Querying APIs with SPARQL: language and worst case optimal algorithms

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, thus obtaining a query language allowing to bring data from the "traditional" Web to the Semantic Web. Looking to evaluate these queries as efficiently as possible, we show that the main bottleneck is the amount of API requests, and 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 for applications of increasing complexity since it enables them to access data obtained from different sources, or describing different entities. The most common way of accessing this information is through SPARQL endpoints; SPARQL is the standard language for accessing data on the Semantic Web [20], and a SPARQL endpoint is a simple interface where users can obtain the RDF data available on the server by executing a SPARQL query.

In the Web context it is rarely the case that one can obtain all the needed information from a single data source, and therefore it is necessary to draw the data from multiple servers or endpoints. In order to address this, a specific operator that allows parts of the query to access different SPARQL endpoints, called `SERVICE`, was included into the latest version of the language [30].

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. It is therefore important to make all of this data available to Semantic Web technologies, in order to create a truly connected Web. One way of achieving this is to extend the `SERVICE` operator of SPARQL with the ability to connect to Web APIs in

the same way as it connects to other SPARQL endpoints. In this paper we make a first step in this direction by extending `SERVICE` with the option to connect to JSON APIs and incorporate their data into SPARQL query answers. We picked JSON because it is currently the most popular data format in Web APIs, but the results presented in the paper can easily be extended to any API format.

By allowing SPARQL to connect to an API we can extend the query answer with data obtained from a Web service, in real time and without any setup. Use cases for such an extension are numerous and can be particularly practical when the data obtained from the API changes very often (such as weather conditions, state of the traffic, etc.). To illustrate this let us consider 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. □

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. For example, when evaluating the query in Example 1, about 80% of the time is spent in API calls. This is mostly because HTTP requests are slower that disk access, something we cannot control. To gauge the time taken for APIs to respond to a `GET` request we did a quick study of five popular Web APIs. The results presented in Table 1 show us the minimum, the maximum, and the average time over our calls for each API.

	Yelp!	**Twitter**	**Open Weather**	**Wikipedia**	**StackOverflow**	**All**
min	0.4	0.4	0.4	0.8	0.3	0.3
max	1.3	0.8	1.4	1.3	1.5	1.5
avg	1.1	0.5	0.6	1.0	0.6	0.76

Table 1. Min, max, and average response time of popular Web APIs based on ten typical calls they support.

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.

Contributions. Our main contributions can be summarized as follows:
- *Formalization.* We formalize the syntax and the semantics of the `SERVICE` extension which supports communication with JSON APIs. This is done in a modular way, similar to the SPARQL formalization of [28], making it easy to incorporate this extension into the language standard.
- *Implementation.* We provide a fully functional implementation of the extended `SERVICE` operator within the Apache Jena framework, and test its functionality on a range of queries over real world and synthetic data sources. We also set up a demo at [2] for trying out the new functionality.
- *Optimization.* Given that the most likely bottleneck for our queries is the number of API calls, we design, implement and test a series of optimizations based on the AGM bound [8,27] for estimating the number of intermediate results in relational joins, resulting in a worst case optimal algorithm for evaluating a large fragment of SPARQL patterns that uses remote `SERVICE` calls.

Related work. Standard `SERVICE` that connects to SPARQL endpoints has been extensively studied in the literature [5,6,7,25,24]. The main conclusions regarding efficiency in this context resonate with our argument that the amount of calls to external endpoints is the main bottleneck for evaluation. For standard SERVICE there are several techniques we can use to alleviate this issue (see e.g. [6,7]), and for our implementation we opt for the one that minimizes the database load. In terms of bringing arbitrary API information into SPARQL most of the works [23,26,31,14] are based on the idea of building RDF wrappers for other formats. This is somewhat orthogonal to our approach and can be prohibitively expensive when the API data changes often (like in Example 1). The most similar to our work are the approaches of [15,16,9] that incorporate API data directly into SPARQL, but do not provide a worst-case optimal implementation, nor formal semantics of the extended `SERVICE` operation. Our paper is a continuation of the demo presentation [22].

Organization. We recall standard notions in Section 2. The formal definition and examples of the extended `SERVICE` are given in Section 3. The worst-case optimal algorithm for evaluating queries that use `SERVICE` is presented in Section 4. Experimental evaluation is given in Section 5. For space reasons detailed proofs can be found in our online appendix [1], and further examples at [2].

2 Preliminaries

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, although our proposal can easily be extended to deal with datasets with multiple graphs.

