

Querying JSON APIs with SPARQL

Matthieu Mosser^a, Fernando Pieressa^a, Juan L. Reutter^{a,b}, Adrián Soto^{c,b,d,*},
Domagoj Vrgoč^{a,b}

^a*Pontificia Universidad Católica de Chile.*

^b*Millennium Institute for Foundational Research on Data.*

^c*Faculty of Engineering and Sciences Universidad Adolfo Ibáñez.*

^d*Data Observatory Foundation.*

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 this information into SPARQL query answers, obtaining a language that combines data from the “traditional” Web to the Semantic Web. Our proposal is based on an extension of the SERVICE operator with the ability to connect to JSON APIs. With the aim of evaluating 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 note that the analysis of this algorithm is studied in terms of an algorithm for evaluating relational queries with access methods with the minimal number of access queries, which is of independent interest. We show the superiority of the worst-case optimal approach in a series of experiments that take existing SPARQL benchmarks, and augment them with the ability to connect to JSON APIs in order to obtain additional information.

Keywords: SPARQL, Semantic Web, SERVICE, JSON, API

*Corresponding author

Email addresses: `mmosser@uc.cl` (Matthieu Mosser), `fapieressa@uc.cl` (Fernando Pieressa), `jreutter@ing.puc.cl` (Juan L. Reutter), `adrian.soto@uai.cl` (Adrián Soto), `dvrhoc@ing.puc.cl` (Domagoj Vrgoč)

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 [21], 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 [38].

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 one of the most popular data formats used 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

30 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, 35 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:

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

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

```
45 {"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 50 records where the weather condition is "clear sky". One can do this manually, but this quickly becomes 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:

```
55 SELECT ?x ?l WHERE {
    ?x wdt:instanceOf wd:mountain .
    ?x wdt:locatedIn wd:Scotland .
    ?x rdfs:label ?l .
```

```

SERVICE <http://weather.api/request?q={?1}>{(["description"]) AS (?d)}
60 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 `?1` 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. \square

70 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 75 API calls. This is mostly because HTTP requests are slower than 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. Based on the API documentation we created ten different calls for each API, and ran each call five times, recording the average value. The results presented in Table 1 80 show us the minimum, the maximum, and the average time over our calls for each API. The least amount of time needed was 0.3 seconds, which is already quite substantial when processing a query that makes a huge amount of API calls, and it can range up to more than a second.

Hence, to evaluate these queries efficiently we need to understand how to 85 produce a query plan for them that minimizes the number of calls to the API. Apart from formally defining the syntax and the semantics of the extended `SERVICE` operator, finding algorithms that minimize the number of API calls is the main question studied in this paper.

	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 (in seconds) of popular Web APIs based on ten typical calls they support.

Contributions. Our main contributions can be summarized as follows:

- 90 - *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 [36], making it easy to incorporate this extension into the language standard.
- *Evaluation algorithms.* We propose several evaluation strategies for the extended **SERVICE** operator, starting with a basic algorithm that mimics the evaluation of basic graph patterns. To minimize the number of API calls, we implement a series of optimizations based on algorithms that evaluate queries under running time comparable to the the AGM bound [6, 35] for estimating the number of intermediate results in relational joins.
- 100 - *Optimality guarantees.* We formally prove that the algorithm based on the AGM bound is indeed worst case optimal for evaluating a large fragment of SPARQL patterns that use remote **SERVICE** calls. More precisely, we show that we do not make more calls than we would need to make in the worst case over all possible RDF graphs, and API datasets of a given size. Our optimality proof 105 also establishes tight bounds for answering join queries over relations with access methods [13, 9], which is of independent interest.
- *Implementation.* We provide a fully functional implementation of the extended **SERVICE** operator within the Apache Jena framework, with the support of several different evaluation algorithms. The source code of our implementation can 110 be found at [1].
- *Experimental evaluation.* We test different evaluation strategies for the ex-

tended **SERVICE** operator over a range of queries and data sources. In particular, we extend the Berlin query benchmark [10], and the recent WikiData benchmark [23] with the ability to pull data from APIs. Our results provide empirical
115 evidence for the superiority of the worst case optimal algorithm to diminish the number of API calls. Additionally, we show how different algorithms scale as the size of the base dataset and the API load increase.

Remark. Several aspects of this work have been presented at different conferences in the past. In particular, a simple demo showcasing the utility of the
120 extended **SERVICE** operator has been presented in [25]. An initial study of worst case optimal algorithms in this context was conducted in [32], and a short paper on the topic was presented in [33]. In this paper we extend these works along the following lines:

- We provide a more general extension for API **SERVICE** calls, that can
125 process request that return JSON arrays instead of JSON values. The semantics has therefore been rewritten to account for this case, and we also include a set of examples to guide the intuition in this case. Moreover, all results that were previously presented are now extended to account for this case.
- We provide a full proof of the worst case optimal bound for the number
130 of API calls, which was only announced in [32]. Given the setting we consider, our proof also shows how to adapt the worst case optimal approach to evaluate joins under access restrictions, which is a problem that, to the best of our knowledge, has not been tackled previously. Additionally, the
135 paper includes all of the remaining proofs for the results announced in [32].
- We update the experimental section with different sizes of datasets using the Berlin benchmark [10], so that we understand how the API calls scale with bigger databases. We also we extend this to the recent Wikidata
140 query benchmark [23].

- Finally, we provide a detailed comparison of our approach with the previous work in Section 2.

Organization. We place our work in the context of the existing literature in Section 2. We recall standard notions in Section 3. The formal definition and
145 examples of the extended **SERVICE** are given in Section 4, where we also give a basic strategy for evaluating this operator. The worst-case optimal algorithm for evaluating queries that use **SERVICE** is presented in Section 6. Experimental evaluation is given in Section 7. We conclude in Section 8.

2. Related work

150 Pulling data from different sources is a fundamental feature of Semantic Web technologies, and there is an abundance of the literature on the subject. Most notably, SPARQL 1.1. [21] introduces the ability to federate queries using the **SERVICE** keyword, which permits connecting to different endpoints in order to bring in data from a diverse set of sources distributed over the Web. Fundamen-
155 tal studies in the area [3, 4, 5, 31, 30] lay down the formal foundations for the **SERVICE** operator, and also identify main challenges for query evaluation in the distributed setting. 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.

160 When it comes to optimizing the evaluation of federated queries, there are several approaches that aim to minimize the amount of **SERVICE** calls and data transfers between different endpoints. The main objective here is to generate an optimal query plan without having all the data available locally. Most approaches to this problem combine Selinger-style query optimization and a clever
165 use of heuristics and/or data statistics for each endpoint to produce efficient execution plans [14, 29, 40]. This is contrast to our approach, since we are agnostic to any statistics on the queried APIs, and simply aim at minimizing the number of API calls in a worst-case optimal manner. The work in [43] starts from the same assumption regarding the lack of information about the data in the end-

170 points, but query planning is done based on heuristics only, without providing
any formal guarantees of their efficiency. Overall, what differentiates us most
from this work is that in contrast to the classical relational join optimization
based on statistics and heuristics, we build our execution plans to be optimal
in the worst case.

175 Additionally, one can view the evaluation of federated queries in a wider
setting. For instance, [22], already proposes a way to execute SPARQL queries
on the linked Web, before federation became part of the language standard, and
identifies network latency as a potential stumbling stone to query execution.
Similarly, [8], casts these ideas in a more general way, while [42] changes the
180 execution paradigm by shifting the processing load from the server executing
the query to the client side.

Complementary to federation in SPARQL, there is also a lot of work on
bringing data from different sources into SPARQL, in a similar way as we do
in this paper. Some of these approaches are based on the idea of building RDF
185 wrappers for other formats [26, 34, 39, 15], which 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 [16, 17, 7, 41] that incorporate API data directly into SPARQL, but do not
provide query execution guarantees.

190 Going in the other direction, there is also a lot of work on bringing SPARQL
query results into Web APIs. For instance, [27] defines a transformation lan-
guage for turning SPARQL results into JSON, while [28] allows building generic
APIs based on RDF and Linked Data.

Finally, it is worth noting that worst case optimal evaluation of SPARQL
195 queries was considered before in [23]. The main difference with this work is the
setting: while [23] assumes all the data to be available locally, we rely on APIs
to provide part of the data. Thus, the bottleneck changes, and whereas [23] is
concerned with minimizing the total running time of queries, we only focus on
API calls (a measure that does not exist in the setting of [23] because everything
200 is assumed to be local). This also requires us to consider evaluating joins under

access restrictions (e.g. in Example 1, the value of the variable ?1 has to be available before making the API call), which is a problem that, to the best of our knowledge, has not been solved in a worst-case optimal manner previously. Of course, it would be interesting to study how to merge our approach, providing
205 an algorithm that is both worst-case optimal in terms of database operations and in terms of API calls. We leave this promising direction as future work.

3. 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*
210 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. SPARQL is the standard query language for RDF [21]. Let \mathbf{V} be
215 a set of variables, where \mathbf{V} is disjoint from \mathbf{T} . A tuple $t \in (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{V})$ is called a triple pattern. Blank nodes in triple patterns can be considered as query variables for our purposes. A sequence $\{t_1\}.\{t_2\}.\dots.\{t_n\}$, where each t_i is a triple pattern, is called a *basic graph pattern*. We denote by $\text{var}(t)$ and $\text{var}(P)$ the set of variables found in a triple pattern t and basic graph
220 pattern P , respectively.

The semantics of graph patterns is defined in terms of *mappings* [21]; that is, partial functions from the set of variables \mathbf{V} to IRIs \mathbf{I} . The *domain* $\text{dom}(\mu)$ 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
225 $\text{dom}(\mu_1) \cap \text{dom}(\mu_2)$. 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 $\text{dom}(\mu_2) \setminus \text{dom}(\mu_1)$. Note that if two mappings μ_1 and μ_2 have no variables in common they are always compatible, and that the empty mapping μ_\emptyset is compatible with any

other mapping. For sets M_1 and M_2 of mappings we also define their join as
 230 $M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 : \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}$.

We can now define the evaluation of a triple pattern and a basic graph pattern over an RDF graph G :

$$\begin{aligned} \llbracket t \rrbracket_G &= \{\mu \mid \text{var}(t) = \text{dom}(\mu) \text{ and } \mu(t) \in G\} \\ \llbracket \{t_1\} \cdots \{t_n\} \rrbracket_G &= \llbracket t_1 \rrbracket_G \bowtie \dots \bowtie \llbracket t_n \rrbracket_G \end{aligned}$$

SPARQL queries can be constructed by combining a wide range of query operators, such as union, optional, filters, aggregates, property paths, etc. We assume the reader is familiar with the syntax and semantics of SPARQL 1.1. If P is a SPARQL pattern, we also denote the *evaluation* of a graph pattern P
 235 over G as $\llbracket P \rrbracket_G$. We do recall the syntax and semantics of the **SERVICE** operator, because it will be heavily used in this paper.

