Incorporating API data into SPARQL query answers

Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč

PUC Chile and Center for Semantic Web Research

Abstract. Although the amount of RDF data has been steadily increasing over the years, the majority of information on the Web is still residing in other formats, and is often not accessible to Semantic Web services. A lot of this data is available through APIs serving JSON. In this work we propose a way of extending SPARQL with the option to connect to JSON APIs and integrate the obtained information into SPARQL query answers, thus allowing to bring data from the "traditional" Web to the Semantic Web.

1 Introduction

Semantic Web provides a platform for publishing data on the Web via the Resource Description Framework (RDF) [13]. Having a common format for data dissemination allows for applications of increasing complexity since it enables them to access data obtained from different sources, or describing different entities. Over the past years billions of facts have been published using RDF, containing an increasing amount of data coming from corporate institutions (e.g. BBC, The New York Times), bioinformatics (e.g. Bio2RDF), or government bodies (e.g. data.gov). One of the biggest contributors to this explosion of semantic data on the Web has been the Linked Open Data community, whose efforts resulted in a huge amount of interconnected datasets being made freely available through public repositories such as DBpedia, YAGO, WikiData, and many other sources. In fact, it has now been long established that Linked Data is the correct way of publishing data on the Semantic Web [15].

However, the majority of data available on the Web today is still not accessible to Semantic Web services, be it because it is not published in the RDF format, or because there is currently no way of converting it to RDF. Huge amount of this data is made available through Web APIs which use a variety of different formats to provide data to the users. It would therefore be opportune to make all of this data available to Semantic Web technologies, thus allowing them to consume this wealth of information and create a truly connected Web. The need for this is further illustrated by the W3C's vision for data on the Web [19], which states that "To achieve and create Linked Data, technologies should be available for a common format (RDF), to make either conversion or on-the-fly access to existing databases (relational, XML, HTML, etc)."

One way of achieving this is to allow SPARQL, the standard language for accessing data on the Semantic Web [11], to consult Web APIs and incorporate their results into its output. In this paper 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 in the paper 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, 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, state of the traffic, opening times for venues, etc.). To illustrate this let us consider the following 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. This is easily achievable by evaluating the following SPARQL query over YAGO (in the paper we use the standard set of YAGO and DBPedia prefixes available at `http://dbpedia.org/sparql?nsdecl`).

```
SELECT ?x WHERE {
   ?x yago:isLocatedIn yago:Hokkaido .
   ?x rdf:type yago:wikicat_Ski_areas_and_resorts_in_Japan }
```

Although this information gives you a good starting point, you will probably want to go skiing in a resort where the weather conditions are favourable (i.e. it is not raining nor snowing). Instead of looking up weather conditions one by one based on the listing obtained by the previous query, you would like to automatically integrate these answers with the information of a weather service provided through an API called `weather.api`, and filter out those locations where the weather is not good. The API implements HTTP requests, so for example to retrieve the weather in Sapporo you use the URL:

`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*. To achieve this we extend SPARQL with the BIND_API operator, which allows us to obtain the desired information via 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 database and obtains the IRI representing the resort and the label of its location (we need the label because APIs do not accept IRIs). We pass the label of the location as a parameter to the URL used to consult the API. The newly introduced operator `BIND_API` takes this URL 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 paper we discuss how such a binding operator should be implemented by extending the functionality of the standard `BIND` operator available in SPARQL. As a proof of concept, we extend the Jena framework [12] with an extra module for handling API calls, thus showing how this extension of the language can be achieved without much overhead on top of currently available SPARQL engines (the source code is available at [2]). To demonstrate that our implementation is indeed feasible we test it against several real-world examples and show that we can usually obtain the query answer in reasonable time. As it turns out, the bottleneck for evaluation is almost always the number of API calls, so we provide several optimisations that reduce this number. Based on experimental data we also provide recommendations on when it makes sense to apply these optimisations.

Related work. Although there has been a lot of work on enabling SPARQL as an API [3,5,9] (most importantly through the idea of endpoints), not much has been done on bringing the information made available by various Web APIs to SPARQL. Some notable exceptions to this have been [14,16,18], where the idea of building a wrapper for JSON documents and APIs has been introduced, thus transforming JSON data into RDF in order to make it available for SPARQL to consume. What we propose, on the other hand, is a way for SPARQL to obtain API data in a way that will not depend on a particular codification into RDF, but can be controlled freely by the user. The line of work most similar to ours is [6], where both an RDF wrapper for existing APIs, and an extension of SPARQL that allows calls to REST services is described. The main difference of this work and the one we present is in the fact that [6] uses many assumptions on the return format of API calls, which would make it uncompatible with modern JSON APIs, and the fact that the implementation we propose fits by its very definition into Jena pipeline as an extension of the standard `BIND` clause.

2 Notation

RDF Graphs. Let $\mathbf{I}$, $\mathbf{L}$, and $\mathbf{B}$ be infinite disjoint sets of *IRIs*, *literals*, and *blank nodes*, respectively. The set of *RDF terms* $\mathbf{T}$ is $\mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$. An *RDF triple* is a triple (s, p, o) from $\mathbf{T} \times \mathbf{I} \times \mathbf{T}$, where s is called *subject*, p *predicate*, and o *object*. An *(RDF) graph* is a finite set of RDF triples. For simplicity we assume that

RDF databases consist of a single RDF graph. In this way we avoid dealing with extra cases generated by the use of the GRAPH operator, although our proposal can easily be extended to deal with datasets with more graphs.

