SPARQL Query Writing with Recommendations Based on Datasets

Gergő Gombos and Attila Kiss

Eötvös Loránd University, Budapest, Hungary
{ggombos,kiss}@inf.elte.hu

Abstract. When we write a SPARQL query, we need to know the structure of the dataset. In the relation databases the tables have a scheme, but the semantic data do not have. Autocompletion function exists in SQL environment, but it does not exist in SPARQL environment. We made a system that can help to write SPARQL query. The system has two features. The first is the prefix recommend. We can write shorter queries if we use prefixes because we do not need to write the long IRIs. The second feature is the predicate-based recommendation based on the type of the variable. If a variable is in the query and it has a type condition, then our system recommends further predicates of this type. Our system needs information about the dataset for the recommendation. We can get these information with simple SPARQL queries. The queries run on a federated system. It is useful because the user does not need any information about the endpoints.

Keywords: SPARQL, Semantic Web, Linked Data, LOD Cloud, Federated system.

1 Introduction

The aim of the semantic web is to make a big knowledge base from the Internet. These knowledges can be combined and we can get more information about the things. We use the SPARQL language for querying this large semantic data. The query has some conditions that decrease the result. The result need to match these conditions. This solution is similar to the SQL in the relational database environment, where the data are stored in the relational tables and we give the conditions that have to be met. If we know the syntax of the SQL, it is not a problem writing these kinds of query. We need to know just the tables and the columns of the tables. Most database clients have autocompletion feature that makes easy the creation of the query. In the semantic web world it is not so simple. We need to know the dataset and its structure in order to write a query. The advantages of the autocompletion can be used for writing SPARQL queries. When we make a query we usually use the prefix form of the entities. The prefixes give us the opportunity to use the short string instead of the long IRIs. This makes easier writing of the query for us, but the query engines need the full IRIs of the things. Therefore the prefix recommendation is an important function of the semantic recommender system. Another important thing is a predicate recommendation because the semantic data is unstructured and we do not know what kind of predicate we can use. For example even, if

S. Yamamoto (Ed.): HIMI 2014, Part I, LNCS 8521, pp. 310–319, 2014.

we know the type of a variable, we do not know the other predicates. Therefore, the recommendation system queries the datasets. Because the datasets are in the LOD Cloud we need to choose the appropriate endpoint. For this we need to know the URL of the endpoint and we need to know what endpoint stores the specific data, which is usually not available for the user. Our aim is to create a general system that uses all endpoints and solves the endpoint selection problem. The federated systems select automatically the required endpoints and they summarize the results from the endpoints.

In this paper, we will describe the formal model of federated systems with Abstract State Machine (ASM). Then, we will refine the model to the current system. Then, we will present a prototype that is able to make recommendations based on a SPARQL query.

2 Related Work

The authors wrote [3] that the semantic data are difficult to access because the non-expert users cannot know the syntax of the SPARQL. They will produce the Linked Query Wizard. The hypothesis for the Linked Data Query Wizard is: the users know spreadsheet applications like Excel, and the idea is to make the semantic data into tabular form. Our solution provides the expert and non-expert users to make SPARQL queries easier.

The SPARQL is often given with visual tool. One of them is the SPARQL Views [5] that is an extension for Drupal. This extension helps the inexperienced users. The system queries the predicates from the given endpoint, and it recommends these to the users. The other function is the automatically prefix adding. When the user chooses a predicate, the system adds the necessary prefix to the query. Our solution uses a federated system and we can use the recommendation without choose any endpoint.

Another visual SPARQL editor is the NITELIGHT [4]. The NITELIGHT tool is a web-based application in JavaScript. The application provides an ontology browser, which allows us to add predicates to our query based on ontology. The queries are made by linking the components. The completed query is syntactically correct. In contrast to our system it has not recommendations. The user of the system needs some knowledge about SPARQL.

Kramer et. all [6] wrote querying Linked Open Data with SPARQL is different from querying relational databases. Their aim is to make autocompletion function for query writing. Their solution is to build indexes to the queries from logs. If the user writes a '<' symbol then the system recommends the potential IRIs. If the user writes a '?' symbol it recommends the variables. When the user chose a variable the system recommends the predicates based on the previous queries. In contrast to our system provides recommendations based on the dataset.