If $a \in \mathbf{I}$, then by $ep(a)$ we denote the graph G that is served by the SPARQL endpoint reached by a . In case that a is not a valid endpoint (i.e. an RDF dataset), we define $ep(a) = \emptyset$. Similarly, if $?x$ is a variable, and $\mu(?x)$ is not defined, then we consider that $ep(\mu(?x)) = \emptyset$. The set $\llbracket P \rrbracket_G$ for a pattern $P = P_1 . \mathbf{SERVICE} \ a \ \{P_2\}$ is defined as

$$\begin{aligned} \llbracket P \rrbracket_G &= \{ \mu \mid \mu = \mu_1 \cup \mu_2, \\ &\text{where } \mu_1 \in \llbracket P_1 \rrbracket_G, \mu_2 \in \llbracket P_2 \rrbracket_{ep(\mu(a))}, \text{ and } \mu_1 \sim \mu_2 \} \end{aligned}$$

As pointed out by Aranda et al. [4, 3], the semantics of service patterns $P = P_1 . \mathbf{SERVICE} \ a \ \{P_2\}$ is not clearly defined when a is a variable and not all mappings in $\llbracket P_1 \rrbracket_G$ bind variable a : in this case we do not know which graphs
 240 need to be queried because they depend on the result of the pattern itself. The way we deal with the situation is by using P_1 as a “guard” in case that a is a variable, only calling P_2 in mappings in $\llbracket P_1 \rrbracket_G$ where $\mu(a)$ is bound. Having service patterns with such a guard is equivalent to the strongly bound safety condition of Aranda et al., since one can simply push all “non-SERVICE” patterns
 245 into P_1 .

Finally, if Q is a query of the form $Q = \mathbf{SELECT} \ W \ \mathbf{WHERE} \ \{ P \}$, with P a

graph pattern, and W a set of variables, then the result of evaluating Q over the graph G , denoted $\llbracket Q \rrbracket_G$, is defined as $\{\mu_W \mid \mu \in \llbracket P \rrbracket_G\}$, where μ_W denotes the mapping obtained by restricting the domain of μ to the variables from W .

250 **JSON.** The JSON format [12] 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, \dots, v_n are JSON values and s_1, \dots, s_n are pairwise distinct string values, then $o =$
 255 $\{s_1 : v_1, \dots, s_n : v_n\}$ 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, \dots, v_n are JSON values then $a = [v_1, \dots, v_n]$ is a JSON value called an *array*. In this case v_1, \dots, v_n are called the *elements of a*. Numeric values, strings and the boolean values **true**, and **false** are called *basic JSON values*.

260 **JSON navigation instructions.** To navigate through JSON documents we use *JSON navigation instructions*. For an object J , the navigation instruction $J[\text{"key"}]$ returns the value of a pair in J whose key is the string "key", and for an array J , the navigation instruction $J[n]$, for a natural number n , returns the n -th element of J . These instructions can be stacked to retrieve values of nested
 265 JSON documents; e.g. $J[\text{"key1"}][7]$, will first fetch the value of the key "key1" (if J is an object), and then, assuming that this value is an array, return the seventh element of the array. If the JSON does not have the corresponding key or array element, the navigation expression returns an error.

Size bounds for join queries. We recall some of the results by Atserias et. al. [6] regarding size bounds as presented in [20]. Consider a join query $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$ with attributes A_1, \dots, A_n , and a database D where the size of each R_i is N_i . The idea is to obtain a bound for the output of Q in terms of the cardinality of each relation. Suppose that we have relations R_{i_1}, \dots, R_{i_k} that contains all the attributes appearing in Q . Then, a bound for the size of the query could be $|Q| \leq \prod_{j=1}^k N_{i_j}$, since the other relations involved only act as filters for the final output. We can get an even better bound by

selecting the smallest set of relations with this property, which corresponds to what is known as the edge cover of the query Q . Moreover, the seminal result by Atserias et. al. shows an optimal bound by considering the following linear program:

$$\begin{aligned}
 & \text{minimize} && \sum_i x_i \log N_i \\
 & \text{where} && \sum_{i : A_j \text{ is an attribute of } R_i} x_i \geq 1 \quad \text{for } j = 1, \dots, n \\
 & && x_i \geq 0 \quad \text{for } i = 1, \dots, m
 \end{aligned} \tag{1}$$

We denote by $\rho^*(Q, D)$ the optimal value of $\sum_i x_i \log N_i$. The AGM bound [6] establishes that $|Q(D)|$ is always bounded by $2^{\rho^*(Q, D)}$, and that this bound is tight, i.e. there are infinitely many queries and families of instances where the bound is realized. We thus refer to $2^{\rho^*(Q, D)}$ as the *AGM bound* of Q over D .

4. Enabling SPARQL to make JSON calls

In this section we define the syntax and the semantics of the overloaded `SERVICE` operator that allows SPARQL to connect to JSON APIs and incorporate API information into its query answers. We begin by describing how JSON APIs function, followed by the syntax and semantics of the overloaded `SERVICE` operation in SPARQL. We finish by illustrating the utility of this extension using a set of real world examples.

4.1. JSON APIs, requests and navigating JSON documents, URI templates

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 these implementation details vary with APIs and systems, so here we just focus on the problem of evaluating these queries.

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. We always assume that the general structure of the JSON response is
 290 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. [37, 18]). This is a common assumption when one works with Web APIs.

The last ingredient we need are URI templates. We use them as a simple
 295 way to define placeholders that will be filled at the time an HTTP requests is made.

Definition 1 (URI templates [24]). *A URI Template is a URI in which the query part may contain substrings of the form $\{?x\}$, for $?x$ in \mathbf{V} . For example, the following are URI templates:*

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

*The idea behind these templates is that variable elements inside brackets are replaced by concrete values at the time the request is made. In what follows, we will refer to the variables in such substrings of a URI template U as the
 305 variables of U , and denote them with $\text{var}(U)$.*

4.2. Syntax and semantics of the extended SERVICE operator

Our proposal is based on overloading the SERVICE operator to allow for SERVICE-to-API patterns, which we define next.

Definition 2 (SERVICE-to-API pattern). *Let P_1 be a SPARQL pattern, U a URI template using only variables that appear in P_1 , $?x_1, \dots, ?x_m$ a sequence of pairwise distinct variables that do not appear in P_1 , and N_1, \dots, N_m a sequence of JSON navigation instructions. Then the following is a SPARQL pattern, that we call a SERVICE-to-API pattern:*

$$P_1 . \text{SERVICE } U \{(N_1, N_2, \dots, N_m) \text{ AS } (?x_1, ?x_2, \dots, ?x_m)\} \quad (2)$$

The idea is that we now allow users writing queries to be able to access
 310 information given by APIs they know of. The intuition behind the evaluation
 of this operator over a graph G is the following. For each mapping μ in the
 evaluation $\llbracket P_1 \rrbracket_G$ we instantiate every variable $?y$ in the URI template U with
 the value $\mu(?y)$, thus obtaining an IRI which is a valid API call¹. We call the
 API with this instantiated IRI, obtaining a JSON document, say J . We then
 315 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 $?x_1, \dots, ?x_m$, which have been assigned values according to J and
 N_1, N_2, \dots, N_m .

320 Notice that in (2) 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.

Example 2. Consider the SPARQL basic graph pattern P_1 given by $P_1 =$
 $\{?x \text{ wdt:P131 wd:Q22 . ?x rdfs:label ?y}\}$ and the URI template U given
 325 by $U = \langle \text{http://weather.api/request?q}=\{?y\} \rangle$. Then the following is a
 SERVICE-to-API pattern.

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

As we explained, the intention of this pattern is to issue a call to U instantiated
 with all values $?y$ in the evaluation of P_1 , and then assign to $?t$ those values
 330 found under the key "**temperature**" of the JSON document that is returned by
 the API.

Semantics. The semantics of a SERVICE-to-API pattern is defined in terms
 of the *instantiation* of a URI template U with respect to a mapping μ (denoted

¹Note 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.

$\mu(U)$), which is simply the IRI that results by replacing each construct $\{?x\}$ in
 335 U with $\mu(?x)$, or an invalid IRI in case some $\mu(?x)$ is not defined. 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, we have the following definition.

Definition 3 (Calling APIs with templates and mappings). *Let U be a
 URI template and μ a mapping. The instantiation of U with respect to μ ,
 340 denoted as $\mu(U)$, corresponds to*

- *An arbitrary invalid IRI, if there is some $?x \in \text{var}(U)$ such that $\mu(?x)$ is not defined, or*
- *The IRI obtained by replacing each construct $\{?x\}$ in U with $\mu(?x)$.*

*Then, the call to U with respect to μ , denoted as $\text{call}(U, \mu)$, is the result of
 345 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**.*

Note that we adopt the convention that HTTP requests that do not give back
 350 a JSON document result in an error, that is, $\text{call}(U, \mu) = \text{error}$ whenever the
 request using U does not result in a valid JSON document.

Example 3. Consider the mapping μ , such that $\mu(?y) = \text{Ben_Nevis}$, and the
 template $U = \langle \text{http://weather.api/request?q}=\{?y\} \rangle$. Then we have that
 $\mu(U) = \langle \text{http://weather.api/request?q}=\text{Ben_Nevis} \rangle$. When this request is
 355 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 of SERVICE-to-API patterns, we need some more
 notation. First, if $?x$ is a variable and $t \in \mathbf{T}$ a term, 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 $M_{?x \mapsto J[N]}$ of mappings as follows:

$$M_{?x \mapsto J} = \begin{cases} \{?x \mapsto J\} & , \text{ if } J \text{ is a basic JSON value} \\ \bigcup_{1 \leq i \leq k} \{?x \mapsto J_i\} & , \text{ if } J = [J_1, \dots, J_k], \text{ and all } J_i \\ & \text{are basic JSON values} \\ \emptyset & , \text{ otherwise.} \end{cases}$$

The idea is that $M_{?x \mapsto J[N]}$ contains the single mapping $?x \mapsto J[N]$, when $J[N]$ is a basic JSON value (integer, string, or boolean), or, if $J[N]$ is the array $[J_1, \dots, J_k]$ of basic JSON values, a set of k mappings each of which maps $?x$ to an element J_i . As per the definition above, we also assume that $M_{?x \mapsto J[N]} = \emptyset$ when J is not a valid JSON document, or $J = \mathbf{error}$.