SPARQL Queries. We assume the reader is familiar with the syntax and semantics of SPARQL 1.1 query language [20], as well as the abstraction proposed in [28]. We use this abstraction for our theoretical work, but examples are stated in the standard syntax.

Let us recall some basic notions from [28] that will be used later on. We define queries via *graph patterns*. Graph patterns are defined over terms $\mathbf{T}$ and an infinite set $\mathbf{V} = \{?x, ?y, \ldots\}$ of *variables*. The basic graph pattern is called a *triple pattern*, and is a tuple $t \in (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V})$. All other graph patterns are defined recursively, using triple patterns, and operators `AND`, `OPT`, `UNION`, `FILTER` and `SERVICE`. We consider `SERVICE` patterns of the form (P_1 `SERVICE` a P_2), with $a \in (\mathbf{I} \cup \mathbf{V})$ and P_1, P_2 graph patterns. If P is a graph pattern we denote the variables appearing in P by $\mathrm{var}(P)$. Finally, we define *SPARQL queries* as expressions of the form `SELECT` W `WHERE` { P }, where P is a graph pattern, and W a set of variables.

We use the usual semantics of SPARQL, defined in terms of *mappings* [20,28]; that is, partial functions from the set of variables $\mathbf{V}$ to IRIs. 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 $\mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1)$. Note that if two mappings μ_1 and μ_2 have no variables in common they are always compatible, and that the empty mapping $\mu_\emptyset$ is compatible with any other mapping. For sets M_1 and M_2 of mappings we define their join as $M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 : \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}$. Given a graph G and a pattern P, we denote the *evaluation* of a graph pattern P over G as $[\![P]\!]_G$. We refer to [20] for a full specification of the semantics.

3 Enabling SPARQL to make JSON calls

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, and we only consider requests of type GET. Of course, any implementation needs to take care of many other details when connecting to APIs (e.g. authentication). Our implementation takes this into consideration, but for space reasons here we just focus on the problem of evaluating these queries. An endpoint allowing the users to try the `SERVICE`-to-API functionality can be found at [2]), and the source code of our implementation can be found at [3].

We assume that all API responses are JSON documents, and we use JSON *navigation conditions* to navigate and retrieve certain pieces of a JSON document, analogous to the way JSON documents are navigated in programming languages. For example, if we denote by J the JSON received in Example 1, we use J[`"temperature"`] to obtain the temperature and J[`"coord"`][`"lat"`] to obtain the latitude. 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. [29,17]).

3.1 Syntax and semantics of the extended SERVICE operator

A *URI Template* [21] is an URI in which the query part may contain substrings of the form $\{?x\}$, for $?x$ in $\mathbf{V}$. For example, the following is a URI template:

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

The elements inside brackets are replaced by concrete values in order to make a request. In what follows, we will refer to the variables in such substrings of a URI template U as the variables of U, and denote them with $\text{var}(U)$.

Here is how we propose to extend SERVICE to enable calls to APIs. Let P_1 be a SPARQL pattern, U a URI template using only variables that appear in P_1, $?x_1, \ldots, ?x_m$ a sequence of pairwise distinct variables that do *not* appear in P_1, and $N_1, \ldots, N_m$ a sequence of JSON navigation instructions. Then the following is a SPARQL pattern, that we call a SERVICE-to-API pattern:

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

The intuition behind the evaluation of this operator over a graph G is the following. For each mapping μ in the evaluation $[\![P_1]\!]_G$ 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[1]. 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, assuming the instruction returns a basic JSON value, store this value into $?x_1$. Similarly, the value of N_2 applied to J is stored into $?x_2$, and so on. The mapping μ is then extended with the new variables $?x_1, \ldots, ?x_m$, which have been assigned values according to J and N_i. Notice that in (1) the pattern P_1 can again be an overloaded `SERVICE` pattern connecting to another JSON API, thus allowing us to obtain results from one or more APIs inside a single query.

[1] Note that replacing $?y$ in a URI template with $\mu(?y)$ may result in a IRI, and not a URI, since some of the characters in $\mu(?y)$ need not be ASCII. To stress this, we use the term IRI for any instantiation of the variables in a URI template.

Semantics. The semantics of a SERVICE-to-API pattern is defined in terms of the *instantiation* of an URI template 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 there is some $?x \in \text{var}(U)$ such that $\mu(?x)$ is not defined, we define $\mu(U)$ as an invalid IRI that will result in an error when invoked.