Lehmann et all.[7] presented a technique for making SPARQL. Their solution is based on the question-answer and the positive learning techniques. The user enters a query for which the system makes recommendations. The user selects a positive example from the recommendations. That is the base of the next recommendations. This iteration runs until the user reaches the appropriate query or there are no more learnable query. This solution uses SPARQL Endpoints like our solutions.

3 Semantic Web

We mentioned the Semantic Web in our related work [12]. The Semantic Web [1] aims at creating the web of data: a large distributed knowledge base, which contains the information of the World Wide Web in a format which is directly interpretable by computers. The goal of this web of linked data is to allow better, more sensible methods for information search, and knowledge inference. To achieve this, the Semantic Web provides a data model and its query language. The data model called the Resource Description Framework (RDF) [13] uses a simple conceptual description of the information: we represent our knowledge as statements in the form of subject-predicate-object (or entity-attribute-value). This way our data can be seen as a directed graph, where a statement is an edge labeled with the predicate, pointing from the subjects node to the objects node. The query language called SPARQL [2] formulates the queries as graph patterns, thus the query results can be calculated by matching the pattern against the data graph.

4 Formal Model

We made a formal model with ASM (Abstract State Machine). ASMs represent a mathematically well founded framework for system design and analysis [10]. It is introduced by Gurevich [9]. The federated model is inspired by the ASM model of the grid systems [8]. The grid systems are distributed and parallel like the federated systems. The ASM algebra is made up of universes, functions, and rules. The universes include the entities. The functions provide the link between the universes. The rules are transaction steps and they have condition to activate. The ASM has a ground model that is a base of the system functions. This model will be refined later. The model describes the expected requirements of the system. We describe below the workflows of the federated systems with ASM and we refine for the SPARQL recommendation system.

4.1 Model for Federated System

A semantic Federated system (FEDERATED universe) operates as follows. The system receives a query (QUERY universe) which sends to several SPARQL Endpoints (ENDPOINT universe) and it summarizes the results (RESULT universe) of the endpoints and it returns with the answer. The ground model does not deal with endpoint selection method. It helps to be the model general. The endpoint selection function can be given in a refined model, but these are not discussed in this paper. The universes have the $true, false, undef$ values too.

The relations between the universes can be described by functions. We describe the state of the federated system with $fstate : FEDERATED \rightarrow \{wait, start\_req, running\}$ function. This state is *wait* if the system is waiting for a request. It is *start_req* when the system prepares the requests to the endpoints. Finally, the state is *running* if the system works on a query. When a request comes into the system we can write the connection with the $fworkingOn : FEDERATED \rightarrow QUERY$ which says that the federated system works on the query. The system converts the query to requests. The exact request

is not important in the ground model. It can be refined in the lower-level model because it depends on the architecture of the federated system. The query and the requests connect with $reqQuery : REQUEST \rightarrow QUERY$ function. A request will be executed in a given endpoint. The connection between the request and the endpoint is written with $eworkingOn : ENDPOINT \rightarrow REQUEST$ function. The start of the request depends on the state of the endpoint. Initially, their state are $waiting$. These endpoints are waiting for the requests. The $estate : ENDPOINT \rightarrow \{running, waiting, finished\}$ function describes the state of the given endpoint. The state of the endpoint changes when an event occurs. We describe an event with $event : ENDPOINT \rightarrow \{timeout, finish\}$. The $timeout$ occurs when the endpoint cannot answer the request and the time is out. It is necessary because the system needs minimal response time for usability. The $finish$ state occurs when the result is complete on time. We get the results with $rres : REQUEST \rightarrow RESULT$ function. The federated system summarizes these results. The method of the summarization is not discussed in the ground model. Finally, the final result stores with $qres : QUERY \rightarrow RESULT$.

The model needs an initial step. Each item of the model need to be reset. First, we set the state of the endpoints: $\forall e \in ENDPOINT : estate(e) := waiting; eworkingOn(e) := undef$ and we set the state of the system: $fstate(f) := wait$.

The system operations are described with rules.