We can finally state the semantics of SERVICE-to-API patterns.

Definition 4 (Semantics of SERVICE-to-API). *Let P be a SERVICE-to-API pattern as specified in Definition 2, with the form*

$$P = P_1 . \mathbf{SERVICE} \ U \ \{(N_1, N_2, \dots, N_m) \ \mathbf{AS} \ (?x_1, ?x_2, \dots, ?x_m)\}.$$

The semantics $\llbracket P \rrbracket_G$ is then defined as

$$\llbracket P \rrbracket_G = \{\mu \bowtie \mu_1 \bowtie \dots \bowtie \mu_m \mid \mu \in \llbracket P_1 \rrbracket_G, \mu_i \in M_{?x_i \mapsto \text{call}(U, \mu)[N_i]}\}.$$

Therefore, a mapping in $\llbracket P \rrbracket_G$ is obtained by extending a mapping $\mu \in \llbracket P_1 \rrbracket_G$ by binding each $?x_i$ to the value in the JSON value $\text{call}(U, \mu)[N_i]$ (or one of the values therein, if said JSON document is an array).

In the case that $\text{call}(U, \mu) = \mathbf{error}$ (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 $\llbracket P \rrbracket_G$. Moreover, if $\mu \in \llbracket P_1 \rrbracket_G$ is not compatible with a mapping in $M_{?x_i \mapsto \text{call}(U, \mu)[N_i]}$ (because, for example, μ assigns a different value to $?x_i$), then this will also not be part of $\llbracket P \rrbracket_G$. This behaviour is inline with the default behaviour of SPARQL SERVICE [38] 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 [38]), we would need to change the
 375 \emptyset in the definition of $M_{?x \rightarrow J[N]}$ to the empty mapping μ_\emptyset , since this mapping can be joined with any other mapping.

Let us now illustrate this definition by means of some examples. First, we illustrate the basics of our definitions in a context where the call is given by a single variable and the values returned are basic JSON values.

380 **Example 4 (Example continued).** Consider the SPARQL basic graph pattern P_1 given by $P_1 = \{?x \text{ wd:P131 wd:Q22} . ?x \text{ rdfs:label } ?y\}$ and the URI template U given by $U = \langle \text{http://weather.api/request?q}=\{?y\} \rangle$. Let

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

be the pattern we are evaluating over some RDF graph G , and assume that
 385 $\llbracket P_1 \rrbracket_G$ contains the following mappings.

	$?x$	$?y$
μ_1	<code>wd:London</code>	<code>London</code>
μ_2	<code>wd:Berlin</code>	<code>Berlin</code>

The evaluation of P over G is then obtained by extending mappings in $\llbracket P_1 \rrbracket_G$ using U . That is, we iterate over $\mu \in \llbracket P_1 \rrbracket_G$ one by one, execute the call $\text{call}(U, \mu)$, and store the value $\text{call}(U, \mu)[\text{"temperature"}]$ into the variable $?t$,
 390 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,

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

then $M_{?t \rightarrow \text{call}(\mu_1, U)[\text{"temperature"}]}$ contains the mapping that assigns the number 22 to the variable $?t$ and $M_{?t \rightarrow \text{call}(\mu_2, U)[\text{"temperature"}]}$ is empty. Thus, the
 395 evaluation $\llbracket P \rrbracket_G$ will contain the following mapping

	$?x$	$?y$	$?t$
μ_1	<code>wd:London</code>	<code>London</code>	22

As we previously remarked, the call $\text{call}(U, \mu_2)$ 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.

Let us now present a more involved example, in which we have a few variables and the APIs return arrays.

Example 5 (Example 4 continued). As in Example 4, we continue using pattern $P_1 = \{?x \text{ wdt:P131 } \text{wd:Q22} . ?x \text{ rdfs:label } ?y\}$. This time, however, we use a different URI template U' , given by

$$U' = \langle \text{http://weather.api/request?q}=\{?y\}\&\text{type=extended} \rangle,$$

which returns more extended weather information, including a 3-day forecast.

Once again, we have a graph G , and we assume that $\llbracket P_1 \rrbracket_G$ contains mappings.

	$?x$	$?y$
μ_1	<code>wd:London</code>	<code>London</code>
μ_2	<code>wd:Berlin</code>	<code>Berlin</code>

This time, however, the documents that the API returns are different. In particular, $\text{call}(\mu_1, U')$ is now the following JSON document:

```
{
  "temperature": {
    "current": 22,
    "forecast": [18,17,18]
  },
  "description": "sunny at times, possible showers in the evening"
}
```

In order to get both the current temperature and the 3-day forecast, we use

the SERVICE-to-API P' , given by

```
P' = P1 . SERVICE U' { ([ "temperature"/"current" ],
  [ "temperature"/"forecast"/0 ], [ "temperature"/"forecast"/1 ],
  [ "temperature"/"forecast"/2 ] ) AS (?c, ?f0, ?f1, ?f2) }
```

The idea is to retrieve the current temperature into variable $?c$, and the three days of forecast in variables $?f0$, $?f1$ and $?f2$. Thus, in this case, to compute the answer we have the following sets of mappings:

- The single mapping in $M_{?c \rightarrow \text{call}(\mu_1, U')["temperature"/"current"]}$ assigns 22 to variable $?c$,
- The single mapping in $M_{?c \rightarrow \text{call}(\mu_1, U')["temperature"/"forecast"/0]}$ assigns 18 to variable $?f0$,
- The single mapping in $M_{?c \rightarrow \text{call}(\mu_1, U')["temperature"/"forecast"/1]}$ assigns 17 to variable $?f1$, and
- The single mapping in $M_{?c \rightarrow \text{call}(\mu_1, U')["temperature"/"forecast"/2]}$ assigns 18 to variable $?f2$,

Once again, if we assume that $\text{call}(\mu_2, U')$ returns **error**, then the evaluation $\llbracket P' \rrbracket_G$ of P' over G is the mapping

	$?x$	$?y$	$?c$	$?f0$	$?f1$	$?f2$
μ	wd:London	London	22	18	17	18

Using JSON arrays directly gives us another option to retrieve this information, which would result in an answer with more mappings, but less variables, which would be specially useful in times we do not know the number of elements in the array we retrieve. For example, if we now use the pattern

```
P'' = P1 . SERVICE U' { ([ "temperature"/"current" ],
  [ "temperature"/"forecast" ] ) AS (?c, ?f) }
```

430 Now the navigation instruction ["temperature"/"forectast"] gives the array [18, 17, 18] when evaluated in $\text{call}(\mu_1, U')$. Then, the set $M_{?c \rightarrow \text{call}(\mu_1, U')["temperature"/"current"]}$ still contains only the mapping that assigns 22 to variable $?c$, but now the set $M_{?c \rightarrow \text{call}(\mu_1, U')["temperature"/"forecast"]}$ contains three mappings:

	$?f$
σ_0	18
σ_1	17
σ_2	18

435 Thus, in this case the semantics of P'' is similar to that of P' , but where the values of f_0 , f_1 and f_2 are distributed into the value of f in three different mappings. That is, the evaluation $\llbracket P'' \rrbracket_G$ of P'' over G is the set of mapping given by:

	$?x$	$?y$	$?c$	$?f$
μ_1^1	wd:London	London	22	18
μ_1^2	wd:London	London	22	17
μ_1^3	wd:London	London	22	18

440 5. A Basic Implementation

In this section 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. 445 The idea here is to obtain all the information needed to execute the API calls, and then do all the calls at once.

Before we describe our algorithms, we have a few important issues to address. First, we remark that all this machinery is designed to work under the assumption that API name parameters are known, as well as the schema of the 450 responses of each API call, and this is why our operators are designed so that users input all of this information at the time of processing queries. Furthermore, we also assume that API results are correct: our goal in this paper is to

produce optimal evaluation of patterns calling APIs, but we do not deal with other important problems such as incomplete information, correctness of web results, or entity-linking.

5.1. Basic processing algorithm

The basic implementation delegates the computation of all SPARQL components to the system. When evaluating the answers of a query over a graph G , whenever the system encounters a pattern of the form

$$P \equiv P_1 . \text{SERVICE } U\{(N_1, N_2, \dots, N_m) \text{ AS } (?x_1, ?x_2, \dots, ?x_m)\}$$

that needs to be processed, our implementation proceeds as follows. First, we compute the answers $\llbracket P_1 \rrbracket_G$ by calling again our implementation. Then, we use Algorithm 1 to compute the answers $\llbracket P \rrbracket_G$ of the full pattern, which can be invoked once we know the answers of the basic pattern accompanying the SERVICE call. Finally, we serialize the set of mappings M using the VALUES operator, as in [5], to allow it to be used by the next graph pattern inside the WHERE clause in which it appears, thus enabling us to delegate once again the computation of the answer to a SPARQL system.

Regarding the final step, the obtained mappings need to be serialized as strings 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 it can be optimized. As we have mentioned in the introduction, the bottleneck in our case is API calls, so if we want to evaluate queries efficiently, we need to do the least amount of API calls as possible. There are a number of optimisations we can immediately apply to our basic implementation that will reduce the number of calls, and we discuss them next. Afterwards, in Section 6, 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?

Algorithm 1: Naive evaluation of a SERVICE-to-API pattern

Input : A graph G , a pattern $P \equiv$

$P_1 . \text{SERVICE } U\{(N_1, N_2, \dots, N_m) \text{ AS } (?x_1, ?x_2, \dots, ?x_m)\},$
set $\llbracket P_1 \rrbracket_G$ of mappings.

Output: Answer $\llbracket P \rrbracket_G$

Initialize M, M_1, \dots, M_m as \emptyset ;

for each $\mu \in \llbracket P_1 \rrbracket_G$ **do**

 Execute $\text{call}(U, \mu)$;

if $\text{call}(U, \mu)$ returns error **then**

 | *continue* to the beginning of the loop ;

end

for each $1 \leq i \leq m$ **do**

 | compute $M_i = M_{?x_i \mapsto \text{call}(U, \mu)[N_i]}$;

 | **if** $M_i = \emptyset$ **then**

 | *break* the loop ;

 | **end**

end

 Let $M = M \cup (\{\mu\} \bowtie M_1 \bowtie \dots \bowtie M_m)$

end

Return M

5.2. Immediate optimisations

Here we describe two simple approaches for reducing the number of API calls: avoiding duplicate calls, and caching. These simple optimisations will also be compared to the more general algorithm for minimising the number of calls that we propose in Section 6.