SPARQL Syntax and Semantics. We assume the reader is familiar with the syntax and semantics of SPARQL 1.1 query language. We recall here the general structure of SPARQL patterns, its semantics, and provide the details for a few operators that are heavily used in the paper.

We distinguish three types of syntactic building blocks— expressions, patterns, and queries, built over terms $\mathbf{T}$ and an infinite set $\mathbf{V} = \{?x, ?y, \ldots\}$ of *variables*, disjoint from $\mathbf{T}$. The official documentation for SPARQL 1.1 defines the syntax for expressions, as well as the syntax for *graph patterns* (or just *patterns*). As for queries, we will only focus on what is known as SELECT queries, but our results are independent of which query form is used (see the standard [11] for more details). We assume readers are familiar with the syntax of expressions, patterns and queries, as well as their semantics. In what follows we just recall the workings of the BIND and VALUES operators, and fix some notation regarding the semantics of expressions, patterns and queries.

If P is a graph pattern, R is an expression and $?x$ is a variable that does not appear in P, then $P\ \mathsf{BIND}\ (R\ \mathsf{AS}\ ?x)$ is a pattern, called a *bind pattern.* Moreover, if $?x_1\, ?x_2\, \cdots\, ?x_n$ is a list of variables and $l_1, \ldots, l_m$ are lists of elements from $\mathbf{T} \cup \{\text{UNDEF}\}$, each of size n, then $\mathsf{VALUES}\ (?x_1\, ?x_2\, \cdots\, ?x_n)\ \{(l_1)\, (l_2)\, \cdots\, (l_m)\}$ is also a pattern, called a *values pattern.*

The semantics of graph patterns is defined in terms of *mappings* [11]; that is, partial functions from the set of variables $\mathbf{V}$ to IRIs $\mathbf{I}$. The *domain* $\mathsf{dom}(\mu)$ of a mapping μ is the set of variables on which μ is defined. Two mappings μ_1 and μ_2 are *compatible* (written as $\mu_1 \sim \mu_2$) if $\mu_1(?x) = \mu_2(?x)$ for all variables $?x$ in $\mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2)$. If $\mu_1 \sim \mu_2$, then we write $\mu_1 \cup \mu_2$ for the mapping obtained by extending μ_1 according to μ_2 on all the variables in $\mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1)$. If $?x$ is a variable and $t \in \mathbf{T}$, we use $?x \mapsto t$ to denote the mapping that assigns t to $?x$ and does not assign values to any other variable.

Given a mapping μ and an expression R, we denote by $[\![R]\!]_\mu$ the result of evaluating the expression with respect to μ. This can be either a value from $\mathbf{T} \cup \{\texttt{error}\}$, if R is some sort of algebraic expression, or a value in $\{\texttt{true}, \texttt{false}, \texttt{error}\}$, if R is a boolean expression. Moreover, given a graph G and a pattern P, we denote the *evaluation* of a graph pattern P over G as $[\![P]\!]_G$. Note that the semantics of patterns includes SELECT queries, since these queries can now be understood as patterns, following the addition of nested queries in the latest version of the standard. We can now define the semantics of BIND, for P a graph pattern, R an expression and $?x$ a variable not in P:

$$[\![P\ \mathsf{BIND}\ R\ \mathsf{AS}\ ?x]\!]_G = \big\{\mu' \mid \mu \in [\![P]\!]_G, \mu' = \mu \cup \{?x \mapsto [\![R]\!]_\mu\}, [\![R]\!]_\mu \neq \texttt{error}\big\} \cup \big\{\mu \mid \mu \in [\![P]\!]_G, [\![R]\!]_\mu = \texttt{error}\big\}$$

Likewise, the following is the semantics of VALUES, for a list $?x_1\, ?x_2\, \cdots\, ?x_n$ of variables and lists $l_1, \ldots, l_m$ of elements from $\mathbf{T} \cup \{\text{UNDEF}\}$, each of size n.

Here $(?x_1\, ?x_2 \cdots ?x_n) \mapsto l_j$ refers to a mapping that assigns the i-th element of l_j to the variable $?x_i$, for each $1 \leq i \leq n$.

$$[\![\mathsf{VALUES}\ (?x_1 \cdots ?x_n)\ \{(l_1) \cdots (l_m)\}]\!]_G = \bigcup_{1\leq j\leq m} \{(?x_1 \cdots ?x_n) \mapsto l_j\}$$

JSON documents and navigation instructions. The JSON format [7] defines the following types of values. First, `true`, `false` and `null` are JSON values. Any decimal number (e.g. 3.14, 23) is also a JSON value, called a *number*. Furthermore, if `s` is a string of unicode characters then `"s"` is a JSON value, called a *string value*. Next, if $v_1, \ldots, v_n$ are JSON values and $s_1, \ldots, s_n$ are pairwise distinct string values, then $o = \{s_1 : v_1, \ldots, s_n : v_n\}$ is a JSON value, called an *object*. In this case, each $s_i : v_i$ is called a key-value pair of o. Finally, if $v_1, \ldots, v_n$ are JSON values then $a = [v_1, \ldots, v_n]$ is a JSON value called an *array*. In this case $v_1, \ldots, v_n$ are called the *elements of* a. Numeric values, strings and the boolean values `true`, and `false` are called *basic JSON values*.

We sometimes use the term JSON document (or just document) to refer to JSON values. The following syntax is normally used to navigate through JSON documents. If J is an object, then $J[\text{“key”}]$ is the value of J whose key is the string “key”. Likewise, if J is an array, then $J[n]$, for a natural number n, contains the (n-1)-th element of J. Both [“key”] and $[n]$, for some string key, and a natural number n, are called *JSON navigation instructions*. Since a value of a key can again be a JSON document, we allow stacking navigation instructions, thus resulting in expressions like e.g. [”key1”][7][”key2”], which assumes that the value of key1 is an array of JSON documents, and that the 7th element of this array is a JSON document which contains a key named key2. Such a concatenation of navigation expressions is again a JSON navigation expression.