Thus, every mapping produces an IRI, which we then use to execute an HTTP request to the API in the body of the IRI. Formally, given an URI template 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. We adopt the convention that HTTP requests that do not give back a JSON document result in an error, that is, $\text{call}(U, \mu) = \texttt{error}$ whenever the request using U does not result in a valid JSON document.

For instance, if μ is a mapping, such that $\mu(?y) = \texttt{Ben\_Nevis}$, and $U = \texttt{<http://weather.api/request?q=\{?y\}>}$ is a URI template, then $\mu(U) = \texttt{<http://weather.api/request?q=Ben\_Nevis>}$. 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, or it is an error.

To define the evaluation we need some more notation. First, 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. Next, given a JSON document J, a navigation expression N, and a variable $?x$, we define the set $M_{?x \mapsto J[N]}$ that contains the single mapping $?x \mapsto J[N]$, when $J[N]$ is a basic JSON value (integer, string, or boolean), and is equal to the empty set $\emptyset$ otherwise. We also assume that $M_{?x \mapsto J[N]} = \emptyset$ when J is not a valid JSON document, or $J = \texttt{error}$.

The semantics of a SERVICE-to-API pattern P of the form (1) is then:

$$[\![P]\!]_G = \bigcup_{\mu \in [\![P_1]\!]_G} \{\mu\} \bowtie M_{?x_1 \mapsto \text{call}(U,\mu)[N_1]} \bowtie \cdots \bowtie M_{?x_m \mapsto \text{call}(U,\mu)[N_m]}$$

Therefore, a mapping in $[\![P]\!]_G$ is obtained by extending a mapping $\mu \in [\![P_1]\!]_G$ by binding each $?x_i$ to $\text{call}(U, \mu)[N_i]$. In the case that $\text{call}(U, \mu) = \texttt{error}$ (e.g. when $\mu(?x)$ is not defined for some $?x \in \text{var}(U)$), or that $\text{call}(U, \mu)[N_i]$ is not a basic JSON value, the mapping μ will not be extended to the variables $?x_i$, and will not be part of $[\![P]\!]_G$. This is consistent with the default behaviour of SPARQL `SERVICE` [30] which makes the entire query fail if the `SERVICE` call results in an error. In the case that we want to implement the `SILENT` option for `SERVICE` which makes the latter behave as an `OPTIONAL` (see [30]), we would need to change the $\emptyset$ in the definition of $M_{?x \mapsto J[N]}$ to the empty mapping $\mu_\emptyset$, since this mapping can be joined with any other mapping.

For example, if $P_1 = \{\texttt{?x wdt:P131 wd:Q22 . ?x rdfs:label ?y}\}$ is a pattern, and $U = \texttt{<http://weather.api/request?q=\{?y\}>}$ a URI template. Let

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

be the pattern we are evaluating over some RDF graph G, and assume that $[\![P_1]\!]_G$ contains the following mappings.

	$?x$	$?y$
μ_1	`wd:London`	`London`
μ_2	`wd: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 $\text{call}(U, \mu)$, and store the value $\text{call}(U, \mu)$[`"temperature"`] into the variable `?t`, in case that the obtained JSON value is a string, a number, or a boolean value, and discard μ otherwise. For example, if we assume that the calls are as follows,

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

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

	$?x$	$?y$	$?t$
μ_1	`wd:London`	`London`	22

Since $\text{call}(U, \mu_2)$ returns an error, the mapping μ_2 can not be extended, so it will not form a part of the output. In the case that the "`SILENT` semantic" is triggered, we would actually output μ_2 where $?t$ would not be bound.

3.2 A Basic Implementation

We propose a way to implement the overloaded `SERVICE` operation on top of any existing SPARQL engine without the need to modify its inner workings. To do so, we partition each query using this operator into smaller pieces, and evaluate these using the original engine whenever possible. More precisely, whenever we find a pattern of the form:

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

in our query, we process it over a local database G using the following algorithm:

1. Compute $[\![P_1]\!]_G$ (recursively if P_1 contains a SERVICE-to-API pattern).
2. Define $M = \emptyset$.
3. For each $\mu \in [\![P_1]\!]_G$ do:
   - Execute $\text{call}(U, \mu)$; if an error is returned, start step 3 with the next μ;
   - For $1 \leq i \leq m$, compute $M_i = M_{?x_i \mapsto \text{call}(U,\mu)[N_i]}$; if there was an error, start step 3 with the next μ;
   - Let $M = M \cup (\{\mu\} \bowtie M_1 \bowtie \cdots \bowtie M_m)$.
4. Finally, to compute $[\![P]\!]_G$, serialize the set of mappings M using the `VALUES` operator, as in [7], 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 serialized 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.