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 {  
485   ?x ex:label1 ?z .  
      ?x ex:label2 ?n .
```

```
SERVICE <http://api.org/{?n}> { (["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 suboptimal. To eliminate this behaviour we proceed as follows. We first obtain all *distinct* values $?n$ that can be bound 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

$$P_1 . \text{SERVICE } U \{ (N_1, N_2, \dots, N_m) \text{ AS } (?x_1, ?x_2, \dots, ?x_m) \}, \quad (3)$$

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

```
{ SELECT var(P1) WHERE {P1} } AND
{ (SELECT DISTINCT ?y1, ..., ?yn WHERE P1)
SERVICE U {(N1, N2, ..., Nm) AS (?x1, ?x2, ..., ?xm) } }
```

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.

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.

Our implementation has support for caching API calls while processing the same SERVICE-to-API pattern, which should remove the need for duplicate requests when processing a single SERVICE-to-API query. This caching is brute-force: 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. Of course, for complex queries there is a memory issue where all cached files may not fit into working memory. If this is the case we just keep the files in disk until we finish processing the

API. We remark that this is a usual problem when caching data, and one can tackle this problem using the same techniques as general caching in databases (for example, by leasing the control of the cache to the operating system), and for this reason we do not discuss this issue any further.

510 Finally, one could also think of a caching strategy that is based on optimising calls across the evaluation of multiple SERVICE-to-API patterns, and over a wider timespan. The problem with this optimisation is that it is only correct when the result of the same API call does not change over time, which is something we cannot control.

515 6. 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
 520 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 a pattern of the following form:

$$\{?x_0 \text{ p } ?x_1\} \dots \{?x_{m-1} \text{ p } ?x_m\} . \text{SERVICE } U \{(N) \text{ AS } ?y\},$$

525 where U is a call that uses all variables $?x_1, \dots, ?x_m$. 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
 530 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 concatenating basic graph patterns and SERVICE-to-API operators, which we denote as *conjunctive patterns*. This is the federated analogue of conjunctive

535 queries, which amount to roughly two thirds of the queries issued on the most popular endpoints on the Web, according to [11].

We tackle this problem with a novel framework that reduces API calls to the problem of 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. [6, 19, 35]), but over an extended relational 540 model that includes *access methods* for each relation. We then devise an algorithm that can evaluate queries over relational databases with access methods, a result which is of independent interest. We start this section with a high-level explanation of our technique, and then proceed to design the algorithm and 545 establish the promised worst-case optimality.

6.1. Overview of our framework

We begin by showing how to cast the problem of processing a SERVICE-to-API pattern as a problem of answering a join over a relational database with access methods. More precisely, given a graph G and a pattern P , we construct 550 a relational instance $I_{P,G}$ and a relational query Q_P such that the evaluation of Q_P over $I_{P,G}$ corresponds precisely to the evaluation $\llbracket P \rrbracket_G$. Note that we can do this because P is a conjunctive pattern, and therefore every mapping in $\llbracket P \rrbracket_G$ bounds the same variables.

So how does Q_P and $I_{P,G}$ look like? We can easily translate basic graph 555 patterns in P as relational tables, simply by storing all the results of the pattern as a table. For example, given a basic graph pattern $P' = \{?x \ p_1 \ ?y\}.\{?x \ p_2 \ ?z\}$, we can store it as a relation R' on attributes $?x, ?y, ?z$, such that $(a, b, c) \in R'$ if and only if both (a, p_1, b) and (a, p_2, c) are triples in G . But APIs require a more careful treatment. We model them as relations with *access methods* (see 560 e.g. [9, 13]). Intuitively, the idea is that these relations have a set of *ouput attributes* that can only be accessed once we know the *input attributes*. Then, the variables used to construct an API call are represented as input attributes, and the information we extract from the call is modelled as output attributes.

In order to state our bounds, denote by $M_{Q,D}$ the maximum size of the pro-
 565 jection of any relation appearing in a relational query Q over a single attribute
 in the database D . The main result we show in this section is the following

Proposition 5. *Any conjunctive service to API P can be evaluated over a
 graph G using a number of calls in*

$$O(M_{Q_P, I_P, G} \times 2^{\rho^*(Q_P, I_P, G)}).$$

The proof of this proposition comes from a novel result establishing tight
 bounds for answering join queries over relations with access methods. The re-
 mainder of this section is devoted to the proof of these bounds, and then relating
 570 it to Proposition 5. We begin by formally defining relations with access meth-
 ods, and then show how one can minimize the number of calls when evaluating
 queries over such relations.

6.2. Relational setup

In the following we assume familiarity with relational databases and schemas,
 575 and relational algebra [2]. We begin with the notion access methods.

Definition 6 (relational schemas with access methods). *An access method
 for a relation R partition its attributes R as input attributes and output at-
 tributes. We denote access methods with the same symbol as the relations
 they specify, but making explicit which of the attributes are input attributes,
 580 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. For example, consider a
 585 schema with relations $R(A^i, B^o)$, $S(A^o, B^o)$ and $T(B^i, C^o)$. Then $S \bowtie R \bowtie T$
 can always be evaluated², since the input for R is an output of S and likewise

²We 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.

for T . However, $R \bowtie T$ can not be evaluated, as we do not have a source for the input of R . Let us formalise these notions.

Definition 7 (Feasible join queries). *A join query over a relational schema \mathcal{S} with access methods is a query of the form $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$, where*
 590 *each R_i is a (not necessarily distinct) access method for a relation in \mathcal{S} , and each join is a natural join.*

An access method for relation R_i in Q is covered if all of its input attributes appear as an output in any of the relations R_1, \dots, R_{i-1} . If all its access methods
 595 *are covered, then Q is said to be feasible.*

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. It turns out that this is also enough for our purposes, as queries we produce out of conjunctive SERVICE-to-API patterns are always feasible.

600 The task of evaluating a query over access methods closely resembles querying APIs, as in both cases we are using known information to query data sources we don't know. Furthermore, an API itself can be understood as a relation with access methods, in which the information requested by the API are their inputs, and the information returned is the output. However, in order to analyse the
 605 cost of answering a relational query with access methods we need to fix the communication cost of accessing a particular set of tuples that is under an access method with an input. As in APIs, we adopt the convention of one call per each different set of inputs.

Definition 8 (Calls in access methods). *For a relation T with input attributes A_1, \dots, A_k and a set R of tuples having all attributes A_1, \dots, A_k , the*
 610 *number of calls required to answer $R \bowtie T$ corresponds to the size of $\pi_{A_1, \dots, A_k}(R)$. 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
 615 for $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$, which corresponds to setting $\phi_1 = R_1$ and

iteratively computing $\phi_{i+1} = \phi_i \bowtie R_{i+1}$ until we obtain ϕ_m , 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
620 $R_1 \bowtie \dots \bowtie R_{n-1}$.

Example 6. Consider a schema $R(A^o, B^o), S(B^i, C^o), T(C^i, A^i)$ and query $Q = R \bowtie S \bowtie T$. The left deep plan for Q first computes $R \bowtie S$, which needs a number of calls equivalent to the number of tuples in $\pi_B(R)$. Then, we use the result of $R \bowtie S$ to compute $R \bowtie S \bowtie T$, which requires a number of calls
625 equivalent to the number of tuples in $\pi_{A,C}(R \bowtie S)$, which is quadratic on the size of R and S .

In the following section we show a tighter bound for the number of calls required, as well as an algorithm fulfilling this bound.

6.3. Optimal algorithm for join queries with access methods

630 Without loss of generality, in this section 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).

Our algorithm is inspired by the optimal plan exhibited in [6, 20] for conjunctive queries without access methods. Starting from a join query Q with
635 attributes A_1, \dots, A_n , we construct a sequence of queries $\Delta_1, \dots, \Delta_n$, where Δ_n is equivalent to Q , and where the evaluation of each such query Δ_i overestimates the evaluation of the query $\pi_{A_1, \dots, A_i}(Q)$. The construction is depicted as pseudocode in Algorithm 2. The idea is to order every attribute participating in the query, and for an (ordered) increasing amount of attributes \mathcal{A} , we obtain
640 all potential values of these attributes that may be part of the final answer. The key component is that these set of potential values is the best over-estimation of the API calls we need to issue when processing relations whose input values are contained in the set \mathcal{A} of attributes.

Algorithm 2: Rewrite

Input : A feasible join query $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$ under access methods

Output: A sequence of queries $\Delta_1, \dots, \Delta_n$

Let A_1, \dots, A_n be an enumeration of all attributes involved in Q , in order of their appearance ;

Let $\text{inputs}(R)$ denote the set of all input attributes of the access method for R ;

let $S_1^1, \dots, S_{k_1}^1$ be all relations in $\{R_1, \dots, R_n\}$ whose set $\text{inputs}(S_\ell^1)$ of input attributes is contained in $\{A_1\}$ (including relations with only output attributes);

$\Delta_1 \leftarrow \pi_{A_1}(S_1^1) \bowtie \dots \bowtie \pi_{A_1}(S_{k_1}^1)$;

for $i = 2$ **to** n **do**

 Let $S_1^i, \dots, S_{k_i}^i$ be all relations such that $\text{inputs}(S_\ell^i)$ is contained in $\{A_1, \dots, A_i\}$;

$\Delta_i \leftarrow \Delta_{i-1} \bowtie \pi_{A_1, \dots, A_i}(S_1^i) \bowtie \dots \bowtie \pi_{A_1, \dots, A_i}(S_{k_i}^i)$;

end

Return $\Delta_1, \dots, \Delta_n$

Next, as each of these queries is feasible, we can evaluate them one-by-one
645 over our database. As the analysis shows, this is in fact an optimal way of
reducing the number of calls issued by the query (as long as we assume that we
do not call two times the same access method with the same parameter).

Analysis. Let us first remark that the feasibility of Q^* follows from the fact
that Q is feasible, so every relation with inputs A_1, \dots, A_i appears after all
650 these attributes are outputs of previous relations, and we order attributes in
the order of appearance. Furthermore, we remark that we assume that we store
the results of all relations R with $\text{Input}(R) \neq \emptyset$ in memory the first time they
are requested, and before we compute any projection over them, so that we do
not have to issue another call to R using the same inputs. This only imposes

Algorithm 3: Evaluating a join query with access methods

Input : A feasible join query $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$ under access methods, a database D

Output: Evaluation of Q over D with minimal calls

Let A_1, \dots, A_n be an enumeration of all attributes involved in Q , in order of their appearance ;

$\Delta_1, \dots, \Delta_n \leftarrow Rewrite(Q)$;

for $i = 1$ **to** n **do**

 Let $S_1^i, \dots, S_{k_i}^i$ be all relations such that $inputs(S_\ell^i)$ is contained in

$\{A_1, \dots, A_i\}$, and recall that

$\Delta_i \leftarrow \Delta_{i-1} \bowtie \pi_{A_1, \dots, A_i}(S_1^i) \bowtie \dots \bowtie \pi_{A_1, \dots, A_i}(S_{k_i}^i)$.

