Incorporating API data into SPARQL query answers

Matías Jünemann, Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč

PUC Chile and Center for Semantic Web Research

Abstract. In this demo we present an extension of SPARQL which allows queries to connect to JSON APIs and integrate the obtained information into query answers. We achieve this by adding a new operator to SPARQL, and implement this extension on top of the Jena framework in order to illustrate how it functions with real world APIs.

1 Introduction and motivation

Semantic Web provides a platform for publishing data on the Web via the Resource Description Framework (RDF). Having a common format for data dissemination allows for applications of increasing complexity since it enables them to access data obtained from different sources. However, the majority of data available on the Web today is still not published in RDF, and is thus not (directly) available to Semantic Web services. Huge amount of this data is accessed through Web APIs which use a variety of different formats to provide data to the users. Therefore it would be useful to allow SPARQL, the standard query language for the Semantic Web, to access and use all of this data.

In this demo we make a first step in this direction by extending SPARQL with the capability of communicating with JSON APIs. We picked JSON because it is currently the most popular data format in Web APIs, however, the results presented here can easily be extended to any API format; we stick with JSON simply to keep the presentation manageable. By allowing SPARQL to connect to an API we can utilise not just the information that is available locally in our dataset, but also extend the query answer with data obtained from a Web service. Use cases for such an extensions are numerous and can be particularly practical when the data obtained from the API changes very often (such as current weather conditions, sensor data, etc.). We illustrate how this extension works and why one might want to use it by means of an example.

Example 1. Suppose that you are travelling around Japan in order to do some skiing. You find yourself at the Hokkaido island and wish to find all the ski resorts close to your location. It is easy to obtain this information by querying e.g. the YAGO database using SPARQL, however, you will probably want to go skiing in a resort where the weather conditions are favourable (i.e. it is not raining nor snowing). Although you can not obtain weather information directly using SPARQL and join it with the list of resorts obtained previously, this information is available through a weather service API called `weather.api`. This API

implements HTTP requests, so for example to retrieve the weather in Sapporo you use the URI template:

`http://weather.api/request?q=Sapporo`,

to which the API responds with a JSON document containing weather information, say of the form

```
{"timestamp": "14/04/2016 11:59:07",
 "temperature": 11, "description": "Sunny"},
```

Thus, all you need is to produce one call to the weather API for each ski centre in Hokkaido, and filter out all those where the description is not *Sunny*. One can do this manually by e.g. querying with SPARQL first, and then executing an API call for each obtained resort. However this might be cumbersome, particularly when the number of answers is large, and does not allow us to incorporate this information into our query answers. On the other hand, we propose to extend SPARQL with the BIND_API operator, which allows us to easily obtain *all* of the desired information using the following query.

```
SELECT ?x ?n WHERE {
   ?x yago:isLocatedIn yago:Hokkaido .
   ?x rdf:type yago:wikicat_Ski_areas_and_resorts_in_Japan .
   ?x rdfs:label ?n
   BIND_API <http://weather.api/request?q={?n}>
                                        (["description"]) AS (?t)
   FILTER regex(?t,"Sunny")
}
```

The first part of our query is executed over the YAGO database and obtains the IRI representing the resort and the label of its location. We pass the label of the location as a parameter to the URI template used to consult the API. The newly introduced operator `BIND_API` takes the (instantiated) URI template and upon executing the API call processes the received JSON document using an expression `["description"]`, which obtains the value of the key `description` of the received JSON, and binds it to the variable $?t$. Generally, the answer we receive is going to be a collection of key-value pairs, so we need to specify which value we want to obtain and store using the `BIND_API` operator.

In this demo we showcase an extension of Jena framework implementing the `BIND_API` operator, and do live testing (on a remote server available at http://107.170.168.31/query/#/) using either preselected queries, or the ones provided by the visitors at the time of the demonstration. To keep the presentation manageable we will use YAGO as our base dataset, but allow arbitrary APIs to be used.

Related work. There have been several proposals to allow SPARQL to communicate with APIs (see e.g. [2]), the main difference here being that we offer communication capability with an arbitrary API as an integral part of the SPARQL query processor. On the other hand, there have been many attempts to transform data residing in other formats to RDF, the most popular paradigm here being that of mappings [1]. The main difference from the work we present is that mappings generally do not support querying "on-the-fly", which can be an issue when the data changes a lot such as e.g. with weather information.