With this implementation, the natural question is whether this basic implementation can be optimized. As we have mentioned, the bottleneck in our case is API calls, so if we want to evaluate queries as efficiently as possible we need to do the least amount of API calls as possible. There are a number of optimisations we can immediately implement in our basic implementation that will reduce the number of calls, and we try some of them in Section 5. However, next we consider a rather different question, for a broad subclass of patterns: Can we reformulate query plans to make sure we are making as few calls as possible?

4 A Worst-case optimal algorithm

Our goal is to evaluate SERVICE-to-API queries as efficiently as possible, which implies minimising the number of API calls we issue when evaluating queries. This takes us to the following question: what is the minimal amount of API calls that need to be issued to answer a given query? Ideally, we would like to issue a number of calls that is linear in the size of the output of the query: for each tuple in the output we issue only those calls that are directly relevant for returning that particular tuple. But in general this is not possible. Consider the pattern

$$(?x_0, p, ?x_1)\ \ \mathsf{AND}\ \ldots\ \ \mathsf{AND}\ \ (?x_{m-1}, p, ?x_m)\ \ \mathsf{SERVICE}\ \ U\{(N)\ \ \mathsf{AS}\ ?y\},$$

where U uses variables $?x_1, \ldots, ?x_m$. Then the number of calls we would need to issue could be of order $|G|^m$ (e.g. when all triples in G are of the form (a, p, b)), but depending on the API data the output of this query may even be empty!

What we can do is aim to be optimal in the worst case, making sure that we do not make more calls than the number we would need in the worst case over all graphs and APIs of a given size. We can devise an algorithm that realises this bound if we focus on the smaller class of SPARQL queries made just from AND, FILTER and SERVICE-to-API operators, which we denote as *conjunctive patterns*. This is the federated analogue of conjunctive queries, which amount to roughly two thirds of the queries issued on the most popular endpoints on the Web, according to [12].

As we shall see, bounding the number of API calls for this fragment is intimately related to bounding the number of tuples in the output of a relational query, a subject that has received considerable attention in the past few years in the database community (see e.g. [8,18,27]). To illustrate this, let P be a conjunctive SERVICE-to-API pattern of the form:

$$(\ldots(((((P_1\ S_1)\ \ \mathsf{AND}\ \ P_2)\ \ S_2)\ \ \mathsf{AND}\ \ P_3)\ \ S_3)\ \ \ldots\ \ P_n)\ \ S_n,$$

where each P_i is a SPARQL pattern (not using SERVICE) and each S_i is of the form $\mathsf{SERVICE}\ U\ \{(N_1, N_2, \ldots, N_m)\ \ \mathsf{AS}\ (?x_1, ?x_2, \ldots, ?x_m)\}$, with U, m, and each of the N_js and $?x_j$s possibly different for different S_i.

We can now cast the problem of processing the query P over an RDF graph G as the problem of answering a join query over a relational database as follows. First, we simulate each SPARQL pattern P_i as a relation R_i with attributes corresponding to the variables of P_i (i.e. we project out the constants from each pattern in P_i since they do not contribute to an output). Second, we view an API U in each `SERVICE` call S_i described above as a relation T_i with *access methods* (see e.g. [10,13]), that has *output attributes* $?x_1, \ldots, ?x_m$ and input attributes $\text{var}(U)$. Intuitively, an access method prevents arbitrary access to a relation; the only way to retrieve the tuples of a relation T with input attributes $A_1, \ldots, A_k$ is to provide appropriate values for $A_1, \ldots, A_k$, after which we are given all the tuples in T that match these input values.

It is now easy to see that answering P over G is the same as answering the following relational query[2]:

$$Q_P = R_1 \bowtie T_1 \bowtie R_2 \bowtie T_2 \bowtie \ldots \bowtie R_n \bowtie T_n, \tag{2}$$

over the relational instance that has the result of $[\![P_i]\!]_G$ stored in R_i, and the API data in T_i, for $i = 1 \ldots n$. Queries of the form (2) are known as *join queries*. Generally, join queries that do not use access methods are one of a few classes of queries for which we know tight bounds for the size of their outputs [8]. In what follows, we show that this bound can be extended even for queries that use access methods, such as (2), thus allowing us to solve the problem of evaluating `SERVICE`-to-API patterns.