3 Enabling SPARLQ to make JSON calls

In this section we define the syntax and the semantics of API calls made available in our framework. We begin by describing how JSON APIs function, and then describe a SPARQL operator which can incorporate the data obtained from APIs into SPARQL query answers. We illustrate the utility of this new operator using a set of real world examples.

3.1 JSON APIs, requests and navigating JSON documents

While theoretically one can use our ideas to connect SPARQL to any Web API, we concentrate on the so-called REST Web APIs, which communicate via HTTP requests. To describe our ideas we adopt in this section a simplified view of requests where we only focus on the request IRI. Of course, any implementation needs to take care of all the other details when connecting to APIs; we discuss this issue in the following section. Finally, we assume that all API responses are JSON documents. We focus on JSON APIs because it is arguably the most popular format to send responses over the HTTP protocol.

As an example of how a JSON API would work, consider an application containing information about weather conditions around the world. The application provides an API to allow other software to access this information. A hypothetical call to this API may be a request containing this URL:

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

by which a client is requesting the current weather conditions in Santiago, Chile. The API gives back an HTTP response containing the following JSON file:

```
{ "City": "Santiago", "Country": "Chile",
  "coord": {"lat": 33.27, "long": 70.40},
  "temperature": 25, "description": "Sunny"},
```

indicating that the temperature is 25 degrees and the day is sunny. To obtain these facts we need to navigate through the retrieved JSON document, using JSON navigation instructions. If we denote the JSON file above by J, then the current temperature and the weather description can be fetched using the instructions J["temperature"] and J["description"], respectively. Note that the value of the key coord is again a JSON document. Therefore if we wanted to obtain e.g. the latitude of the city, we would need to navigate first to this document, and then to the value of the appropriate key. For the document J above, this is done using the instruction J["coord"]["lat"], which first retrieves the value of the key "coord" (this is again a JSON document), and then navigates to the key "lat" inside the JSON document retrieved by J["coord"].

We always assume that the general structure of the JSON response is known by users; this can be achieved, for example, by including the schema of the response in the documentation of the API (see e.g. [4,8,17]).

3.2 The API call operator

A *parameterised IRI* is an IRI in which the query part may contain substrings of the form $\{?x\}$, for $?x$ in $\mathbf{V}$. For example, the following are parameterised IRIs:

```
http://weather.api/request?q={?city}
http://other.api/request?q={?city},{?country}
```

A parameterised URL is just a parameterised IRI which is also a URL. When talking about API calls we use the term IRI and URL interchangeably, with the former being more general. Given a parameterised IRI u, we denote by $\text{var}(u)$ the set of variables between brackets that are contained in u, and we generally say that u *uses* variables $\text{var}(u)$. For example, the sets of variables of the IRIs above are $\{?\text{city}\}$ and $\{?\text{city}, ?\text{country}\}$, respectively.

We now define the syntax of our BIND-from-API operator. Let P_1 be a graph pattern, U a parameterised IRI, $?x_1, \ldots, ?x_m$ a sequence of pairwise distinct variables, and $N_1, \ldots, N_m$ a sequence of JSON navigation instruction. Then the following is a SPARQL pattern, that we call a BIND-from-API pattern

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

The intuition behind the evaluation of this operator over a graph G is the following. For each mapping μ in $[\![P_1]\!]_G$ we instantiate every variable $?y$ in the parameterised IRI 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 $?x_1$. 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.

By our definition BIND-from-API patterns can appear anywhere usual SPARQL patterns can. For instance, in Example 1, we showcased how such patterns can be used inside a `WHERE` clause to obtain weather conditions from a climate API. Notice that in (1) the pattern P_1 can again be a BIND-from-API pattern, which allows us to use several $\mathsf{BIND\_API}$ operators in order to obtain results from one or more APIs inside a single query. Some use cases utilizing this functionality are displayed in Section 3.4 below.

3.3 Semantics

The semantics of a BIND-from-API pattern is defined in terms of the *instantiation* of a parameterised IRI U with respect to a mapping μ (denoted $\mu(U)$), which is simply the IRI that results by replacing each construct $\{?x\}$ in U with $\mu(?x)$, if $?x$ is in $\text{dom}(\mu)$, or with the empty string in case $?x$ is not in $\text{dom}(\mu)$.

Thus, every different mapping produces a different IRI, which we then use to produce an HTTP request to the API in the body of the IRI. Formally, given a parameterised IRI U and a mapping μ, we denote by $\text{call}(U, \mu)$ the result of the following process:

1. Instantiate U with respect to μ, obtaining the IRI $\mu(U)$.
2. Produce a request to the API signed by $(\mu(U))$, obtaining either a JSON document (in case the call is successful) or an error.

Informally, we refer to this process as the call to U with respect to the mapping μ. Note that we adopt the convention that HTTP requests that do not give back a JSON document result in an error, that is, $\text{call}(U, \mu) =$ `error` whenever the request using U does not result in a valid JSON document.

For instance, if we have a mapping μ, such that $\mu(?y) =$ `Santiago,CL`, and a parameterised IRI $U =$ `<http://weather.api/request?q={?y}>`, then $\mu(U) =$ `<http://weather.api/request?q=Santiago,CL>`. When this request is executed against the weather API in the IRI, the answer result is either a JSON document similar to the one from Example 1, describing the weather conditions in Santiago, Chile, or it is an error.

