) {
    data.addColumn(%{Number.Type}%,raw_data[i][0]);
  }
  data.addRows(label.length);
  for (var j = 0; j < label.length; ++j) {
    data.setValue(j, 0, label [j].toString());
  }
  for (var i = 0; i  < raw_data.length; ++i) {
    for (var j = 1; j  <
      raw_data[i].length; ++j) {
      data.setValue(j-1, i+1, raw_data[i][j]);
    }}
  new google.visualization.BarChart(
    document.getElementById('visualization')).
      draw(data,
           {title:'Bar Chart', width:600, height:400,
            vAxis: {title: %{Label1.Name}%},
            hAxis: {title: %{Number.Name}%}});}
google.setOnLoadCallback(drawVisualization);
```

Listing 1: The generalised JavaScript template for a Google Bar Chart.

As previously introduced, the result set returned by a SPARQL query is represented, in our process, by an instance of the concept `ResultSet` and the variables are represented by the instances of the concept `Label` related to it. The `.Name` and `.Type` properties of the concept `Label` represent the name of the variable and the data type respectively. These values are then extracted from the Chart Ontology and replaced in the code template. On the other hand, the `.Value` string, is replaced recursively with the values contained in the result set of the initial SPARQL query. In case the chart is associated with a result set that has more than one `Label` element of same `.Type` we use a number in order to distinguish between them. The elements `%{Start_Repeat}%` and `%{End_Repeat}%` identifies part of code that need to be replicated every time a value is replaced.

For each type of chart that the visualization library can generate, an instance of each of the sub concepts of `VisualizationMethod` is created, with the required parameters set, code template. This is then stored in the ontology.

3.3 Inference Process

The ontologies discussed so far are defined using OWL-DL (a subset of the Web Ontology Language (OWL) (Dean et al., 2004) based on Description Logics) and SWRL (Semantic Web Rule Language) (Horrocks et al., 2004); these two W3C standard definitions, allow the definition of rules and logic constraints for ontologies.

In order to make use of the information stored in the Label and Chart Ontologies, we use an hybrid solution, mixing logical inference with DL-Safe (Motik et al., 2005) forward chaining rules. The logical inference process consists of translating the semantics of the RDF, RDFS (RDF Schema) (Brickley et al., 2004) and OWL-DL syntax into inferred triples that can fire rules that otherwise would remain inactive. For example, the `SubClassOf` predicate used to define sub concept relations is used to infer that each instance of the concept `TimeLine` is also an instance of a `VisualizationMethod`, in the same way an instance of the concept `Country` is also an instance of the concept `Location` and therefore also an instance of the concept `Label`.

Using OWL-DL, it is possible to define *Equivalent Concepts*, which are defined using appropriate logical restrictions. For example, we can define an equivalent concept `TemporalResultSet` as a sub concept of `ResultSet` by restricting the types of elements in `ResultSet` to labels representing temporal information:

$$TemporalResultSet \equiv \exists elements.Time \wedge \geq 2 elements.Label \quad (1)$$

Fully this means that a temporal result set has at least one element of type `Time` and at least an additional element that represents the value at that point in time.

The inference process will be used to re-classify the existing concept instances into the hierarchy

[3] http://developer.yahoo.com/yui/3/charts/

[4] http://www.simile-widgets.org/exhibit/

and appropriate equivalent concepts. Once inference is completed, the rule engine will apply the SWRL rules, to create instances of the relation `visualizeWith`, which defines which visualization method should be used to visualize a particular result set. An example of a rule that generates this kind of relation will be:

$$TemporalResultSet(?r) \wedge TimeLine(?t) \rightarrow visualizeWith(?r, ?t) \quad (2)$$

$$ResultSet(?r) \wedge PieChart(?v) \wedge elements(?r, ?e) \wedge Percentage(?e) \rightarrow visualizeWith(?r, ?v) \quad (3)$$

A third ontology, called the Instance Ontology, which imports the two ontologies described above, is then created. This contains the instances of the concepts defined in the two imported ontologies. The instances in the new ontology are defined specifically for the application environment. Every time a SPARQL query is executed its result set is associated with an identifier and stored as a new instance of the concept `ResultSet`.

Once these rules and equivalent concepts have been finalized an automatic inference process (e.g. supported by Pellet reasoner (Parsia and Sirin, 2004)) can be executed on the Instance Ontology. This will add additional statements regarding the instances of the ontology, then a rule engine (in our case we still use Pellet reasoner because it also supports SWRL rules) applies the rules in order to associate the `ResultSets` with the possible visualizations.

3.4 Chart Creation Process

Once the instances of the various `ResultSets` have been associated with one or more instances of `VisualizationMethod` and stored in the Instance Ontology, this ontology is processed by the chart creation process (Figure 1) which makes use of existing graphical libraries to generate the chart for the user interface.

The first step is to extract the visualization code (which has been introduced in Section 3.2); this can be extracted with a SPARQL query on the relation `visualizeWith`.