We say that every action of matching values for the input attributes of one of the T_i's is a *call* operation, and we are able to offer a tight bound on the number of calls needed to answer a join query with access method such as Q_P. The main result we show in this section is the following. Take any feasible[3]join 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 [8] 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[4]. 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*

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

To show this proposition we provide an algorithm that, given a query Q_P obtained from a conjunctive SERVICE-to-API pattern as described above, con-

[2] We abuse the notation and denote relational joins using the same symbol that we use for mappings; the two operators are always distinguished by the context.

[3] Some queries with access methods may not be answerable, e.g. a query with only input attributes. We call queries that can be answered feasible (see Subsection 4.1).

[4] The bound is obtained by solving a specific linear program that depends on the query and the arity of the relations in D. We do not have space to formally state this result or this program, but refer to [19] for an excellent summary

structs a query plan for Q_P that is guaranteed to make a number of calls satisfying the bound. We also show that this bound is tight: one can construct a family of patterns and instances of growing size where one actually needs that amount of API calls. We then show that the query plan for Q_P can be used to construct a query plan for P that is worst-case optimal.

4.1 API calls as relational access methods

Our shift into the relational setting is to facilitate the presentation when merging all the different data paradigms involved in the evaluation of SERVICE-to-API patterns. In the following we assume familiarity with relational databases and schemas, and relational algebra. For a quick reference see [4].

We denote access methods with the same symbol as relations, but making explicit which of the attributes are input attributes, and which are output attributes. For example, an access method for a relation $R(A, B, C)$ with attributes A, B and C and where A and C are input attributes is denoted by $R(A^i, B^o, C^i)$ (letter i is a shorthand for *input* and o for *output*).

Access methods impose a restriction on the way queries are to be evaluated, as there are queries that cannot be evaluated at all. To formalise the intuition that access methods impose a restriction on the way queries are to be evaluated, we say that a relation R_i in a join query $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$ is *covered* if all of its input attributes appear as an output in any of the relations $R_1, \ldots, R_{i-1}$. Then such a join query is said to be *feasible* when all its relations are covered. For example, consider a schema with with relations $R(A^i, B^o)$, $S(A^o, B^o)$ and $T(B^i, C^o)$. Then $S \bowtie R \bowtie T$ is feasible: the input for R is an output of S and likewise for T. But $R \bowtie T$ is not feasible, as we do not have a source for the input of R. Naturally, a join query can only be answered if it is equivalent to a feasible query, so without loss of generality we focus on feasible queries. This is also enough for our purposes, as all queries we produce out of conjunctive SERVICE-to-API patterns are feasible.

We adopt the convention that, for a relation T with input attributes $A_1, \ldots, A_k$ and a set R of tuples having all attributes $A_1, \ldots, A_k$, the number of calls required to answer $R \bowtie T$ corresponds to the size of $\pi_{A_1,\ldots,A_k}(R)$. Intuitively, this means that we answer $R \bowtie T$ by selecting all different inputs coming from the tuples of R, and issue one call for each of these inputs.

We can then analyse the number of calls for the naive left-deep join plan for $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$, which corresponds to setting $\phi_1 = R_1$ and iteratively computing $\phi_{i+1} = \phi_i \bowtie R_{i+1}$ until we obtain ϕ_n, which corresponds to the answers of Q. How many calls do we issue? In the worst case where all except R_1 are relations representing APIs, we would need to issue a number of calls corresponding to the sum of the tuples in R_1, $R_1 \bowtie R_2$, and so on until $R_1 \bowtie \ldots \bowtie R_{n-1}$.

It turns out that we can provide a much better bound for the number of calls required, as well as an algorithm fulfilling this bound. In the following section we show an algorithm that produces a reformulation of Q whose left-deep plan issues a number of calls that agrees with the bound in Theorem 1. We also show that the algorithm is as good as it gets for arbitrary feasible join queries.

4.2 The algorithm

Let $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$ be a feasible join query under some access methods, and let $A_1, \ldots, A_n$ be an enumeration of all attributes involved in Q, in order of their appearance. Without loss of generality, we assume that there is exactly one access method per relation in Q (if not one can construct two different relations, the worst case analysis does not change). We use $\mathit{Input}(R)$ to denote the set of all input attributes of the access method for R.