 Then, Δ_i is feasible.

 Evaluate Δ_i over database D using the evaluation of Δ_{i-1}

end

Return the evaluation of Δ_n over database D .

655 a memory requirement that is at most as big as what we would need with the basic implementation described in the previous section.

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. We then have the following result.

Proposition 9. *Let Q be a feasible join query over a schema with access methods and D a relational instance of this schema. Let Δ_n be the query constructed from Q by the algorithm above. Then the number of calls required to evaluate Δ_n over D using a left-deep plan is in*

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

PROOF. Let us assume that $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$, all over attributes A_1, \dots, A_n , and let $\Delta_1, \dots, \Delta_n$ as explained above. As we mentioned, observe that each Δ_i is also feasible. Now any API call we do for a relation R_j with

input attributes A_1, \dots, A_i in our left deep plan is of the form

$$(\Delta_{i-1} \bowtie \pi_{A_1, \dots, A_i}(R_1) \bowtie \dots \bowtie \pi_{A_1, \dots, A_i}(R_{j-1})) \bowtie \pi_{A_1, \dots, A_i}(R_j).$$

The size of $\Delta_{i-1} \bowtie \pi_{A_1, \dots, A_i}(R_i)$ is bounded by the size of $\Delta_{i-1} \times \pi_{A_i}(R_i)$, where \times denotes the Cartesian product of relations, as Δ_{i-1} already contains a subexpression joining with $\pi_{A_1, \dots, A_{i-1}}(R_i)$. Hence, the number of tuples in the output of the expression

$$(\Delta_{i-1} \bowtie \pi_{A_1, \dots, A_i}(R_1) \bowtie \dots \bowtie \pi_{A_1, \dots, A_i}(R_{j-1}))$$

660 is bounded by $M_{Q,D} \cdot |\Delta_{i-1}(D)|$, as we only add to Δ_{i-1} the values $\pi_{A_i}(R)$ of the relation R with the greatest projection over A_i , the others act only as semijoins that filter the result. So to evaluate an appearance of a relation R_j in some Δ_i we need to pose at most $M_{Q,D} \cdot |\Delta_{i-1}|$ calls. Since by the AGM bound we have that $|\Delta_{i-1}| \leq 2^{\rho^*(Q,D)}$, we can therefore evaluate the appearance of
 665 R_j in Δ_i using at most $M_{Q,D} \cdot 2^{\rho^*(Q,D)}$ calls. Therefore, the total number of calls is bounded by $m \cdot M_{Q,D} \cdot 2^{\rho^*(Q,D)}$ (as we are caching results), which is in $O(M_{Q,D} \times 2^{\rho^*(Q,D)})$ when we assume the query to be fixed (that is, when we consider the data complexity).

Let m be the number of relations in Q and n the total number of attributes.
 670 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 storing results of the calls in memory during the process of evaluating the queries.

675 **Optimality.** Our last result establishes the optimality of our solution, as there are queries that must be evaluated using at least the number of calls demanded by Proposition 9.

Proposition 10. *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*

680 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.

PROOF. Consider the schema with access methods with relations $R(A^o, B^o, C^o)$, $S(B^i, C^i, D^o)$, $U(B^i, E^o)$ and $T(A^i, D^i, E^i)$, and an instance D where R contains tuples $\{(i, 1, 1) \mid 1 \leq i \leq n\}$, S contains tuples $\{(1, 1, i) \mid 1 \leq i \leq n\}$, U contains tuples $\{(1, i) \mid 1 \leq i \leq n\}$ and T contains n arbitrary tuples. Now let $Q = R \bowtie S \bowtie U \bowtie T$. The maximum size of the projection of a relation in D over one attribute is n , and it can be checked that the bound $2^{\rho^*(Q,D)}$ corresponds to n^2 , by solving the appropriate linear program. Furthermore, since T has inputs which are only in the union of all R , S and U , any algorithm looking to answer Q must issue one call for each tuple in $\pi_{A,D,E}(R \bowtie S \bowtie U)$ (if not, one can create an interpretation for T where this algorithm does not answer the query correctly). We obtain that n^3 calls are needed, as this is the size of the output of $R \bowtie S \bowtie U$ over D .

6.4. Algorithm for SERVICE-to-API patterns

695 Let us now come back to SERVICE-to-API patterns. The way we provide an algorithm is by (1) translating SERVICE-to-API patterns into join queries with access methods, and then (2) constructing 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 . Next we describe the construction of the relational query Q_P and explain how to construct the instance $I_{P,G}$ in which Q_P should be evaluated.

Definition 11 (Relational counterparts of SERVICE-to-API patterns).

Let G be a graph and P be a conjunctive SERVICE-to-API pattern of the form:

705
$$\{ \{ \{ \{ \{ \{ P_1 \cdot S_1 \} \cdot P_2 \} \cdot S_2 \} \cdot P_3 \} \cdot S_3 \} \dots P_n \} \cdot S_n,$$

where each P_i is a basic graph pattern (not using SERVICE) and each S_i is a different SERVICE-to-API pattern.

Query Q_P is then defined as

$$Q_P = R_1 \bowtie T_1 \bowtie R_2 \bowtie T_2 \bowtie \dots \bowtie R_n \bowtie T_n, \quad (4)$$

where each R_i is a relation whose attributes corresponds to the variables in P_i , and each T_i is a relation with access methods constructed from S_i as follows. Suppose T_i is of the form

$$\text{SERVICE } U \{(N_1, N_2, \dots, N_m) \text{ AS } (?x_1, ?x_2, \dots, ?x_m)\}.$$

Then the output attributes in T_i are $?x_1, \dots, ?x_m$, and the input attributes of T_i are $\text{var}(U)$, the variables mentioned in the URI template U .

710 Next, the relational instance $I_{P,G}$ is defined as follows.

- Each relation R_i in $I_{P,G}$ with attributes $?x_1, \dots, ?x_m$ contains the set of tuples

$$\{(\mu(?x_1), \dots, \mu(?x_n)) \mid \mu \in \llbracket P \rrbracket_G\}.$$

- Each relation T_i in $I_{P,G}$ with input attributes $?z_1, \dots, ?z_k$ and output attributes $?y_1, \dots, ?y_p$ contains the set of tuples

$$\{(\mu(?z_1), \dots, \mu(?z_k), \mu(?y_1), \dots, \mu(?y_p)) \mid \mu \in \llbracket P \rrbracket_G\}.$$

With this in mind, we can now show the soundness of using a relational translation to answer conjunctive patterns:

Lemma 12. *Let P be a conjunctive SERVICE-to-API pattern using variables $\{?x_1, \dots, ?x_\ell\}$. A tuple (a_1, \dots, a_ℓ) is in the evaluation of Q_P over $I_{P,G}$ if and*
 715 *only if there is a mapping $\mu \in \llbracket P \rrbracket_G$ such that $(a_1, \dots, a_\ell) = (\mu(?x_1), \dots, \mu(?x_\ell))$.*

PROOF. Let us assume P and G are of the form described in Definition 11, where in particular the set of variables mentioned in P is $\{?x_1, \dots, ?x_\ell\}$, $Q_P = R_1 \bowtie T_1 \bowtie R_2 \bowtie T_2 \bowtie \dots \bowtie R_n$ and $I_{P,G}$ is constructed as explained above.

For the **if** direction, consider a tuple (a_1, \dots, a_ℓ) in the evaluation of Q_P over $I_{P,G}$. By definition, there must be mappings μ_1, \dots, μ_n and $\mu'_1, \dots, \mu'_{n-1}$ such that (i) all these mappings are compatible and their value over $\{?x_1, \dots, ?x_\ell\}$

must coincide whenever at least two of them have one of these variables in the domain, (ii) μ_i is in $\llbracket P_i \rrbracket_G$ and (iii) Letting

$$\mu = \bigcup_{1 \leq j \leq n} \mu_j \cup \bigcup_{1 \leq k \leq n-1} \mu'_k$$

we have that for each SERVICE call

$$S_j = \text{SERVICE } U(N_1, N_2, \dots, N_m) \text{ AS } (?z_1, ?z_2, \dots, ?z_m),$$

we have that $\text{call}(U, \mu)[N_k]$ is either $\mu'_j(?z_k)$ or is an array that contains $\mu'_j(?z_k)$.

720 By the semantics of SPARQL and (conjunctive) SERVICE-to-API patterns we get that $\mu \in \llbracket P \rrbracket_G$, and that $(a_1, \dots, a_\ell) = (\mu(?x_1), \dots, \mu(?x_\ell))$.

For the **only if** direction, consider such a mapping μ in $\llbracket P \rrbracket_G$ and let $(a_1, \dots, a_\ell) = (\mu(?x_1), \dots, \mu(?x_\ell))$. It is then easy to see that each of the relations are populated with the appropriate subsets of (a_1, \dots, a_ℓ) , so that this
725 tuple must be in the evaluation of Q_P over I_P .

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 $\llbracket P \rrbracket_G$. The lemma above is just to state that the translation is correct.

Next we show how to construct an optimal plan for P from the optimal plan
730 we know how to construct from Q_P . Together with Lemma 12, this completes the proof of Proposition 5.

Proposition 13. *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 in-
735 stance $I_{P,G}$ 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 Δ_n is the reformulation of Q_P from Section 6.3, and that $\Delta_n = E_1 \bowtie \dots \bowtie$

⁷⁴⁰ E_r , with each E_r a join-free expression. . Starting with $\phi_1 = E_1$, we iteratively
 compute the the set $\llbracket \phi_i \rrbracket_G = \llbracket \phi_{i-1} \rrbracket_G \bowtie E_i$ of mappings for each $i = 2 \dots r$.
 This is done in the following way. Whenever $E_i = \pi_{?x_1, \dots, ?x_p} R_i$, we evaluate the
 query `SELECT ?x1, . . . , ?xp WHERE Pi` over G . On the other hand, if E_i is a
⁷⁴⁵ 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 JSON file we store while the
 evaluation continues. (If we are not using this mild form of caching, then we
 need to call the API for each E_k , where $k > i$, that uses T_j .) Since the query ϕ_r
⁷⁵⁰ is equivalent to Q^* , it is also equivalent to Q_P . Thus the output of this query
 plan correctly computes $\llbracket P \rrbracket_G$ by Lemma 12. The number of calls is worst-case
 optimal by Proposition 9 and Proposition 10.