The evaluation of a BIND-from-API pattern P of the form (1) is defined as:

$$[\![P]\!]_G = \big\{\mu' \mid \mu \in [\![P_1]\!]_G,$$

$$\mu' = \mu \bigcup_{1 \leq j \leq m} \begin{cases} \{?x_j \mapsto \text{call}(U, \mu)[N_j]\}, & \text{if } \text{call}(U, \mu) \text{ is a JSON document,} \\ & \quad \text{and } \text{call}(U, \mu)[N_j] \text{ returns a basic JSON value,} \\ \emptyset, & \text{otherwise.} \end{cases}$$

Let us briefly discuss the idea of this definition. Consider a pattern P_1 = `?x rdf:type dbo:Place . ?x rdfs:label ?y`, and a parameterised IRI U = `<http://weather.api/request?q={?y}>`. Then the pattern

$$P = P_1 \texttt{ BIND\_API } U \texttt{ (["temperature"]) AS (?t)}$$

forms a BIND-from-API pattern. Suppose that we are evaluating P over some RDF graph G, and that we know that $[\![P_1]\!]_G$ contains the following mappings.

μ_1

$?x$	$?y$
`dbr:Tokio`	`Tokyo`

μ_2

$?x$	$?y$
`dbr:Berlin`	`Berlin`

The evaluation of P over G is then obtained by extending mappings in $[\![P_1]\!]_G$ using U. That is, we iterate over $\mu \in [\![P_1]\!]_G$ one by one, execute the call call(U, μ), and store the value call(U, μ)[`"temperature"`] into the variable `?t`, in case that the obtained JSON value is a string, a number, or a boolean value, and leave `?t` unbound otherwise. For example, if we assume that the calls are as follows,

$$\text{call}(\mu_1, U) = \ \{\texttt{"temperature": 22}\ \} \qquad \text{call}(\mu_2, U) = \ \texttt{error}$$

then the evaluation $[\![P]\!]_G$ will contain the following two mappings

μ_1

$?x$	$?y$	$?t$
`dbr:Tokio`	`Tokyo`	22

μ_2

$?x$	$?y$	$?t$
`dbr:Berlin`	`Berlin`	

Note that `?t` would also remain unbound if call(U, μ_2) returned a JSON document not containing the key `"temperature"`, or if the value of this key is not a basic JSON value.

Safety conditions. When considering a BIND-from-API pattern of the form (1), there are a few syntactic conditions which can help us avoid unnecessary API calls, or contradictory information. First, we require that all the variables appearing in the parameterised IRI U are also mentioned in the pattern P_1, in order to avoid API calls with completely unspecified values. We also forbid the variables $?x_1, \ldots, ?x_m$ to appear in the pattern P_1, to avoid the scenario where we are trying to rewrite the existing piece of information with the value obtained from the API. In the remainder of the paper, and in our implementation, we will always assume that the BIND-from-API patterns satisfy these two restrictions.

3.4 Further use-cases

In this subsection we provide several use cases where incorporating API data into SPARQL query answers could be useful. We begin with an example showing how the `BIND_API` operator can be used to combine multiple APIs in a single query.

Example 2. We find ourselves in London and want to visit a museum. We can use DBPedia to obtain standard information about London's museums, but we would also like to know which museums are currently open, and which ones are the most interesting. Thus, we set up to augment our answers with API information. We shall collect the opening times using the Yelp API, while the user feedback about the venue will be the most relevant tweet provided by Twitter API. YELP API allows a search for terms, and returns an object with key `"business"` that contains an array of places, ordered by relevance to the term. We select the first item, and then retrieve the value of `"is_closed"`, which is true or false depending on whether the museum is open or closed at the moment. Summing up, we navigate the response using `[businesses][0][is_closed]`. Twitter works in a similar way. When a search is requested, the API responds with an object with key `"statuses"`, containing several tweets ordered by relevance. We select the first status and look for the value of `"text"`, all of which corresponds to the instruction `["statuses"][0]["text"]`. The query is shown below[1]:

```
SELECT ?x ?n ?h ?t WHERE {
    ?x dbo:subject dbp:Museums_in_London .
    ?x rdfs:label ?n
    BIND_API <https://api.yelp.com/v2/search?term={?n}>
             (["businesses"][0]["is_closed"]) AS (?h)
    BIND_API <https://api.twitter.com/1.1/search/tweets.json?q={?n}>
             (["statuses"][0]["text"]) AS (?t)
    }
```

Figure 1 shows a sample answer for this query. □

Next, we use Yelp information in order to obtain restaurant recommendation based on our current location. This query also illustrates how the information obtained from a JSON API can be stored into multiple variables simultaneously.

Example 3. We are planning a visit to Chile, and would not want to miss the opportunity to try Chilean burgers. Since we do not know our itinerary, we decide to look for the best burger bars in each of Chile's cities and landmarks. We can use YAGO to fetch all the locations in Chile, and YELP will take care of choosing the best burger place at each location. The query is given below.

```
SELECT ?x ?n ?b ?r WHERE {
   ?x yago:isLocatedIn yago:Chile .
   ?x rdf:type yago:wikicat_Communes_of_Chile .
   ?x rdf:label ?n
   BIND_API <https://api.yelp.com/search?term=Burguers&location={?n}>
     (["businesses"][0]["name"], ["businesses"][0]["rating"])
                                                         AS (?b, ?r)
}
```

Figure 2 shows a sample answer for this query. This query also showcases how to retrieve multiple values from a single API call, by parsing the same JSON response with different navigation expressions (separated by commas), and binding the result of each instruction to a different variable. In this particular case, the first value is stored into `?b`, and the second one into `?r`. □

In our next use case we show how to combine the information on geographical location of world cities with the Open Weather API, in order to find climactic

[1] Some of the queries have been simplified to fit into the paper. For full queries please see [2]. All the queries use standard DBPedia and YAGO prefixes.