Our algorithm is inspired by the optimal plan exhibited in [8,19] for conjunctive queries without access methods. Starting from Q, we construct a query $Q^* = \Delta_n$, where the sequence $\Delta_1, \ldots, \Delta_n$ is defined as:

1. For Δ_1, let $S_1^1, \ldots, S_{k_1}^1$ be all relations in $\{R_1, \ldots, R_n\}$ whose set $\mathit{Input}(S_\ell^1)$ of input attributes is contained in $\{A_1\}$ (including relations with only output attributes). Then
$$\Delta_1 = \pi_{A_1}(S_1^1) \bowtie \ldots \bowtie \pi_{A_1}(S_{k_1}^1).$$
2. For Δ_i, let again $S_1^i, \ldots, S_{k_i}^i$ be all relations such that $\mathit{Input}(S_\ell^i)$ is contained in $\{A_1, \ldots, A_i\}$. Then
$$\Delta_i = \Delta_{i-1} \bowtie \pi_{A_1,\ldots,A_i}(S_1^i) \bowtie \ldots \bowtie \pi_{A_1,\ldots,A_i}(S_{k_i}^i).$$

The feasibility of Q^* follows from the fact that Q is feasible, so every relation with inputs $A_1, \ldots, A_i$ appears after all these attributes are outputs of previous relations, and we order attributes in the order of appearance. According to the construction above, we can write Q^* as a natural join $Q^* = E_1 \bowtie \ldots \bowtie E_r$ of expressions E_i which are join free. We then evaluate Q^* using a left-deep join plan: we start with the leftmost expression $\phi_1 = E_1 = \pi_{A_1}(R)$ in Q^*, where R is some relation, and then keep computing $\phi_t = \phi_{t-1} \bowtie E_t$, for $t = 2, \ldots, r$. The relation ϕ_r contains our output. Part of our plan involves caching the results of all relations R with $\mathit{Input}(R) \neq \emptyset$ the first time they are requested, and before we compute any projection over them. This only imposes a memory requirement that is at most as big as what we would need with the basic implementation.

Analysis. Recall that for a query Q and instance D, $M_{Q,D}$ is the maximum size of the projection of any relation in Q over a single attribute, and $2^{\rho^*(Q,D)}$ is the AGM bound of the query. Theorem 1 now follows from the following proposition.

Proposition 1. *Let Q be a feasible join query over a schema with access methods and D a relational instance of this schema. Let Q^* be the query constructed from Q by the algorithm above. Then the number of calls required to evaluate Q^* over D using a left-deep plan is in*
$$O(M_{Q,D} \times 2^{\rho^*(Q,D)}).$$

Let m be the number of relations in Q and n the total number of attributes. If we are considering combined complexity (i.e. Q is part of the input), the bound above raises to $O(m \times M_{Q,D} \times 2^{\rho^*(Q,D)})$ for the algorithm that does caching. Likewise, the number of calls is in $O(n \times m \times M_{Q,D} \times 2^{\rho^*(Q,D)})$ if we rule out the possibility of caching.

For the worst case optimality we show queries realising the upper bound.

Proposition 2. *There is a schema $\mathcal{S}$, a query Q and a family of instances $(D_n)_{n\geq 1}$ such that: (i) The maximum size of the projection of a relation in D over one attribute is n, (ii) The AGM bound is n^2, and (iii) Any algorithm evaluating Q must make at least n^3 calls to a relation with access methods.*

SERVICE-to-API patterns. To create optimal plans for SERVICE-to-API patterns, we need to show that (1) our translation from patterns to relational queries is sound and creates feasible queries, and (2) how to devise an optimal plan for the SERVICE pattern when given a plan for the relational query.

For (1), let P be a SERVICE-to-API pattern and Q_P the constructed join query, and consider an RDF graph G. Then the instance $I_{P,G}$ in which Q_P should be evaluated is defined next, and the correctness lemma follows.

- Each relation R_i in $I_{P,G}$ with attributes $?x_1, \ldots, ?x_m$ contains the set of tuples $\{(\mu(?x_1), \ldots, \mu(?x_n)) \mid \mu \in [\![P]\!]_G\}$.
- Each relation T_i in $I_{P,G}$ with input attributes $?z_1, \ldots, ?z_k$ and output attributes $?y_1, \ldots, ?y_p$ contains the set of tuples $\{(\mu(?z_1), \ldots, \mu(?z_k), \mu(?y_1), \ldots, \mu(?y_p)) \mid \mu \in [\![P]\!]_G\}$.