7. Experiments

⁷⁵⁵ The goal of this section is to give empirical evidence that the worst-case
 optimal algorithm of Section 6 does indeed minimize the number of API calls.
 For this, we run two sets of experiments: the first one is based on the Berlin
 SPARQL Benchmark (v3.1) [10], and the second one is the recently proposed
 Wikidata Benchmark [23] that tests the performance of evaluating BGPs in
 SPARQL engines. These benchmarks were designed for standard SPARQL,
⁷⁶⁰ but we adapted the queries to make them useful for testing SERVICE-to-API
 patterns. In the corresponding subsections, we specify how we simulate API
 calls and responses. All 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. The
 source code of our implementation, together with the scripts used to simulate
⁷⁶⁵ APIs can be found at [1].

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 four evaluation
 algorithms for SERVICE-to-API patterns: (1) **Vanilla**, the base implementation

described in Section 4; (2) **Cache**, the base algorithm that uses caching to avoid
 770 doing the same API call more than once; (3) **WCO**, the implementation of our
 WCO algorithm (3) **WCO + Cache**, which is the worst-case optimal algorithm
 avoiding duplicate calls via caching.

7.1. Berlin Benchmark

The Berlin benchmark dataset [10] is inspired by an e-commerce use case.
 775 It consists of 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 benchmark itself is composed of 12 queries, named Q1-Q12,
 and the size of the dataset is specified by the user. To test our implementation
 we generate 3 distinct datasets of different size. The specifications can be found
 780 at the Table 2.

Dataset	Number of triples	Size
D_1	19625500	5 Gb
D_2	41795444	10 Gb
D_3	79881347	20 Gb

Table 2: Sizes of the datasets created for the Berlin Benchmark.

Adapting the Berlin benchmark to include API calls. Our adapta-
 tion consists of exposing the data of five recurrent patterns we find in the
 benchmark queries as APIs that return JSON documents. For instance, one
 such pattern is $\{?x \text{ bsbm:productPropertyNumericZ } ?y\}$, where Z is a num-
 ber between 1 and 5. This pattern is used to return the value of some nu-
 meric property of a product with the label $?x$, so we created a (local) API
 route `api/numeric-properties/{label}`, that will return all the values of nu-
 meric properties of an object as a JSON file. That is, if a product with the
 IRI `bsbm:Product1` has a label "Product1", and its numeric properties are
`PropertyNumeric1 = 3, PropertyNumeric2 = 10`, then the request

`api/numeric-properties/Product1`

returns the JSON: { "p1": 3, "p2": 10 }.

All together, we simulate an API that has the following five routes:

- `api/numeric-properties/{label}`, which receives a label of a product and returns a JSON containing all of the product's numeric values as pairs
785 "propertyNumericX" : n, with n the value of `propertyNumericX`.
- `api/textual-properties/{label}`, which receives a label of a product and returns a JSON containing all of the product's numeric values as pairs
"propertyTextualX" : s, with s the value of `propertyTextualX`.
- `api/features/{label}`, which receives a label of a product and it returns
790 an array with all the labels of its features.
- `api/offers/{offer-id}`, which receives the id of an Offer and returns the properties of this offer.
- `api/review/{review-id}`, which receives the id of a Review and returns the properties of this review.

795 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:

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

We do a similar transformation for each of the other four patterns that were exposed as APIs. We run all the queries of the Berlin Benchmark except Q6, Q9, and Q11, because they were too short to include API calls in their patterns. To understand the results of the experiment, first we have to discuss the meaning
805 of each query [10].

- **Q1:** Find products for a given set of generic features.

- **Q2:** Retrieve basic information about a specific product for display purposes.
- **Q3:** Find products having some specific features and not having one feature.
- 810 • **Q4:** Find products matching two different sets of features.
- **Q5:** Find products that are similar to a given product.
- **Q7:** Retrieve in-depth information about a product including offers and reviews.
- 815 • **Q8:** Retrieve recent English language reviews for a specific product.
- **Q10:** Get cheap offers which fulfill the consumer’s delivery requirements.
- **Q12:** Export information about an offer into another schema.

We change the `OPTIONAL` operator in each query by `AND`, because the two are the same in terms of worst case optimal analysis. A specification of the queries that we run is available in Appendix A. Note that those queries depend
820 on some constants that we had to choose. Thus the values that we use and for more details about how to construct an API from the benchmark, you can view its source code at [1].

Results. Since our goal is to show how more involved algorithms reduce API
825 calls, we report the total number of calls issued by the Vanilla implementation when evaluating queries, as well as the calls issued by the other three implementations, in terms of the percentage with respect to the Vanilla evaluation. Here we do not include the results of the WCO algorithm without caching, since it does not present a significant improvement with respect to the Cache implementation and the best performance is obtained with the WCO+Cache algorithm.
830 Results for D_1 are presented in Table 3.

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

	Calls Vanilla	% Vanilla	% Cache	% WCO+Cache
Q1	28	100%	100%	89%
Q2	43	100%	6%	0%
Q3	28	100%	100%	39%
Q4	139	100%	100%	10%
Q5	346	100%	8%	8%
Q7	6	100%	100%	100%
Q8	2	100%	100%	0%
Q10	6	100%	100%	0%
Q12	1	100%	100%	100%
AVG	66	100%	40%	14%

Table 3: Number of API calls performed by the Vanilla implementation over D_1 as well as the percentage of this total issued by all four implementation. Berlin queries are adapted into SERVICE-to-API patterns. The last row shows the average percentage of calls that the algorithm does compared to Vanilla. The combination of WCO + Cache does 14% of the original calls done by the naive algorithm.

algorithm. As summary we show the average percentage of API calls done with
835 respect to the original algorithm. In average, the worst-case optimal algorithm
with caching does a 14% of the calls done by the naive algorithm.

Here we note that Q2, Q5, Q7 and Q8 are queries that fix some product.
Thus, when we use the WCO algorithm, all the triple patterns related to such
product are resolved before requesting data to the API, hence the number of
840 calls to the API is reduced dramatically except for Q7, where the resolution of
the triple patterns related to the product does not filter the possible answers.
Also for query Q12, the query involves a single offer that is also fixed but it is
not being filtered by the following triple patterns, and then the number of API
calls remains the same for every algorithm. It is important to note that not
845 every product is related with certain properties (such as `propertyNumeric4` or
`propertyNumeric5`), and then, it is possible that our algorithm resolves that
a query answer is empty before executing a SERVICE-to-API. This happens
for queries Q2, Q8 and Q10, where we reduce the number of API calls to zero.

850 Finally we note that for the other queries, the WCO algorithm is definitely an improvement with respect to the naive algorithm. The trend shown for this dataset holds for D_2 and D_3 as we can see in tables 4 and 5 respectively.

	Calls Vanilla	% Vanilla	% Cache	% WCO+Cache
Q1	170	100%	81%	77%
Q2	49	100%	6%	0%
Q3	163	100%	100%	34%
Q4	169	100%	100%	63%
Q5	1348	100%	7%	7%
Q7	6	100%	100%	100%
Q8	4	100%	100%	50%
Q10	6	100%	100%	0%
Q12	1	100%	100%	100%
AVG	209	100%	30%	20%

Table 4: Percentage of total issued by all four implementations over D_3 . The combination of WCO + Cache does 21% of the original calls done by the naive algorithm.

	Calls Vanilla	% Vanilla	% Cache	% WCO+Cache
Q1	343	100%	80%	73%
Q2	49	100%	6%	0%
Q3	323	100%	100%	31%
Q4	275	100%	100%	58%
Q5	2642	100%	10%	10%
Q7	5	100%	100%	100%
Q8	6	100%	100%	50%
Q10	5	100%	100%	20%
Q12	1	100%	100%	100%
AVG	405	100%	31%	21%

Table 5: Percentage of total issued by all four implementations over D_3 . The combination of WCO + Cache does 21% of the original calls done by the naive algorithm.

7.2. Wikidata Benchmark

Although the Berlin benchmark tests several operators of SPARQL, it produces patterns that are structurally simple, and therefore leave fewer room for
 855 better algorithms. To test for more complex patterns we use the Wikidata benchmark proposed by Hogan et al. [23] for testing the implementation of a worst-case optimal join algorithm in SPARQL.

The benchmark proposes 17 query templates, which are basic graph patterns that contains only variables. By instantiating the predicates in these patterns
 860 with different constants, each template gives rise to 50 instantiated queries. The predicates are taken from the Wikidata dataset, and each one of the queries is not empty when evaluated to such database. For example, one of the instantiations of the template

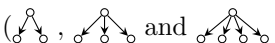
$$\{?x \text{ ?p1 } ?y . ?x \text{ ?p2 } ?z\}$$

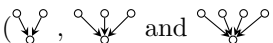
865 is the SPARQL query

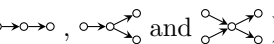
```
SELECT * WHERE { ?x wdt:P57 ?y . ?x wdt:P166 ?z }.
```

Let us describe the 17 templates used in the benchmark. For this, they are divided into templates with a single join variable and templates with multiple join variables:


870 • Single join variable:







– Trees (): join trees over a single join variable rooted on the subject.

– Inverted trees (): join trees over a single join variable but now rooted on the object.


875 – Joins (): the join variable combines subjects with objects.

• Multiple join variables:

– Paths (): paths of length three or four respectively.

- Triangles ( and ): BGPs asking for triangles over the database.
- 880 – Squares (, ,  and ): BGPs asking for squares over the database.

The definition of each template can be found in Appendix B.

Adapting the Wikidata benchmark to include API calls. Just as we did for the Berlin Benchmark, we need to adapt these 17 templates to include
 885 API calls into them. The way we do this is by choosing one of the triples of the pattern and replacing them with an API call that has the same information as the original database. We always chose a triple pattern in the middle of the template, unless the template has only two triple patterns, where for syntactic reasons, we chose the last one. For instance, consider the template  of the
 890 benchmark:

$$\{ ?x \text{ ?p01 } ?y . ?x \text{ ?p02 } ?z . ?x \text{ ?p03 } ?u \},$$

representing a tree where the join variable is the subject. One possible instantiation as a SERVICE-to-API query is the following:

```
SELECT * WHERE {
  895   ?x wdt:P2918 ?y .
      SERVICE <api/wdt:P1366/{?y}>{ (["z"]) AS (?z) } .
      ?x wdt:P1807 ?u
}
```

Again, from this template we produce 50 SERVICE-to-API patterns by re-
 900 placing property variables ?p01, ?p02 and ?p03 by concrete properties found in the Wikidata graph. A complete list of the adapted templates can be found in Appendix C.