?x	?n	?h	?t
dbr:Chisenhale_Gallery	Chisenhale Gallery	false	Nothing to see here: the artist giving gallery staff a month off work

Fig. 1. Results from query in Example 2. Variable `?x` stores the IRI of the museum, `?n` its name, `?h` is true when the museum is closed (and false otherwise), and `?t` stores the latest relevant tweet about the museum.

?x	?n	?b	?r
yago:Santiago	Santiago de Chile	La Burguesía	4.5
yago:Torres_del_Paine,_Chile	Torres del Paine	Masay	4.5
yago:Río_Bueno,_Chile	Río Bueno	Mcdonald's	2.0

Fig. 2. Results from query in Example 3. Variable `?x` stores the IRI of the location, `?n` its name, `?b` stores the name of the restaurant and `?r` its rating.

?x	?y	?xtemp	?ytemp
dbr:Xinyi,_Guangdong	dbr:Calama,_Chile	289	288
dbr:Wuhai	dbr:Valdivia	283	281

Fig. 3. Results from query in Example 4. Variables `?x` and `?y` store the IRI of the antipode cities, and `?xtemp` and `?ytemp` their respective temperature (in Kelvin).

antipodes guaranteed by the famous Borsuk-Ulam theorem [1], which states that on the surface of the Earth there are always two point which are antipodes (i.e. they are on opposite sides of the planet), and which have the same temperature and air pressure. For simplicity our query finds antipodes with the same temperature. Extending the result to include air pressures as well is straightforward.

Example 4. We use a function `ANTIPODE(?xLat,?xLon,?yLat,?yLon)` that receives pairs of geographical coordinates (`?xLat,?xLon`) and (`?yLat,?yLon`) , and outputs true if these coordinates are antipodes on Earth, with a possible error margin of 0.25 degrees for both latitude and longitude. For full details see [2].

```
SELECT ?x ?y ?xtemp ?ytemp WHERE {
   ?x rdf:type dbo:City . ?x rdfs:label ?xname .
   ?x geo:long ?xlon . ?x geo:lat ?xlat .
   ?y rdf:type dbo:City . ?y rdfs:label ?yname .
   ?y geo:long ?ylon . ?y geo:lat ?ylat .
   FILTER ANTIPODE(?xLat,?xLon,?yLat,?yLon)
   BIND_API <http://api.openweathermap.org/data/2.5/weather?q={?x2}>
       (["main"]["temp"]) AS (?t)
   BIND_API <http://api.openweathermap.org/data/2.5/weather?q={?y2}>
       (["main"]["temp"]) AS (?t2)
   FILTER ( (?t >= 0.9*?t2) && (?t <= 1.1*?t2) )
   }
```

After obtaining all pairs of cities from DBPedia, we filter out those which are not antipodes, and then use BIND-from-API calls to retrieve the temperature of both cities, filtering again those in which the temperature is not the same (we allow a 10% error margin). Figure 3 shows a sample answer for this query. □

4 Implementing API calls

There are two main challenges behind the implementation of our proposal. First, there is a security component: the communication to APIs usually involves some authentication protocol which can be either a special key that needs to be requested beforehand or a more involved process like oauth [10]. Thus, in order to enable SPARQL systems to process API calls we need a way of providing such strategies. We discuss how to do this in the following section.

The second issue is how to process queries that use the BIND-from-API operator. The straightforward way of doing this is of course to extend the way systems process BIND assignments. However, we provide here a more lightweight option that can be mounted on top of a SPARQL processor, and whose processing can be optimised in a much cleaner way. We discuss the main algorithm in Section 4.2 and some optimisation options in Section 5.

Full details and the source code of our implementation are available at [2].

4.1 Communication and authentication

As we have mentioned, the communication with APIs usually requires some degree of authentication. While there are several authentication protocols that are widely used, there is still no standard for this process. In our opinion there are two protocols that really stand out in terms of popularity: authentication via an API key, and OAuth. The first protocol is based on a token that is obtained either from a Web Service or is given directly to the users. This token is then passed in the IRI of each request to the API, and the authorisation depends only on the validity of the token. The OAuth protocol can be seen as a safer alternative to the API key. Here, the user needs to obtain various keys and secret tokens associated to these keys. Before a particular request can be done to the API, the user needs to request a specific token key for that request, attaching all the necessary credentials. Once these credentials are verified, the API responds with the access token, which can now be used to proceed with the API call.

Regardless of the popularity of the protocols above, in order to provide a general framework, we abstract from a particular process, and instead provide a general solution based on the assumption that all access protocols are carried over with HTTP requests. Under these assumptions, any different strategy can then be implemented following the same principles: for each strategy a number of keys are required to the user, and, whenever there is more than one API call, each request URL must be assigned to a precise strategy (the system can also learn which strategy needs to be associated simply by trial and error). In our implementation [2] we have decided to store these tuning values in a separate JSON file, but without much effort one could devise a way of integrating them into queries, (for example by providing an extra form in a SPARQL endpoint).