Lemma 1. *Let P be a conjunctive SERVICE-to-API pattern using variables $\{?x_1, \ldots, ?x_\ell\}$. A tuple $(a_1, \ldots, a_\ell)$ is in the evaluation of Q_P over $I_{P,G}$ if and only if there is a mapping $\mu \in [\![P]\!]_G$ such that $(a_1, \ldots, a_\ell) = (\mu(?x_1), \ldots, \mu(?x_\ell))$.*

While not obvious, this lemma also shows that the query is feasible, as long as P is not trivially unanswerable (i.e., as long as there is a graph G for which $[\![P]\!]_G$ is nonempty). Note that for finding the worst-case optimal plan for Q_P we do not need to construct the instance $I_{P,G}$, as this would amount to pre-computing the answer $[\![P]\!]_G$. Next, for (2): we show how the optimal plan for Q_P gives us an optimal plan for P:

Proposition 3. *Let P be SERVICE-to-API pattern, G an RDF graph and Q_P, $I_{P,G}$ the corresponding relational query with access methods and instance as constructed above. Then any optimal query plan Q^* for Q_P over an instance $I_{P,G}$ can be transformed (in polynomial time) into a query plan for P that evaluates P over G using the same amount of API calls as the evaluation of Q^*.*

Proof. The plan for P mimics step-by-step the plan for Q_P. That is, assume that $Q^* = E_1 \bowtie \cdots \bowtie E_r$ is the reformulation of Q_P from Section 4.2. Starting with $\phi_1 = E_1$, we iteratively compute the the set $[\![\phi_i]\!]_G = [\![\phi_{i-1}]\!]_G \bowtie E_i$ of mappings for each $i = 2 \cdots r$. This is done in the following way. Whenever $E_i = \pi_{?x_1,\ldots,?x_p} R_i$, we evaluate the query $\mathsf{SELECT}\ ?x_1, \ldots, ?x_p\ \mathsf{WHERE}\ P_i$ over G. On the other hand, if E_i is a relation using T_j for the first time, we call the API (because Q_P is feasible we will have all the needed input parameters), cache all the API results and then only retrieve the attributes that are not projected out in E_i. All subsequent appearances of T_j are evaluated directly on the cached JSON file. (If we are not using caching, then we need to call the API for each E_k, where $k > i$, that uses T_j.) Since the query ϕ_r is equivalent to Q^*, it is also equivalent to Q_P. Thus the output of this query plan correctly computes $[\![P]\!]_G$ by Lemma 1. The number of calls is worst-case optimal by Propositions 1 and 2.

5 Experiments

The goal of this section is to give empirical evidence that the worst-case optimal algorithm of Section 4 is indeed a superior evaluation strategy for executing queries that use API calls. We also constructed several real world use cases, for space reasons we defer them to the appendix [1] and the online demo [2].

Experimental setup. To construct a benchmark for SERVICE-to-API patterns we reformulate the queries from the Berlin benchmark [11] by designating certain patterns in a query to act as an API call. We then run a battery of tests that simulate real-world APIs by sampling from the distributions of the response times presented in the introduction. The experiments where run on a 64-bit Windows 10 machine, with 8 GB of RAM, and Intel Core i5 7400 @ 3.0 GHz processor. Experiments were repeated five times, reporting the average value.
Adapting the Berlin benchmark to include API calls. The Berlin benchmark dataset [11] is inspired by an e-commerce use case. It has products that are offered by vendors and are reviewed by users. Each one of those entities has properties related with them (such as labels, prices, etc.). The size of the dataset is specified by the user. To test our implementation we created a database of 5000 products consisting of 1959874 triples.

The benchmark itself is composed of 12 queries. Our adaptation consists of exposing the data of five recurrent patterns we find in the benchmark queries as APIs that return JSON documents. For instance, {`?x bsbm:productPropertyNumericZ ?y`} is one such pattern, where `Z` is a number between 1 and 5. This pattern is used to return the value of some numeric property of a product with the label `?x`, so we created a (local) API route `api/numeric-properties/{label}`, that will give us all the values of numeric properties of an object. For instance, if a product with the IRI `bsbm:Product1` has a label `"Product 1"`, and its numerics properties are `PropertyNumeric1` = 3, `PropertyNumeric2` = 10, the request `api/numeric-properties/Product_1` returns the JSON: `{ "p1": 3, "p2": 10}`. The other API routes we implemented are similar (details can be found in [1]).