Results. We use the same measure as before: if we say that the calls issued by queries under the Vanilla implementation corresponds to the 100% of calls,
 905 then we are interested in reducing the percentage of those calls that are issued by the other implementations: **Cache**, **WCO** and **WCO+Cache**. Note that

here we include the API calls done by the WCO algorithm without caching to understand how this technique on its own is an improvement with respect to the naive algorithm. Recall that there are 50 queries per each of the 17 templates; we only present the aggregate of all these 50 queries. The results can be seen in figures 1 and 2. Here *y axis* represents the percentage of calls done to the API with respect to the Vanilla implementation. The dot is the average percentage of calls over all 50 queries, and the boxes are bounded by the first quartile and the third quartile for each one of the templates (according to the results of their 50 queries). Within the box, the line represents the median.

Here we can see that the behaviour from the previous section is repeated again: the **WCO+Cache** algorithm produces a dramatical decrease in the number of API calls. But this benchmark allows us to understand the performance of the **WCO** algorithm on its own. We have to note that there are some patterns where the maximum value of the **WCO** algorithm goes beyond 100%; this means that for certain queries the algorithm does more API calls than the naive evaluation. This is because the order of execution of our algorithm, in some cases, induces more intermediate results. However, as the interquartile interval suggest, in most of the queries we have less calls to the API than the naive algorithm and even than the Cache version of the algorithm. This hints that our WCO algorithm, which evaluates in a different way the query than the naive algorithm, is essential for the performance obtained by the **WCO+Cache** algorithm.

Finally, we note some queries such as \mathcal{P}_8 or \mathcal{P}_9 where the WCO algorithm does not mark any difference. This is because this queries are formed by two patterns: a single triple pattern and a SERVICE-to-API pattern. Therefore, the query plan for the WCO algorithm is equivalent to the naive algorithm, because there is a single way to evaluate the query.

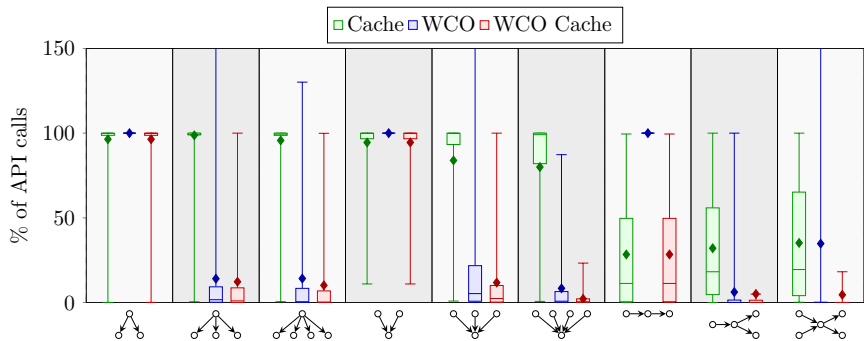


Figure 1: Boxplots of runtimes for adapted queries of the Wikidata benchmark

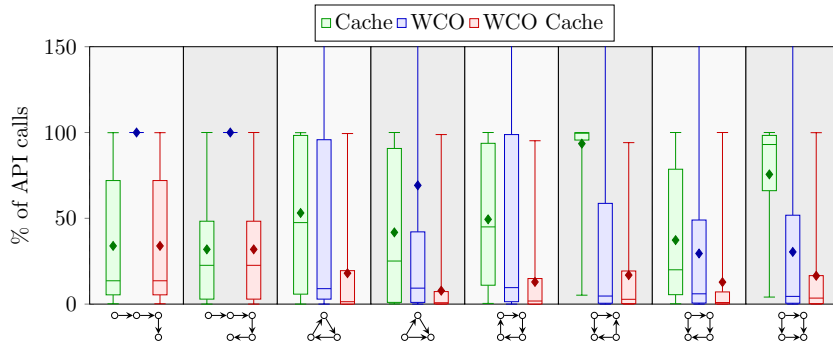


Figure 2: Boxplots of runtimes for adapted queries of the Wikidata benchmark. WCO and WCO + Cache produce the steepest decrease in API calls.

8. Conclusion

935 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 through a series of experiments.

940 Moving forward, one of our main goals is to support and test the extended
 SERVICE operator in the context of public SPARQL endpoints. This will require
 us to deal with the authentication issue present in many APIs, and adding
 support for such operations to SPARQL seems to be a non-trivial task. In this

context, we would also like to test whether one can issue API calls in parallel,
945 and how this affects the performance of the algorithms we propose in this work.

An orthogonal line of future work is to also explore the potential for automatic entity resolution based on an API answer; this is, to transform the API information such as literals representing names of people or places back into IRIs. Thus, it would be possible to achieve a better integration between the
950 API data and the RDF Graph. Also, we would like to explore how our extension can handle real-world issues related to APIs, such as to manage APIs that paginate the results or that do not accept a high number of requests in short intervals of time. Obviously, it is also necessary to explore how our worst-case optimal algorithm can be adapted to handle this kind of issues. Finally, we are
955 also interesting in extending the API coverage by supporting formats other than JSON.

Acknowledgements. This work was funded by ANID - Millennium Science Initiative Program - Code ICN17_002, by CONICYT FONDECYT Regular Project Number 1170866 and by CONICYT-PCHA Doctorado Nacional 2017-
960 21171731.

- [1] SERVICE-to-API implementation. <https://github.com/alanezz/IS-SPAPI>.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- 965 [3] 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.
- [4] 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.
- 970 [5] C. B. Aranda, A. Polleres, and J. Umbrich. Strategies for executing federated queries in SPARQL1.1. In *ISWC 2014*, pages 390–405, 2014.

- [6] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.
- [7] 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. 975
- [8] W. Beek, L. Rietveld, S. Schlobach, and F. van Harmelen. Lod laundromat: Why the semantic web needs centralization (even if we don’t like it). *IEEE Internet Computing*, 20(2):78–81, 2016.
- [9] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6):690–701, 2015. 980
- [10] C. Bizer and A. Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
- [11] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *CoRR*, abs/1708.00363, 2017.
- [12] T. Bray. The javascript object notation (json) data interchange format. 2014. 985
- [13] A. Cali and D. Martinenghi. Querying data under access limitations. In *ICDE 2008*, pages 50–59, 2008.
- [14] A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos. Semagrow: Optimizing federated sparql queries. In *Proceedings of the 11th International Conference on Semantic Systems*, pages 121–128, 2015. 990
- [15] 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.
- [16] P. Fafalios and Y. Tzitzikas. SPARQL-LD: a SPARQL extension for fetching and querying linked data. In *ISWC Demos*, 2015. 995

- [17] P. Fafalios, T. Yannakis, and Y. Tzitzikas. Querying the web of data with SPARQL-LD. In *TPDL 2016*, pages 175–187, 2016.
- [18] F. Galiegue and K. Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.
- 1000 [19] 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.
- [20] 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.
- 1005 [21] S. Harris and A. Seaborne. SPARQL 1.1 query language. *W3C*, 2013.
- [22] O. Hartig, C. Bizer, and J.-C. Freytag. Executing sparql queries over the web of linked data. In *International Semantic Web Conference*, pages 293–309. Springer, 2009.
- 1010 [23] A. Hogan, C. Riveros, C. Rojas, and A. Soto. A worst-case optimal join algorithm for SPARQL. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I*, pages 258–275, 2019.
- [24] IETF. URI Template. <https://tools.ietf.org/html/rfc6570>, 2012.
- 1015 [25] M. Junemann, J. L. Reutter, A. Soto, and D. Vrgoč. Incorporating API data into SPARQL query answers. In *ISWC 2016 Posters & Demos*, 2016.
- [26] N. Kobayashi, M. Ishii, S. Takahashi, Y. Mochizuki, A. Matsushima, and T. Toyoda. Semantic-json. *Nucleic Acids Research*, 39:533–540, 2011.
- [27] P. Lisena, A. Meroño-Peñuela, T. Kuhn, and R. Troncy. Easy web api development with sparql transformer. In *International Semantic Web Conference*, pages 454–470. Springer, 2019.
- 1020

- [28] A. Meroño-Peñuela and R. Hoekstra. Automatic query-centric api for routine access to linked data. In *International Semantic Web Conference*, pages 334–349. Springer, 2017.
- 1025 [29] G. Montoya, H. Skaf-Molli, and K. Hose. The odyssey approach for optimizing federated sparql queries. In *International Semantic Web Conference*, pages 471–489. Springer, 2017.
- [30] G. Montoya, M. Vidal, and M. Acosta. A heuristic-based approach for planning federated SPARQL queries. In *COLD 2012*, 2012.
- 1030 [31] 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.
- [32] M. Mosser, F. Pieressa, J. L. Reutter, A. Soto, and D. Vrgoc. Querying apis with SPARQL: language and worst-case optimal algorithms. In *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, pages 639–654, 2018.
- 1035 [33] M. Mosser, F. Pieressa, J. L. Reutter, A. Soto, and D. Vrgoc. Querying apis with SPARQL. In *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*, 2019.
- 1040 [34] 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.
- [35] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *PODS 2012*, pages 37–48, 2012.
- 1045 [36] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

- [37] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In *WWW 2016*, pages 263–273, 2016.
- 1050 [38] E. Prud’hommeaux and C. Buil-Aranda. SPARQL 1.1 Federated Query. *W3C Recommendation*, 21, 2013.
- [39] L. Rietveld and R. Hoekstra. YASGUI: not just another SPARQL client. In *ESWC 2013*, pages 78–86, 2013.
- [40] M. Saleem, A. Potocki, T. Soru, O. Hartig, and A.-C. N. Ngomo. Costfed:
1055 Cost-based query optimization for sparql endpoint federation. *Procedia Computer Science*, 137:163–174, 2018.
- [41] B. Stringer, A. Meroño-Peñuela, A. Loizou, S. Abeln, and J. Heringa. Scry: Enabling quantitative reasoning in sparql queries. In *SWAT4LS*, pages 214–215, 2015.
- 1060 [42] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple pattern fragments: a low-cost knowledge graph interface for the web. *Journal of Web Semantics*, 37:184–206, 2016.
- [43] T. Yannakis, P. Fafalios, and Y. Tzitzikas. Heuristics-based query reordering for federated queries in SPARQL 1.1 and SPARQL-LD. In *Proceedings of the 3rd International Workshop on Geospatial Linked Data and the 2nd Workshop on Querying the Web of Data co-located with 15th Extended Semantic Web Conference (ESWC 2018), Heraklion, Greece, June 3, 2018*, volume 2110 of *CEUR Workshop Proceedings*, pages 74–88. CEUR-WS.org,
1065 2018.
- 1070

Appendix A. Berlin Benchmark Queries

- Q1:

```
PREFIX ex: <http://example.org/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
1075 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product rdf:type %ProductType% .
1080 SERVICE <http://localhost:5000/features/{label}>{
        ($.values[*]) AS (?v1)
    }
    FILTER(?v1 = %ProductFeature1%)
    SERVICE <http://localhost:5000/features/{label}>{
1085 ($.values[*]) AS (?v2)"
    }
    FILTER(?v2 = %ProductFeature2%)
    ?product bsbm:productPropertyNumeric1 ?v3
    FILTER (?v3 > 500)
1090 }
```