In the case where more than one visualization method is returned, the user will have the option to switch between all the possible charts, once the default chart is created and displayed. An alternative will be to create all the charts associated with a result set or to define a specific selection method to extract the most meaningful chart.

The second step is to define a process that will use the visualization code extracted during the previous step. How to use the code stored in the `visualizationCode` property, it depends on the visualization library that has been selected, this aspect relates to the specific implementation of the system and is therefore extremely flexible. In this approach we do not want to put any constraint on the technology to use: as an example Java, PHP, Ruby can all be used to process the code template and replace the keywords between %{}% and return the final JavaScript code that can be submitted to the Google Visualization API which will then create the selected chart.

4 USE CASE: SERVICE MONITOR VISUALIZATION

In this section, we discuss how we applied our method to a web service monitoring application in order to dynamically visualize monitoring data from different SLA perspectives.

The function of the monitoring application is to collect information about web services and store it into a database for further analysis. However, collecting information is not the ultimate purpose, the data needs to be processed and analysed in order to show whether the monitored services are compliant with pre-agreed SLAs. Previously, the analytical results were presented using Oracle BI™through dashboards. As discussed before, this solution is lacking in flexibility, especially when the end users want to see different aspects of an SLA, such as service availability, response time, and error rate, and those aspects cannot be represented using the same diagrams or charts.

We now apply the newly proposed method to this scenario to examine how the preexisting situation can be improved.

4.1 Expose Data to Visualize as RDF

The monitoring data is stored in a relational database. The system can monitor web services either directly through SOAP messages or through an Enterprise Service Bus (ESB), there is a separate table which stores monitoring data for each method. To generate the dashboard, we would need to manually construct a SQL query to collect the required data, and then save it into Oracle BI.

To apply our proposed method, we first need to convert the relational data model into a semantic data model in RDF (Hayes and McBride, 2004) and RDFS (Brickley et al., 2004). (The RDFS part of the model

is used to describe the relational data structure. The RDF part of the model is to describe the records in each table). This was done using a translator (Sahoo et al., 2009) such as D2R (Bizer, 2004) to provide then mapping between the database and RDF.

4.2 Create the Instance Ontology

Once the service monitoring information is accessible through a SPARQL endpoint it is possible to query the monitoring information using an appropriate SPARQL query.

The query is stored in the Instance Ontology by defining the set of variables in the `SELECT` declaration as an instance of the concept `ResultSet` in the Chart Ontology, with the respective concepts in the Label Ontology.

In our use case, there are three variables that compose the result set: `serviceEndPoint`, `messageType` and `countMessageType`. By analysing the result set for each one of the variables by checking the type of the value returned manually, we generated the instance of `ResultSet` from the Instance Ontology and all of the instances of related concepts.

The instances are therefore defined as:

$$\begin{aligned} &ResultSet(ResultSet1), Integer(Number1) \\ &Label(Label1), URI(Label2) \\ &name(Number1, countMessageType) \\ &name(Label1, messageType) \\ &name(Label2, serviceEndPoint) \\ &element(ResultSet1, Number1) \\ &element(ResultSet1, Label1), element(ResultSet1, Label2) \end{aligned} \quad (4)$$

Instances of the `VisualizationMethod` concept are also defined. We create an instance of the visualization method for each of the chart types (more precisely for each different configuration), for example:

$$\begin{aligned} &BarChart(GoogleBarChart1) \\ &visualizationCode(GoogleBarChart1, \texttt{'func...);'}) \\ &TimeLine(GoogleTimeLine1) \\ &visualizationCode(GoogleTimeLine1, \texttt{'func...);'}) \\ &LineChart(GoogleLineChart1) \\ &visualizationCode(GoogleLineChart1, \texttt{'func...);'}) \\ &ScatteredChart(GoogleScattered1) \\ &visualizationCode(GoogleScattered1, \texttt{'func...);'}) \end{aligned} \quad (5)$$

Each instance also defines the content of the additional properties specific of the selected library, an an example in the case of Google Chart API the property `visualizationCode` of the concept `VisualizationMethod`, will contain the template of the JavaScript code that is required to create the chart (as an example the templates in Table 1 for the instances `GoogleBarChart1`).

Once the instances of the visualization methods and results sets concepts have been created and stored in the Instance Ontology, it is possible to execute the inference process. As already described in the previous section this inference process relates instances of `ResultSet` to `VisualizationMethod` with the `visualizeWith` relation to produce the following:

$$\begin{aligned} &visualizeWith(ResultSet1, GoogleLineChart1) \\ &visualizeWith(ResultSet2, GoogleBarChart1) \end{aligned} \quad (6)$$

4.3 Generate the Final Charts

For this case we used Ruby as the enabling technology to process the various SPARQL result sets and generate the final charts.

The stack used in this case is illustrated in Figure 2: it contains a Ruby runtime, including *Sinatra*, *haml* and *sparql-client* Ruby *gems*[5]. For SPARQL endpoints we use OWLIM[6] for the three ontologies (Label, Chart and Instance) supporting our system and D2R[7] to create a virtual RDF-graph of the database.