4.2 Query Evaluation

Our implementation does not modify the inner workings of the BIND operator in Jena, but focuses instead on obtaining first all mappings that will generate an

API call, and then evaluating all of these calls at once. As we see in the following section, this algorithm allows us to define a number of rules for optimising queries with API calls in a clear way. The other advantage of this approach is that it can be implemented (as we did) with just minor modifications into the query engines of existing SPARQL systems. To do this, whenever a pattern of the form

$$P \equiv P_1 \ \ \mathsf{BIND\_API} \ \ U \ (N_1, N_2, \ldots, N_m) \ \ \mathsf{AS} \ (?x_1, ?x_2, \ldots, ?x_m)$$

is found, we process it over our local database G as follows:

1. First, we evaluate $[\![P_1]\!]_G$ by running `'SELECT * WHERE {P1}'`;
2. In the next stage we do the following steps for each $\mu \in [\![P_1]\!]_G$:
3. Execute call(U, μ) using the appropriate strategy (see Section 4.1);
4. For $1 \leq i \leq n$, extend μ with $?x_i$, execute N_i over call(U, μ) and, if a proper SPARQL literal is obtained, store this literal into $?x_i$.
5. Now serialise the (extended) mappings obtained in the first four steps using the `VALUES` operator, to allow it to be used by the next graph pattern inside the `WHERE` clause in which it appears.

Regarding the final step, the obtained mappings need to be serialised in case P is followed by another graph pattern P_2. In particular, if we are processing a query of the form `SELECT * WHERE` $\{P \ . \ P_2\}$, with P as above, then P_2 needs to be able to access the values from the mappings matched to P.

5 Experiments and Optimisation

API calls require a completely different treatment that the rest of SPARQL, as it is conceivable that the time required to process API calls is going to dominate query processing times by a huge margin. To illustrate this we ran the four queries from Example 1 thorough Example 4 over a piece of YAGO database of size 300 MB (queries from Examples 1 and 3) and a piece of DBPedia database of size 1.3GB (queries in Examples 2 and 4). Our implementation of the BIND-from-API operator was carried out using the Apache Jena framework [12] as an add-on to the ARQ SPARQL query engine, and all the experiments were ran on a MacBook Air with an Intel Core i5 1.3 GHz processor and 4GB of main memory. Figure 4 shows how much time is consumed by API calls in comparison to the total runtime of the queries (the time is averaged over five runs). As we can see, API calls can require up to 99% of total query processing time (as was the case with Examples 2 and 3), and even when just a few calls are needed this may end up taking at least a third of the total processing time.

Thus, the bottleneck when evaluating queries enabling API calls will most likely be the amount of calls the query makes. This is due to the fact that in general we have no control over how quickly we receive the data from the API, as this depends on the efficiency of the API server and on the internet connection: two factors that we can never control. It is therefore reasonable to consider optimisations that minimise the number of API calls a query can make. In the following we describe three approaches for doing this: avoiding duplicate calls, reformulating query plans, and caching.

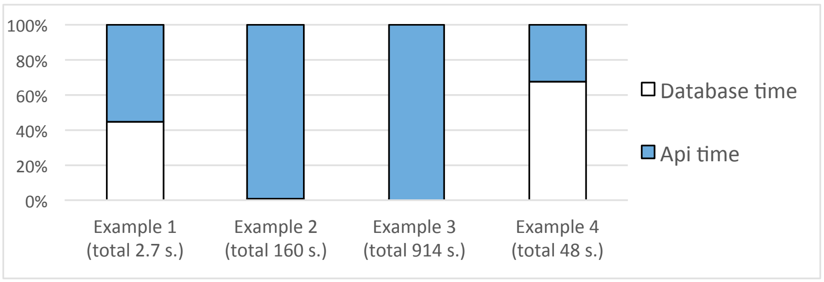

Fig. 4. Proportion of the query processing time consumed by the API (blue) against the database (white), for queries in Example 1 thorough Example 4. Values are the average of several runs, total query time is indicated between brackets.

5.1 Removing duplicate calls

In many scenarios we might end up with several API calls for the same URL. For example, a simple query of the form

```
SELECT ?x ?t WHERE {?x ex:label1 ?z . ?x ex:label2 ?n
    BIND_API <http://localhost:3000/{n}/5>(["time"]) AS (?t) }
```

might contain several mappings where the same value is bound to the variable $?n$, resulting in several calls using the same value, which is clearly suboptimal. To eliminate this behaviour we proceed as follows. We produce first a value of all distinct values $?n$ that can be produced in the query, produce one call for each value, and then join with the result of the rest of the query. More generally, we can replace any BIND-from-API pattern P of the form

$$P_1 \ \textsf{BIND\_API} \ U \ (N_1, N_2, \dots, N_m) \ \textsf{AS} \ (?x_1, ?x_2, \dots, ?x_m), \tag{2}$$

where U uses variables $y_1, \dots, y_n$, with the SPARQL pattern

$$\begin{aligned}&\{\textsf{SELECT}\, \mathrm{var}(P_1) \ \textsf{WHERE} \ \{P_1\}\} \ . \\ &\qquad\qquad\qquad \{\textsf{SELECT} \ \textsf{DISTINCT} \ y_1, \dots, y_n \ \textsf{WHERE} \ P_1\} \\ &\qquad\qquad\qquad\qquad \textsf{BIND\_API} \ U \ (N_1, N_2, \dots, N_m) \ \textsf{AS} \ (?x_1, ?x_2, \dots, ?x_m)\}\end{aligned}$$

It is easy to see that these patterns are equivalent. However, this transformation introduces a significant increase of the workload of our local database, so the usage depends heavily on how slow we expect APIs to respond: the slower the API response time, the better that this optimisation performs.