- Q2:

```
PREFIX ex: <http://example.org/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
1095 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
    %Product% rdfs:label ?label .
    %Product% rdfs:comment ?comment .
```

```

%Product% bsbm:producer ?p .
1100 ?p rdfs:label ?producer .
SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?f)
}
SERVICE <http://localhost:5000/textual/{label}>{
1105 ($.p1, $.p2, $.p3) AS
    (?propertyTextual1, ?propertyTextual2, ?propertyTextual3)
}
SERVICE <http://localhost:5000/numeric/{label}>{
    ($.p1, $.p2) AS (?propertyNumeric1, ?propertyNumeric2)
1110 }
%Product% bsbm:productPropertyTextual4 ?p4 .
%Product% bsbm:productPropertyTextual5 ?p5
}

```

- Q3:

```

1115 PREFIX ex: <http://example.org/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
1120 ?product rdfs:label ?label .
?product rdf:type %ProductType% .
SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?f)
}
1125 FILTER(?f = %ProductFeature%)
?product bsbm:productPropertyNumeric1 ?p1 .
FILTER ( ?p1 > %value%)

```

```

        ?product bsbm:productPropertyNumeric2 ?p2
        FILTER (?p2 < %value2% )
1130     }

```

- Q4:

```

PREFIX ex: <http://example.org/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
1135 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
    ?product rdfs:label ?label .
    ?product rdf:type %ProductType% .
    SERVICE <http://localhost:5000/features/{label}>{
1140     ($.values[*]) AS (?v1)
    }
    FILTER(?v1 = %ProductFeature%)
    SERVICE <http://localhost:5000/features/{label}>{
    ($.values[*]) AS (?v2)
1145     }
    FILTER((?v2 = %ProductFeature2% || ?v2 = %ProductFeature3%))
    ?product bsbm:productPropertyNumeric1 ?p1 .
    FILTER (?p1 > %value%)
}

```

- Q5:

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
1155 PREFIX rev: <http://purl.org/stuff/rev#>

```

```

PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
    ?product rdfs:label ?label .
1160    ?product rdf:type %ProductType% .
    %Product% rdfs:label ?label2
    FILTER (%Product% != ?product)
    %Product% bsbm:productFeature ?f .
    ?product bsbm:productFeature ?f
1165    SERVICE <http://localhost:5000/numeric/{label}>{
        ($.p1) AS (?simp1)
    }
    SERVICE <http://localhost:5000/numeric/{label2}>{
        ($.p1) AS (?origp1)
1170    }
    FILTER (?simp1 < (?origp1 + 120) && ?simp1 > (?origp1 - 120))
    SERVICE <http://localhost:5000/numeric/{label}>{
        ($.p2) AS (?simp2)
    }
1175    SERVICE <http://localhost:5000/numeric/{label2}>{
        ($.p2) AS (?origp2)
    }
    FILTER (?simp2 < (?origp2 + 500) && ?simp2 > (?origp2 - 500))
}

1180 • Q7:

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>

```

```

1185 PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
    %Product% rdfs:label ?label .
1190 ?offer bsbm:product %Product% .
    ?offer ex:id ?id
    SERVICE <http://localhost:5000/offer/{id}>{
        ($.price, $.vendor, $.country) AS (?pr, ?vendor, ?country)
    }
1195 FILTER(?country = \"http://downlode.org/rdf/iso-3166/countries#GB\")
    ?review bsbm:reviewFor %Product% .
    ?review ex:id ?id2 .
    SERVICE <http://localhost:5000/review/{id2}>{
        ($.revName, $.revTitle) AS (?revName, ?revTitle)
1200 }
    ?review bsbm:rating1 ?rating1 . ?review bsbm:rating2 ?rating2
}

```

- Q8:

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
1205 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
1210 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
    %Product% rdfs:label ?label .
    ?review bsbm:reviewFor %Product% .
}

```

```

?review ex:id ?id2 .
1215 SERVICE <http://localhost:5000/review/{id2}>{
    ($.revName, $.revTitle, $revText) AS (?revName, ?revTitle, ?revText)
}
?review bsbm:rating1 ?rating1 .
?review bsbm:rating2 ?rating2 .
1220 ?review bsbm:rating3 ?rating3 .
?review bsbm:rating4 ?rating4
}

```

- Q10:

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
1225 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
1230 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
    %Product% rdfs:label ?label .
    ?offer bsbm:product %Product% .
    ?offer ex:id ?id .
1235 SERVICE <http://localhost:5000/offer/{id}>{
    ($.price, $.vendor, $.country) AS (?price, ?vendor, ?country)
}
    ?offer bsbm:deliveryDays ?devDays .
    FILTER(?devDays < %value%)
1240 }

```

- Q12:

```

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
1245 PREFIX ex: <http://example.org/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
1250   %Offer% bsbm:product ?p .
        ?p rdfs:label ?label .
        %Offer% ex:id ?id
        SERVICE <http://localhost:5000/offer/{id}>{
          ($.price, $.vendor) AS (?price, ?vendor)
1255   }
        %Offer% bsbm:deliveryDays ?devDays .
        %Offer% bsbm:offerWebpage ?offerURL .
        %Offer% bsbm:validTo ?validTo
    }

```

1260 **Appendix B. Templates of the Wikidata benchmark**

Appendix C. Adapted templates of the Wikidata benchmark



```

SELECT * WHERE {
    ?x ?p1 ?y .
1265   SERVICE <api/{p2}/{x}>{ (["z"]) AS (?z) } .
    }

```



```

SELECT * WHERE {

```

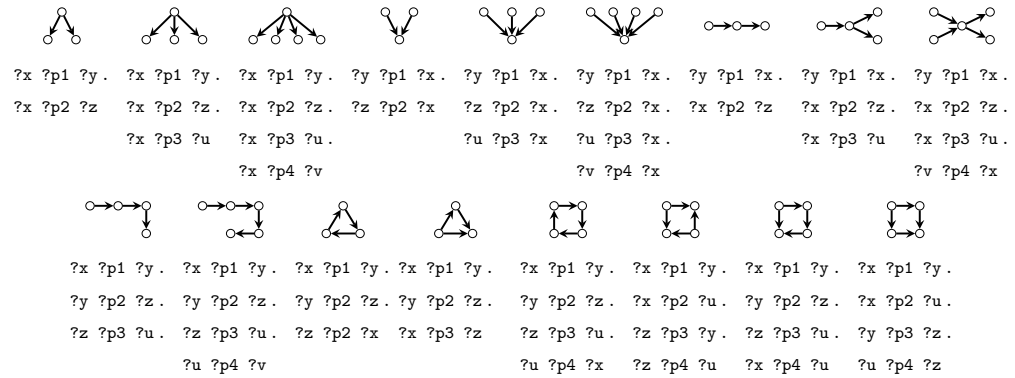





Figure B.3: Templates and their associated diagram. Obtained from [23].

```


?x ?p1 ?y .
1270 SERVICE <api/{?p2}/{?x}>{ (["z"]) AS (?z) } .
?x ?p3 ?u .
}

• 

SELECT * WHERE {
?x ?p1 ?y .
1275 SERVICE <api/{?p2}/{?x}>{ (["z"]) AS (?z) } .
?x ?p3 ?u .
?x ?p3 ?v .
}

1280 • 

SELECT * WHERE {
?y ?p1 ?x .
SERVICE <api-inv/{?p2}/{?x}>{ (["z"]) AS (?z) } .
}

1285 • 

```

```

SELECT * WHERE {
  ?y ?p1 ?x .
  SERVICE <api-inv/{?p2}/{?x}>{ (["z"]) AS (?z) } .
  ?u ?p3 ?x .
1290 }

```



```

SELECT * WHERE {
  ?y ?p1 ?x .
  SERVICE <api-inv/{?p2}/{?x}>{ (["z"]) AS (?z) } .
1295 ?u ?p3 ?x .
  ?v ?p4 ?x .
}

```



```

SELECT * WHERE {
1300 ?y ?p1 ?x .
  SERVICE <api/{?p2}/{?x}>{ (["z"]) AS (?z) } .
}

```



```

SELECT * WHERE {
1305 ?y ?p1 ?x .
  SERVICE <api/{?p2}/{?x}>{ (["z"]) AS (?z) } .
  ?x ?p3 ?u .
}

```



```

1310 SELECT * WHERE {

```

```

    ?y ?p1 ?x .
    SERVICE <api/{?p2}/{?x}>{ (["z"]) AS (?z) } .
    ?x ?p3 ?u .
    ?v ?p4 ?x .
1315 }

```



```

    SELECT * WHERE {
        ?x ?p1 ?y .
        SERVICE <api/{?p2}/{?y}>{ (["z"]) AS (?z) } .
1320 ?z ?p3 ?u .
    }

```



```

    SELECT * WHERE {
        ?x ?p1 ?y .
1325 SERVICE <api/{?p2}/{?y}>{ (["z"]) AS (?z) } .
        ?z ?p3 ?u .
        ?u ?p4 ?v .
    }

```



```

1330 SELECT * WHERE {
        ?x ?p1 ?y .
        SERVICE <api/{?p2}/{?y}>{ (["z"]) AS (?z) } .
        ?z ?p3 ?x .
    }

```



```

SELECT * WHERE {
    ?x ?p1 ?y .
    SERVICE <api/{?p2}/{?y}>{ (["z"]) AS (?z) } .
    ?x ?p3 ?z .
1340 }

```



```

SELECT * WHERE {
    ?x ?p1 ?y .
    SERVICE <api/{?p2}/{?y}>{ (["z"]) AS (?z) } .
1345 ?z ?p3 ?u .
    ?u ?p4 ?x .
}

```



```

SELECT * WHERE {
1350 ?x ?p1 ?y .
    SERVICE <api/{?p2}/{?x}>{ (["u"]) AS (?u) } .
    ?z ?p3 ?y .
    ?z ?p4 ?u .
}

```



```

SELECT * WHERE {
    ?x ?p1 ?y .
    SERVICE <api/{?p2}/{?y}>{ (["z"]) AS (?z) } .
    ?z ?p3 ?u .
1360 ?x ?p4 ?u .
}

```



1365

```
SELECT * WHERE {  
  ?x ?p1 ?y .  
  SERVICE <api/{?p2}/{?x}>{ (["u"]) AS (?u) } .  
  ?y ?p3 ?z .  
  ?u ?p4 ?z .  
}
```