Querying APIs with SPARQL

Matthieu Mosser, Fernando Pieressa, Juan Reutter,
Adrián Soto, Domagoj Vrgoč

Pontificia Universidad Católica de Chile

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 documents. In this work we propose a way of extending SPARQL with the option to consume JSON APIs and integrate the obtained information into SPARQL query answers. Looking to evaluate these queries as efficiently as possible we present an algorithm that produces "worst-case optimal" query plans that reduce the number of requests as much as possible. We also do a set of experiments that empirically confirm the optimality of our approach.

1 Introduction

The 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 applications to access data obtained from different sources. However, the majority of the data available on the Web today is still not published as RDF, which makes it difficult to connect it to Semantic Web services. A huge amount of this data is made available through Web APIs which use a variety of different formats to provide data to the users.

We think that is important to make all of this data available to Semantic Web technologies, in order to create a truly connected Web. We propose an extension of SPARQL that allows us to connect to Web APIs, extending query answers with data obtained from a Web Service, in real time and without any setup.

With the ability of querying endpoints and APIs in real time we face an even more challenging task: How do we evaluate such queries? Connecting to APIs poses an interesting new problem from a database perspective, as the bottleneck shifts from disk access to the amount of API calls. Hence, to evaluate these queries efficiently we need to understand how to produce a query plan for them that minimizes the number of calls to the API.

Work supported by Millennium Institute for Foundational Research on Data (IMFD), Chile. This work was presented in ESWC 2018. [7]

2 SERVICE-to-API Queries

We extended the `SERVICE` operator to allow SPARQL to query Web APIs. We call a query that uses this extended `SERVICE` a **SERVICE-to-API** query. We assume the reader is familiar with the syntax and semantics of SPARQL 1.1 query

language [6]. We concentrate on the so-called REST Web APIs, which communicate via HTTP requests,assuming that all API responses are JSON documents. To illustrate how our extension works we will use the following example:

Example 1. We find ourselves in Scotland in order to do some hiking. We obtain a list of all Scottish mountains using the WikiData SPARQL endpoint, but we would prefer to hike in a place that is sunny. This information is not in WikiData, but is available through a weather service API called `weather.api`. This API implements HTTP requests, so for example to retrieve the weather on Ben Nevis, the highest mountain in the UK, we can issue a GET request with the IRI:

```
http://weather.api/request?q=Ben_Nevis
```

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

```
{"timestamp": "24/10/2017 11:59:07", "temperature": 3,
"description": "clear sky", "coord": {"lat": 56.79, "long": -5.02}}
```

Therefore, to obtain all Scottish mountains with a favourable weather all we need to do is call the API for each mountain on our list, keeping only those records where the weather condition is `"clear sky"`. One can do this manually, but this quickly become cumbersome, particularly when the number of API calls is large. Instead, we propose to extend the functionality of SPARQL `SERVICE`, allowing it to communicate with JSON APIs such as the weather service above. For our example we can use the following (extended) query:

```
SELECT ?x ?l WHERE {
   ?x wdt:instanceOf wd:mountain . ?x wdt:locatedIn wd:Scotland .
   ?x rdfs:label ?l .
   SERVICE <http://weather.api/request?q={?l}>{(["description"]) AS (?d)}
   FILTER (?d = "clear sky")
  }
```

The first part of our query is meant to retrieve the IRI and label of the mountain in WikiData. The extended `SERVICE` operator then takes the (instantiated) URI template where the variable `?l` is replaced with the label of the mountain, and upon executing the API call processes the received JSON document using an expression `["description"]`, which extracts from this document the value under the key `description`, and binds it to the variable $?d$. Finally, we filter out those locations with undesirable weather conditions. □

We proposed a way to implement the overloaded `SERVICE` operation on top of any existing SPARQL engine. To do so, we partition each query using this operator into smaller pieces, and evaluate these using the original engine whenever possible. The answers given by the engine are extended with the results provided by the APIs. A full specification can be found in [7].

But can we optimize this queries? We discover that for SERVICE-to-API queries the bottleneck are the API calls. This is mostly because HTTP requests are slower that disk access and its something that we cannot control. So if we want to evaluate queries as efficiently as possible we need to do the least amount of API calls as possible. Then, can we reformulate query plans to make sure we are making as few calls as possible?

3 A Worst-case optimal algorithm

What we did is to propose an algorithm that is optimal in the worst case. This algorithm does not make more calls than the number we would need in the worst case over all graphs and APIs of a given size. For matters of space we will not explain all the algorithm and the proofs. The details can be found in [7].