We test this optimisation by posing the query in the example above over four different synthetic databases, while varying the response time of APIs. The synthetic databases contain 10.000 triples, and are constructed so that the amount of duplicates in the query answer equals 0%, 25%, 50% and 75%, respectively. We show the results in two cases: when the response time is instantaneous (Figure 5, left) and when the response time is 10 milliseconds (Figure 5, right). As the figures indicate, this optimisation is pointless when API times are dismal, but becomes crucial as the time for API responses increases.

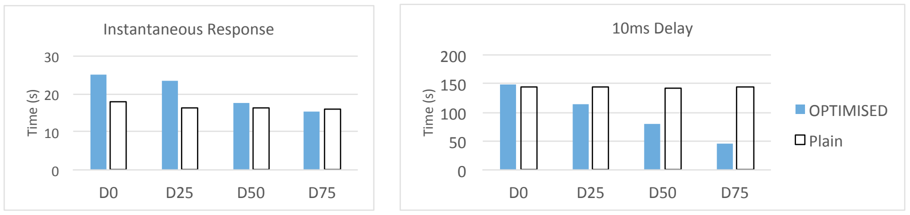

Fig. 5. Query times for the query from Section 5.1 over databases D0, D25, D50, D75 (the number indicates the percentage of duplicates), under two settings: when API response is instantaneous (left) and when each response takes 10 milliseconds (right).

5.2 Reducing API calls by optimising query plans

Even if we ensure that no duplicate API calls are to be made, one could still reduce the number of calls by producing an equivalent reformulation of the query that nevertheless reduces the amount of calls. In general, the idea is as follows. Given a pattern of the form $\{P_1 \ \mathsf{BIND\_API} \ U \ N_1 \ \mathsf{AS} \ ?x_1\} \ . \ P_2$, where U mentions variables $?y_1, \ldots, ?y_n$, obtain patterns Q_1 and Q_2 such that (1) queries $P_1 \ . \ P_2$ and $Q_1 \ . \ Q_2$ are equivalent (that is, they give the same answer independently of the graph) and such that the size of the set of tuples $\{(\mu(?y_1), \ldots, \mu(?y_n)) \mid \mu \in [\![Q_1]\!]_G\}$ is minimal for every graph G. Unfortunately, it is not difficult to show that this decomposition in general may not exist. However, as a starting point we point out some simple heuristics, but we believe this is a very interesting direction for further research.

Pushing Filters. This optimisation applies on all patterns $P \ . \ P_2$, where P is a BIND-from-API pattern of the form (2), and P_2 uses a `FILTER` clause which does not mention the variables introduced in `BIND_API`, and is not nested using `UNION` or `OPTIONAL`. Then we can safely apply this `FILTER` clause to the mappings obtained by evaluating P_1 (the base part of P). This will generally reduce the number of API calls, since we now have fewer mappings to process.

Rearranging joins. Rearrangement of joins, in general, is not guaranteed to reduce the number of total mappings. However, if we only care for distinct bindings of the variables used in BIND_API calls, then all we need is a greedy rearrangement of triples. To see this, consider the following three equivalent patterns:

```
Q1: ?x ex:a ?y BIND_API <http://api.org/?q={?x}> (["value"]) AS (?v)
       ?x ex:b ?z
Q2: ?x ex:b ?z BIND_API <http://api.org/?q={?x}> (["value"]) AS (?v)
       ?x ex:a ?y
Q3: ?x ex:a ?y . ?x ex:b ?z
        BIND_API <http://api.org/?q={?x}> (["value"]) AS (?v)
```

None of these queries can be guaranteed to give the least amount of mappings. However, query `Q3` is minimal in terms of bindings for $?x$: regardless of the RDF graph to which is posed, it will always give a lower or equal number of distinct bindings for $?x$ when compared to the other two queries.

Thus, by combining a greedy search with our strategy to remove duplicates we can devise the following rule for rearranging joins. Given a pattern P . P_2, where P is a BIND-from-API pattern of the form (1), and P_2 is a concatenation of triple and filter patterns, rewrite P_1 by adding all filter and triple patterns in P_2 that do not mention any of the variables in $\{?x_1, \dots, ?x_m\}$.

5.3 Reducing API calls by caching

Clearly the best way of reducing API calls is to not do them at all, because we already have them in the system. This is important even if we are dealing with just a single query, as several mappings may actually require the same API calls. For example, recall the query of Example 4. The SPARQL part of this query uses geo locations to compute all pairs (c_1, c_2) of cities that are antipodes. Then, for each pair a call to Open Weather API was used to retrieve the temperature in both cities. Now obviously c_1 is an antipode of c_2 if and only if c_2 is an antipode from c_1, so at the very least this query will make two calls per each pair of antipode cities, and we cannot get rid of this double call by removing duplicates, since (c_1, c_2) and (c_2, c_1) are different pairs. If we put each queried city in a cache, this results in a reduction of at least 50% in calls (there is even more repetition thanks to the 0.25 degree margin we allow for antipodes).

This is precisely what we do: every time a call to an API is produced, we cache the exact IRI that was used in this call, as well as the resulting JSON document (if the call was successful). Thus, before each call we first retrieve the IRI in the cache (which is a very fast operation since we maintain an index on the cache), and only proceed with the call if we cannot find the answer. Note that since at the moment we are only caching answers during a single query execution, we do not loose generality nor freshness in our answers. However, we could also decide to maintain a cache over time, ranging across the execution of different queries, at the sacrifice of freshness of answers.