Rule 1 (Send a Query to the Federated System). The first rule describes that the system receives a query ($q \in QUERY$). The query can run only if the system is in $waiting$ state. In this time the state of the system is changed to $start\_req$ that means it prepares the requests and we set the query result to empty, and we set the system work on this query.

```
if fstate(f) = wait then
    fworkingOn(f) := q
    fstate(f) := start_req
    qres(q) := undef
endif
```

Rule 2 (Federated System Send the Request to the Endpoints). The evaluation of the query needs requests. The following rule creates a request to each endpoint that has $waiting$ state. In this case, we do not deal whit what the request is. In some system this may be the whole query, in another system just the conditions of the query. When the system makes a request it is set the query and the endpoint for the request. It changes the state of the endpoint to $running$ and the result of the request to empty.

```
if fstate(f) = start_req && fworkingOn(f) = q then
  do forall e ∈ ENDPOINT
      if estate(e) = waiting then
          EXTEND REQUEST by req with
              reqQuery(req) := q
              eworkingOn(e) := req
              estate(e) := running
              rres(req) := undef
```

```
                endextend
            endif
            fstate(f) = running
        enddo
endif
```

The EXTEND means that we create a new item into the universe, in this case in the REQUEST.

Rule 3 (Endpoint Finish or Timeout). A request may end in two states. One is if the query was run without any problems. The second state is if the request could not finish within a certain time. In both cases an event occurs. We take the result of the request to the result of the query with '+' operator. We do not deal what is mean the '+' operator and how is it work.

```
let  req = eworkingOn(e)
if  event(e) = finish || event(e) = timeout  then
    eworkingOn(e) = undef
    estate(e) = finished
    qres(q) := qres(q) + rres(req)
    rres(req) := undef
    REQUEST(req) = undef
endif
```

The $REQUEST(req) = undef$ means that the item (req) is removed from the universe ($REQUEST$).

Rule 4 (terminate) The last process is the termination process. This process is run when the state of each endpoint changed to $finished$. We get the result in $qres(q)$. Afterthat we need to restore the system state to the initial state for the further requests.

```
if  ∀e ∈ ENDPOINT : estate(e) = finished && fstate(f) = running  then
    do  forall  e ∈ ENDPOINT
            estate(e) := waiting
            fstate(f) := wait
            fworkingOn(f) := undef
    enddo
endif
```

4.2 Finite Model for SPARQL Recommendation

We showed in the previous model how the system gets a query and how a federated system will process this query. Now we refine this model for the current task. The aim is to make query, so the input of the system is not a QUERY, just a part of the query. For this reason, we need to introduce new universes. In this task we focus on two parts of the process. One is a prefix recommendation. For recommendation we need the SHORTPREFIX and the LONGPREFIX universes. These universes will store the short

and long form of the prefixes. Another aim is the condition recommendation. For this we need introduce the CONDITION universe.

We make new expectations on the new refined model. The system is able to define the prefixes without sending a request to the endpoints. This may be because the prefixes usually are fixed, so we can use these as a constant. The REQUESTs contain CONDITION instead of QUERY. The REQUEST depends on the CONDITION and it is made if the CONDITION has a type information.

The new universes need new functions. The first is a $prefMapped$: $SHORTPREFIX \rightarrow LONGPREFIX$ which performs the mapping of the prefixes. We need to resolve the short prefix during the query writing, so this mapping is just one direction. Because the query is now divided into several parts, we need the $pbelongsTo : SHORTPREFIX \rightarrow QUERY$ and the $cbelongsTo : CONDITION \rightarrow QUERY$ functions for the connection of the three universes. We need to check that the CONDITION has a type ($rdf : type$) information, this check is made by the $hasType : CONDITION \rightarrow \{true, false\}$ function. Another checking functions are the $hasCondition : QUERY \rightarrow \{true, false\}$ and the $hasPrefix : QUERY \rightarrow true, false$. These functions check that the QUERY has PREFIX or CONDITION. In the ground model we used the $reqQuery$ function, but now we need to change this on the current model. The input of this function was QUERY, but now this will be CONDITION.

We extend the initial step with a loads process that load the short and long version of prefixes to the $prefMapped$. The exact implementation of the loading is not included the model. Another supplement is that we set the value of the $cbelongsTo, pbelongsTo$ functions based on the (sub)query.