Our algorithm is inspired by the optimal plan exhibited in [1,5] for conjunctive queries. To illustrate it, consider the SERVICE-to-API query of the example1. Note that the query is formed of basic graph patterns and a SERVICE-to-API pattern. We treat each basic graph pattern as a relation and each SERVICE-to-API as a relation with access methods (see e.g. [2,4]).

Then we can think that a SERVICE-to-API query has the form $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$. For the sake of presentation we consider that R_j $(1 \leq j \leq m)$ can represent a basic graph pattern or a SERVICE-to-API pattern where its attributes are the variables of the basic graph pattern or the input and output variables of the API. Then, let $A_1, \ldots, A_n$ be an enumeration of all attributes (variables of SPARQL) involved in Q, in order of their appearance. Starting from Q, we construct a query $Q^* = \Delta_n$, where the sequence $\Delta_1, \ldots, \Delta_n$ is defined as:

$$\Delta_1 = \pi_{A_1}(R_1) \bowtie \ldots \bowtie \pi_{A_1}(R_m).$$

$$\Delta_i = \Delta_{i-1} \bowtie \pi_{A_1,\ldots,A_i}(R_1) \bowtie \ldots \bowtie \pi_{A_1,\ldots,A_i}(R_m).$$

The idea is to process the query variable by variable. First we obtain the result of the intersection of all A_1 from the BGPs, and then we extend such BGPs with the values obtained from APIs where their input is the attribute A_1. Then we resolve the join for BGPs with variables A_1 and A_2 (considering the results of the previous iteration) and we extend the results with the output of the APIs with inputs A_1 and A_2. We continue this process until answer the query.

Our main result is the following. Take any SERVICE-to-API query Q, and a database D. Denote by $M_{Q,D}$ the maximum size of the projection of any relation appearing in Q over a single attribute in the database D. Furthermore, let $2^{\rho^*(Q,D)}$ be the AGM bound [1] of the query Q over D, i.e. the maximum size of the output of Q over any relational database having the same number of tuples in each relation as D. Then we can prove the following:

Theorem 1. *Any feasible join query under access methods Q can be evaluated over any database instance D using a number of calls in*

$$O(M_{Q,D} \times 2^{\rho^*(Q,D)}).$$

4 Experiments

We wanted to give empirical evidence that the worst-case optimal algorithm of Section 3 is indeed a superior evaluation strategy for executing queries that use API calls. To construct a benchmark for SERVICE-to-API patterns we reformulate the queries from the Berlin benchmark [3] by designating certain patterns in a query to act as an API call.

Algorithms. We consider three algorithms for `SERVICE`-to-API patterns: (1) *Vanilla*, a naive implementation without optimization; (2) *Without duplicates*, the base algorithm that uses caching to avoid doing the same API twice; and (3) *WCO*, the worst-case optimal algorithm of Section 3.
Results. The number of API calls done for each of the three versions of our algorithm are shown in Table 1. As we see, avoiding duplicate calls reduces the number of calls to some extent, but the best results are obtained when we use the worst-case optimal algorithm. We also measured the total time taken for the evaluation of these queries. Query times range from over 8000 seconds to just 0.7 seconds for the Vanilla version, and in average the use of *WCO* reduces by 40% the running times of the queries.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q7**	**Q8**	**Q10**	**Q12**	**AVG**
Vanilla	5332	77	5000	5066	2254	15	1	7	1	0%
W/O Duplicates	4990	3	4990	4990	608	15	1	7	1	20%
WCO	2971	0	3284	4571	608	13	0	0	1	53%

Table 1. The number of API call per query for each algorithm. WCO plans average 53% reduction in API calls.

5 Conclusion

In this paper we propose a way to allow SPARQL queries to connect to HTTP APIs returning JSON. We give an intuition of the syntax and the semantics of this extension, discuss how it can be implemented on top of existing SPARQL engines, show an sketch of a worst-case optimal algorithm for processing these queries, and show the usefulness of this algorithm in practice. In future work, we plan to support formats other than JSON, and explore how to support it in public endpoints.

References

1. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.
2. M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6):690–701, 2015.
3. C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
4. A. Calì and D. Martinenghi. Querying data under access limitations. In *ICDE 2008*, pages 50–59, 2008.
5. M. Grohe. Bounds and algorithms for joins via fractional edge covers. In *In Search of Elegance in the Theory and Practice of Computation*. Springer, 2013.
6. S. Harris and A. Seaborne. SPARQL 1.1 query language. *W3C*, 2013.
7. M. Mosser, F. Pieressa, J. L. Reutter, A. Soto, and D. Vrgoc. Querying apis with SPARQL: language and worst-case optimal algorithms. In *ESWC 2018*, pages 639–654, 2018.