Our experiments with the implemented cache confirm this intuition: We were able to reduce the average time that it took us to compute the query of Example 4 by 35%, from almost 48 seconds to 31 seconds, thanks to a 55% decrease in the portion of the query time that was consumed by the calls (details in the appendix). We did not try parallelising requests in order to maintain a reasonable level of politeness when calling the API services.

6 Conclusions and Future Work

In this paper we propose a way to extend the functionality of SPARQL by allow it to connect to REST APIs returning JSON. We describe the syntax and the semantics of this extension, and show how it can be implemented on top of existing SPARQL engines. Although in the presentation we focus on JSON APIs, the BIND-from-API operator can easily be extended to support different return formats without changing the underlying SPARQL engine.

In future work we would like to demonstrate how API calls can be supported in a public SPARQL endpoint. The way we envision this is by an interface in which we combine a predefined set of supported APIs with the option to enter new authentication tokens for other APIs. Another line of work we plan to pursue is to support automatic entity resolution based on an API answer, thus allowing us to transform API information back into IRIs to be used again by SPARQL, instead of just literals. In this way we will be able to determine if the JSON we obtain represents the same entity as an IRI already existing in our database, thus providing us with extra information. Finally, another limitation that we would like to overcome is the possibility of generating several mappings form a single JSON document (such as, for example, generating a new mapping for each of the top 5 burger joints in each location of Chile). In order to do this we need the support of a proper query language for JSON documents, which unfortunately does not exist at the moment in a standardised form.

References

1. Borsuk-Ulam Theorem. https://en.wikipedia.org/wiki/Borsuk-Ulam_theorem.
2. Online Appendix. http://dvrgoc.ing.puc.cl/APIs.
3. Open Data Portal. https://open-data.europa.eu/en/linked-data.
4. Swagger: The World's Most Popular Framework for APIs. http://swagger.io/, 2015.
5. C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In *ISWC 2013*, pages 277–293, 2013.
6. R. Battle and E. Benson. Bridging the semantic web and web 2.0 with representational state transfer (REST). *J. Web Sem.*, 6(1):61–69, 2008.
7. T. Bray. The javascript object notation (json) data interchange format. 2014.
8. F. Galiegue and K. Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.
9. P. T. Groth, A. Loizou, A. J. G. Gray, C. A. Goble, L. Harland, and S. Pettifer. Api-centric linked data integration: The open PHACTS discovery platform case study. *J. Web Sem.*, 29:12–18, 2014.
10. D. Hardt. The oauth 2.0 authorization framework. 2012.
11. S. Harris and A. Seaborne. SPARQL 1.1 query language. *W3C Recommendation*, 21, 2013.
12. The Apache Jena Manual. http://jena.apache.org, 2015.
13. G. Klyne and J. Carrol. Resource Description Framework (RDF): Concepts and Abstract Syntax, February 2004.
14. N. Kobayashi, M. Ishii, S. Takahashi, Y. Mochizuki, A. Matsushima, and T. Toyoda. Semantic-json: a lightweight web service interface for semantic web contents integrating multiple life science databases. *Nucleic Acids Research*, 39(Web-Server-Issue):533–540, 2011.
15. T. B. Lee. Linked data. https://www.w3.org/DesignIssues/LinkedData.html, 2007.
16. H. Müller, L. Cabral, A. Morshed, and Y. Shu. From restful to SPARQL: A case study on generating semantic sensor data. In *ISWC 2013*, pages 51–66, 2013.
17. F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *WWW 2016*, pages 263–273, 2016.
18. L. Rietveld and R. Hoekstra. YASGUI: not just another SPARQL client. In *ESWC 2013*, pages 78–86, 2013.
19. W3C. Linked data. https://www.w3.org/standards/semanticweb/data, 2015.

Average query execution times

Query	**total time**	**API time**	**DB time**	**num. of mappings**
Ski Hokkaido	2.7093	1.5010	1.2084	8
Museums	159.8161	157.9474	1.8687	34
Hamburgers	914.0295	911.5008	2.5287	463
Antipodes	47.7115	15.9211	32.5852	45

Ski Hokkaido	**total time**	**API time**	**DB time**	**num. of mappings**
1	3.6404	2.5657	1.0747	8
2	2.3282	1.2398	1.0884	8
3	2.3691	1.2341	1.1350	8
4	2.6122	1.2399	1.3723	8
5	2.5968	1.2254	1.3714	8

Museums	**total time**	**API time**	**DB time**	**num. of mappings**
1	154.2847	152.6703	1.6144	34
2	169.2832	166.7504	2.5328	34
3	158.8432	157.5133	1.3299	34
4	156.6222	155.1668	1.4554	34
5	160.0472	157.6362	2.4110	34

Hamburgers	**total time**	**API time**	**DB time**	**num. of mappings**
1	922.0850	919.4614	2.6236	463
2	935.0920	933.5253	1.5667	463
3	884.9116	881.5157	3.3959	463

Antipodes	**total time**	**API time**	**DB time**	**num. of mappings**
1	46.2403	14.5797	31.6606	45
2	48.5434	14.3401	34.2033	45
3	45.2993	14.2187	31.0806	45
4	47.2659	14.2752	32.9907	45
5	51.2084	22.1919	32.9907	45

Cached Antipode query

C. Antipodes	**total time**	**API time**	**DB time**	**num. of mappings**
1	37.8760	7.3416	30.5344	45
2	39.9109	8.0415	31.8694	45
3	38.7690	6.8254	31.9436	45
4	36.7499	6.8144	29.9355	45
5	38.5756	6.8010	31.7746	45
AVG	38.3763	7.1648	31.2115	45