Rule 1 (Refined). On the first rule we need just a minimal change. The ground model sent the query to the system every time, but now it sends just if the query has PREFIX or CONDITION.

if $fstate(f) = wait$ && $(hasPrefix(q)$ || $hasCondition(q))$ **then**
  $fworkingOn(f) := q$
  $fstate(f) := start\_req$
  $qres(q) := undef$
endif

Rule 2 (Refined). The second rule sends the queries to the endpoints. If the query has PREFIX it does not need to send the query because we can answer the prefix recommendation without it. If the query has CONDITION, the system works like a ground model.

if $fstate(f) = start_req$ && $fworkingOn(f) = q$ **then**
  if $hasCondition(q)$ **then**
    do forall $c \in CONDITION$
      if $cbelongsTo(c) = q$ && $hasType(c)$
        do forall $e \in ENDPOINT$
          if $estate(e) = waiting$
            EXTEND $REQUEST$ **by** req **with**
              $reqQuery(req) := c$

```
                                    eworkingOn(e) := req
                                    estate(e) := running
                                    rres(req) := undef
                                endextend
                            endif
                        enddo
                    endif
                enddo
                fstate(f) = running
            endif
            if hasPrefix(q) then
                do forall p ∈ PREFIX
                    if pbelongsTo(p) = q
                        qres(q) := qres(q) + prefMapped(p)
                    endif
                enddo
            endif
        endif
```

5 Implemented System

We built a prototype based on the previous model. The features of the prototype are the prefix recommendation and condition recommendation that were described above. On Fig. 2 we can see the Web UI of the system. It has a query box, where the user writes the query and the system send an AJAX request to the backend, where the model is working. The system uses predefined SPARQL endpoints: factbook[1], dataGov[2], dblp[3], dbpedia[4], factforge[5], openlinkSW[6], linkedMDB[7], void[8]. In addition, the system stores the short and long forms of the prefixes. The prefixes are from the prefix.cc. The federated system [11] usage is advantageous because the user does not need to know, what endpoint store the data or what is the URL of the endpoint.

The system works as follows. If there is any change in the query, then that will be sent to the backend asynchronously. We use ARQ[9] to process the SPARQL query. We get the condition from the WHERE with this tool. If the query is wrong, then the ARQ write the problem and we show them on the UI, see that on Fig. 1. If the system finds a prefix that is not defined previously, it searches them from prefixes and make a recommendation. If we want to use this recommendation, we need just click on the 'add' button. It is possible that we wrote a prefix, that the system does not recognize - this may be if we

[1] http://wifo5-04.informatik.uni-mannheim.de/factbook/sparql
[2] http://services.data.gov/sparql
[3] http://dblp.rkbexplorer.com/sparql
[4] http://dbpedia.org/sparql
[5] http://factforge.net/sparql
[6] http://lod.openlinksw.com/sparql
[7] http://data.linkedmdb.org/sparql
[8] http://void.rkbexplorer.com/sparql
[9] http://jena.sourceforge.net/ARQ/

Fig. 1. Web UI of the system with error

use another short form of an IRI - then we get an error message. On Fig. 2 we can see the *dbpedia* : *Person* IRI that has the *dbpedia* as prefix. The system knows this prefix and recommends this line: 'PREFIX dbpedia: <http://dbpedia.org/resource>'.

Another function of the system is that the condition recommendation. The system makes recommendations to extend the query with new filters. The basis for that is the variable with type (*rdf* : *type* or short form *a*) information. The system collects additional information about a type with simple SPARQL queries. It sends the following query to all endpoints.

```
SELECT DISTINCT ?x WHERE {
    ?x rdf:type dbpedia:Person .
} LIMIT 3
```

We ask three items because one item may not have some predicates and another item has. Ask three items is fast enough that the system is able to respond in time. When the system gets three items that has a same type, the system asks the possible predicates of the items. We write another SPARQL query for this and the system makes the unique result.

```
SELECT DISTINCT ?s WHERE {
    item ?s ?p .
}
```

The first query returns the items that are Persons and after the second query the system makes the unique predicates.

The system completes this process on all endpoints. For fastest response time these queries run in parallel. Since some endpoint may not be available or overloaded, the answer would be a long time, so the system has a timelimit which will drop the request if it does not receive result before the limit. The limit is on our system is 5 seconds. The system is faster with limit, but we cannot get all results. In many cases this is not a problem because a lot of data are stored more endpoints.