Next, we transform the original benchmark queries by replacing each pattern used when creating the APIs by a `SERVICE` call to the corresponding API. For instance, in the case of the "numeric properties API" described above, we replace each pattern of the form: {`?product bsbm:productPropertyNumericX ?valueX`}, by the following API call:

```
SERVICE <api/numeric-properties/{label}>{ (["pX"]) AS (?valueX) }
```

We did a similar transformation for each pattern including entities served by our APIs. We ran all the queries of the Berlin Benchmark except Q6, Q9, and Q11, because they were too short to include API calls in their patterns. Also, we change the `OPTIONAL` operator in each query by `AND`, because the two are the same in terms of worst case optimal analysis.

Implementation. Our implementation of `SERVICE`-to-API patterns is done on top of Jena TBD 3.4.0 using Java 8 update 144. We differentiate three evaluation

algorithms for `SERVICE`-to-API patterns: (1) *Vanilla*, the base implementation described in Section 3; (2) *Without duplicates*, the base algorithm that uses caching to avoid doing the same API call more than once; and (3) *WCO*, the worst-case optimal algorithm of Section 4.

Results. The number of API calls done for each of the three versions of our algorithm are shown in Table 2. 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. Full details in [1].

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

6 Conclusion

In this paper we propose a way to allow SPARQL queries to connect to HTTP APIs returning JSON. We describe the syntax and the semantics of this extension, show how it can be implemented on top of existing SPARQL engines, provide a worst-case optimal algorithm for processing these queries, and demonstrate the usefulness of this algorithm both formally and in practice.

In future work, we plan to support formats other than JSON, and explore how to support it in public endpoints. It would be also interesting to test how issuing API calls in parallel affects the running times of diferent algorithms. 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.

References

1. Online Appendix. http://67.205.159.121/query/appendix/.
2. Online demo of `SERVICE`-to-API. http://67.205.159.121/query/#/.
3. `SERVICE`-to-API implementation. http://67.205.159.121/query/code/.
4. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
5. C. B. Aranda, M. Arenas, and Ó. Corcho. Semantics and optimization of the SPARQL 1.1 federation extension. In *ESWC 2011*, pages 1–15, 2011.
6. C. B. Aranda, M. Arenas, Ó. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.*, 18(1):1–17, 2013.

7. C. B. Aranda, A. Polleres, and J. Umbrich. Strategies for executing federated queries in SPARQL1.1. In *ISWC 2014*, pages 390–405, 2014.
8. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.
9. 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.
10. M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6):690–701, 2015.
11. C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
12. A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *CoRR*, abs/1708.00363, 2017.
13. A. Calì and D. Martinenghi. Querying data under access limitations. In *ICDE 2008*, pages 50–59, 2008.
14. 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.
15. P. Fafalios and Y. Tzitzikas. SPARQL-LD: a SPARQL extension for fetching and querying linked data. In *ISWC Demos*, 2015.
16. P. Fafalios, T. Yannakis, and Y. Tzitzikas. Querying the web of data with SPARQL-LD. In *TPDL 2016*, pages 175–187, 2016.
17. F. Galiegue and K. Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.
18. G. Gottlob, S. T. Lee, G. Valiant, and P. Valiant. Size and treewidth bounds for conjunctive queries. *J. ACM*, 59(3):16:1–16:35, 2012.
19. 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.
20. S. Harris and A. Seaborne. SPARQL 1.1 query language. *W3C*, 2013.
21. IETF. URI Template. https://tools.ietf.org/html/rfc6570, 2012.
22. M. Junemann, J. L. Reutter, A. Soto, and D. Vrgoč. Incorporating API data into SPARQL query answers. In *ISWC 2016 Posters & Demos*, 2016.
23. N. Kobayashi, M. Ishii, S. Takahashi, Y. Mochizuki, A. Matsushima, and T. Toyoda. Semantic-json. *Nucleic Acids Research*, 39:533–540, 2011.
24. G. Montoya, M. Vidal, and M. Acosta. A heuristic-based approach for planning federated SPARQL queries. In *COLD 2012*, 2012.
25. G. Montoya, M. Vidal, Ó. Corcho, E. Ruckhaus, and C. B. Aranda. Benchmarking federated SPARQL query engines: Are existing testbeds enough? In *ISWC 2012*, pages 313–324, 2012.
26. 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.
27. H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *PODS 2012*, pages 37–48, 2012.
28. J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.
29. F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *WWW 2016*, pages 263–273, 2016.
30. E. Prud'hommeaux and C. Buil-Aranda. SPARQL 1.1 Federated Query. *W3C Recommendation*, 21, 2013.
31. L. Rietveld and R. Hoekstra. YASGUI: not just another SPARQL client. In *ESWC 2013*, pages 78–86, 2013.