2 The proposed extension

In order to allow SPARQL to obtain API data, we propose to extend the language with a new operator called BIND_API. Formally, we allow SPARQL to contain patterns of the form

$$P_1 \ \text{BIND\_API} \ U \ (N_1, N_2, \ldots, N_m) \ \text{AS} \ (?x_1, ?x_2, \ldots, ?x_m), \tag{1}$$

which we call BIND-from-API patterns. Here P_1 is an arbitrary SPARQL graph pattern [4], U a URI template [3] used to contact the API, and $N_1, \ldots, N_m$ a sequence of JSON navigation instructions [5] which tell us how to extract the desired value from the retrieved JSON document. By our definition BIND-from-API patterns can appear anywhere usual SPARQL patterns can. This can be e.g. inside a `WHERE` clause, such as in Example 1, or even as P_1 in (1), thus allowing us to obtain results from multiple APIs inside a single query.

Evaluating this operator over an RDF dataset G is done as follows. For each mapping μ in $[\![P_1]\!]_G$ we instantiate every variable $?y$ in the URI template U with the value $\mu(?y)$, thus obtaining an IRI which is a valid API call. We call the API with this instantiated IRI, obtaining a JSON document, say J. We then apply the navigation instruction N_1 to J and store the obtained value into $?x_1$. If the API call produced an error, or if the returned value is not a literal, we do not assign a value to $?x1$. Similarly, the value of N_2 applied to J is stored into $?x_2$, etc. After this is done, the mapping μ is extended with the new variables $?x_1, \ldots, ?x_m$, which have been assigned values according to J and N_is.

The implementation is done on top of the Jena framework, and does not alter the inner workings of the standard BIND operator. Full definitions of the syntax and the semantics are available at http://dvrgoc.ing.puc.cl/APIs/. We have also made the implementation available on github for the readers who would like to further test the capabilities of our extension: https://github.com/CSWR/SPARQL-JSONAPI.

3 Demonstration overview

The main focus of this demonstration will be the live query interface available at http://107.170.168.31/query/#/, which will allow the demo visitors to test out arbitrary queries which use API calls. The service we provide runs the extended version of the Jena TDB framework through Fuseki, and the query interface connects to this implementation using a Python script. The interface will check every five seconds if the results are available. In order to make the presentation more streamlined, we have decided to use a reasonable sized (2 GB) chunk of the YAGO and DBpedia database containing information about geographical locations as our base dataset. Apart from the usual query window available in SPARQL endpoints, we also have a separate window where the users can enter their strategies for accessing the APIs. As a default, we provide support for OAuth and the typical strategy of providing a personalised token to access the API. The visual presentation of the query interface is illustrated in Figure 3.

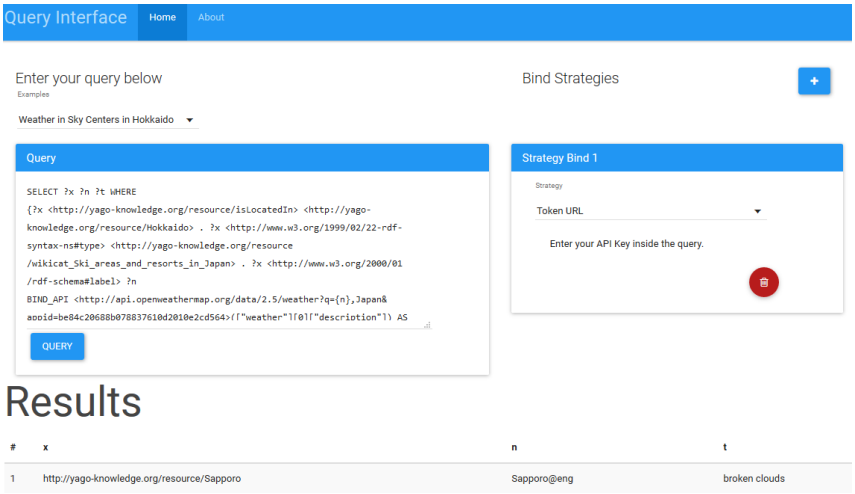

Fig. 1. Live query interface available at http://107.170.168.31/query/#/. The user types the query in the window to the left, and provides a strategy in the window to the right.

The demonstration will consist of two parts:

- First, we will give a brief introduction by example showcasing the different functionalities supported by our implementation.
- Second, we will allow the users to specify queries using API calls which they wish to execute (here we allow arbitrary APIs).

The aim of the demo is to emphasise the potential uses of such an extension to SPARQL through a series of examples, and also to show that our implementation can handle multiple (and simultaneous) user provided queries in real time on a remote server, thus simulating a typical SPARQL endpoint.

Acknowledgements. Work funded by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004.

References

1. A. Dimou, M. V. Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. V. de Walle. RML: A generic language for integrated RDF mappings of heterogeneous data. In *LDOW*, 2014.
2. P. Fafalios and Y. Tzitzikas. SPARQL-LD: a SPARQL extension for fetching and querying linked data. In *ISWC Demos*, 2015.
3. IETF. URI Template. https://tools.ietf.org/html/rfc6570, 2012.
4. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3), 2009.
5. F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *WWW 2016*, pages 263–273, 2016.