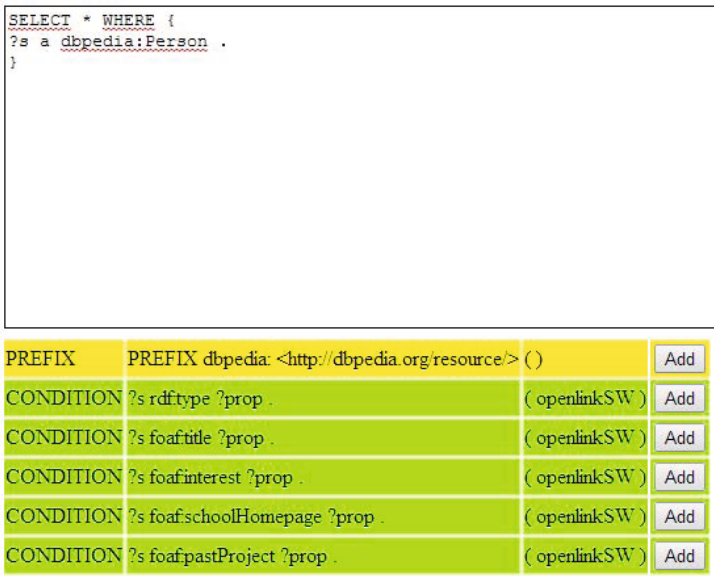

Fig. 2. Web UI of the system with recommendation

6 Conclusion and Future Work

One of the difficulties of writing a SPARQL query that we do not know the scheme of the dataset. Without the scheme we do not know what we can query about an item. Another problem is the long IRIs in the query. The SPARQL provides a solution to use prefixes, but it is often required to search them. The system, which is described in this paper, gives a solution to these problems. The system makes recommendations when we are writing the SPARQL query. It offers the necessary prefixes and the possible properties of a variable.

The system is currently used only for the preparation of SPARQL query. The final query can be used on another system. Our plan is that the query can be automatically sent to the appropriate endpoint. In addition, we would like to make the extraction of the prefixes automatically as mentioned above. Create a query usually starts with an initial item. In this system the search IRI function is not available, in turn, for example the Virtuoso has the Facet for this function. We plan to write this function to the model and implement to the system. We could make the system when we use some cache. This cache can store information about the previous requests. If some endpoint did not send any result about some type, then the system does not need to ask again.

Acknowledgments. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013). We are grateful to Bálint Molnár for helpful discussion and comments about ASM.

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American 284(5), 28–37 (2001)
2. Prud Hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation 15 (2008)
3. Hoefler, P.: Linked Data Interfaces for Non-expert Users. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 702–706. Springer, Heidelberg (2013)
4. Russell, A., Smart, P.R., Braines, D., Shadbolt, N.R.: NITELIGHT: A Graphical Tool for Semantic Query Construction (2008)
5. Clark, L.: SPARQL Views: A Visual SPARQL Query Builder for Drupal. ISWC Posters & Demos (2010)
6. Kramer, K., Dividino, R., Grner, G.: SPACE: SPARQL Index for Efficient Autocompletion. In: ISWC Posters & Demonstrations Track, pp. 157–160 (2013)
7. Lehmann, J., Bühmann, L.: AutoSPARQL: Let users query your knowledge base. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 63–79. Springer, Heidelberg (2011)
8. Nmeth, Z., Sunderam, V.: A formal framework for defining grid systems. In: 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid. IEEE (2002)
9. Gurevich, Y.: Evolving algebras: An attempt to discover semantics. In: Current Trends in Theoretical Computer Science, pp. 266–292 (1993)
10. Börger, E.: High level system design and analysis using abstract state machines. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 1–43. Springer, Heidelberg (1999)
11. Rakhmawati, N.A., Umbrich, J., Karnstedt, M., Hasnain, A., Hausenblas, M.: Querying over Federated SPARQL Endpoints-A State of the Art Survey. arXiv preprint arXiv:1306.1723 (2013)
12. Matuszka, T., Gombos, G., Kiss, A.: A New Approach for Indoor Navigation Using Semantic Webtechnologies and Augmented Reality. In: Shumaker, R. (ed.) VAMR/HCII 2013, Part I. LNCS, vol. 8021, pp. 202–210. Springer, Heidelberg (2013)
13. Lassila, O., Swick, R.R.: Resource Description Framework (RDF) Schema Specification, `http://www.w3.org/TR/rdf-schema`