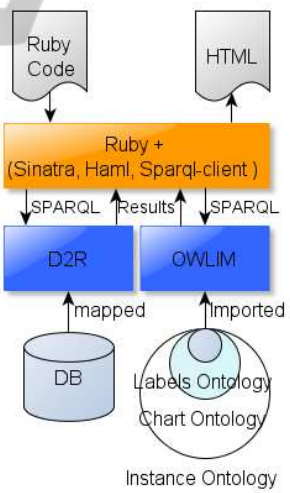

Figure 2: The set up of the system used in this scenario.

To generate the HMTL page containing the Chart, the first step is to extract the visualization code from the `visualizationCode` property. This code is then processed by the ruby stack described to create HTML output containing the JavaScript function to generate the chart.

[5]http://rubygems.org/gems
[6]http://www.ontotext.com/owlim/
[7]http://www.w3.org/2001/sw/wiki/D2RServer

The execution of all the code described so far will generate an HTML page showing the chart in Figure 3, which in our system replaces the chart in the dashboard.

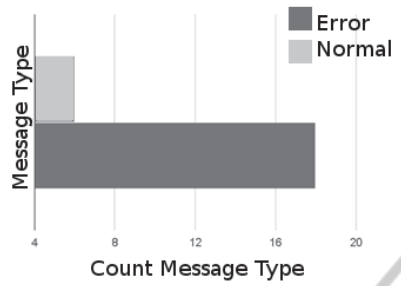

Figure 3: The Google Bar Chart generated by our system.

The use case described so far is a simple case but gave us good feedback on the quality of the proposed system.

In order to have higher degree of flexibility in the generation of the JavaScript code, and especially in enhancing the quality of the fully automated chart generation process we believe that the use of more modular languages such as Java is a better solution. We plan to implement a first prototype using Java, together with Pellet reasoner, Sesame[8] and Apache Tomcat as server-side technology to generate the required JavaScript.

5 CONCLUSIONS

In this paper we have presented a promising approach for the automatic generation of charts from a SPARQL query. The system exploits inference processes in order to generate an appropriate chart that can be used to visualize the result set returned by a specific SPARQL query. An application of this approach in the field of service monitoring has been presented.

The major benefit of our approach is the automatic on-the-fly generation of charts without the need for manual mapping between visualization and data storage layers (as would be required in current BI systems).

Future work will consider extending the scenario presented in this paper to the implementation of a generic framework, which is able to automatically create charts and dashboards in a generic context. Additional work will be to capture user interaction with the chart in order to automatically generate new SPARQL queries that will consequently lead to new views of the data, this future direction will be especially interesting in the case where the data to analyze can be linked to an external data such as Linked Data.

Our approach is particularly relevant in that case, as the queries that can be submitted can not be predicted at design time because of the high dimensionality and the highly connected nature of the data. This, in turn, may lead to the visualization of data that initially was considered irrelevant, which is a situation that current BI systems can not handle.

[8]http://www.openrdf.org

REFERENCES

Bizer, C. (2004). D2rq - treating non-rdf databases as virtual rdf graphs. In *In Proceedings of the 3rd International Semantic Web Conference (ISWC2004*.

Brickley, D., Guha, R., and McBride, B. (2004). Rdf vocabulary description language 1.0: Rdf schema. Recommendation, W3C.

Dean, M., Schreiber, G., Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004). Owl web ontology language reference. Recommendation, W3C.

Hayes, P. and McBride, B. (2004). Resource description framework (rdf). Recommendation, W3C.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., and Dean, M. (2004). Swrl: A semantic web rule language combining owl and ruleml. Member submission, W3C.

Katifori, A., Halatsis, C., Lepouras, G., Vassilakis, C., and Giannopoulou, E. (2007). Ontology visualization methods, a survey. *ACM Comput. Surv.*, 39.

Knublauch, H. (2010a). Sparql web pages (swp, aka uispin). Technical report, TopQuadrant.

Knublauch, H. (2010b). Uispin - charts. Technical report, TopQuadrant.

Leida, M., Afzal, A., and Majeed, B. (2010). Outlines for dynamic visualization of semantic web data. In *OTM 2010 Workshops*, volume 6428 of *Lecture Notes in Computer Science*, pages 170–179. Springer Berlin / Heidelberg.

Motik, B., Sattler, U., and Studer, R. (2005). Query answering for owl-dl with rules. *Web Semant.*, 3:41–60.

Parsia, B. and Sirin, E. (2004). Pellet: An owl dl reasoner. In *3rd International Semantic Web Conference (ISWC2004)*.

Sahoo, S. S., Halb, W., Hellmann, S., Idehen, K., Jr, T. T., Auer, S., Sequeda, J., and Ezzat, A. (2009). A survey of current approaches for mapping of relational databases to rdf.