On the Fly SPARQL Execution for Structured Non-RDF Web APIs

Torsten Spieldenner
*German Research Center for Artificial Intelligence (DFKI), Saarland Informatics Campus D 3 2,
Saarbrucken Graduate School of Computer Science, 66123 Saarbrucken, Germany*

Keywords: Linked Data, Semantic Web, Resource Description Framework, RDF, SPARQL, Web API, Structured Data.

Abstract: The concept of the Semantic Web, built around the idea of semantically described Linked Data and the data model of the Resource Description Framework (RDF), has become a prominent idea of seamless access to, and integration of, data. The number of tools to translate from non-RDF to RDF-representation of data has since then been ever increasing. However, to this day, numerous Web APIs do offer time critical data only in non-RDF-formats. Examples for this are traffic and public transport live data. Due to its nature, an offline data dump, as mostly generated by RDF lifting translation tools, is not practical, as it becomes inconsistent with the original data quickly. In this paper, we for this present an approach, that, published as a micro-service, allows to send semantic queries against the legacy Web interfaces directly, and return the result in RDF. The service API follows the SPARQL 1.1 Query API specification, and also supports federated queries over distributed endpoints, allowing an easy and accessible way for semantically enriched data integration over legacy endpoints.

1 INTRODUCTION

In the last years, the concept of the *Semantic Web* (Berners-Lee et al., 2001) has developed to a prominent idea of seamless integration of, and access to, data between different providers and applications. Among others, Verborgh (Verborgh et al., 2011) and Mayer (Mayer et al., 2016) emphasize the importance of sufficiently semantically described server APIs (along with their data). The Semantic Web is based on the idea of Linked Data (Bizer et al., 2011), a set of best practices for publishing structured data on the Web. Linked Data is usually published in terms of RDF (*Resource Description Framework*) graphs[1] to establish links between all kinds of addressable Web resources. However, to this day, there is still a tremendous number of resources being published as structured data that does not yet follow Linked Data principles. For many text-based structured resource representations like XML, JSON or CSV, generic approaches have been developed to map them to RDF (Das et al., 2012; Scharffe et al., 2012; Dimou et al., 2013; Michel et al., 2017). These approaches, however, are commonly used to create a copy of the original data as Linked Data data dump. This is not practical for data that is changing fast, such as live feeds or streams. For these, data dumps may soon become inconsistent with the original live data when freshness of the data would be crucial.

An example for this is live traffic and public transport connection data. For this kind of data, the JSON-based General Transport Feed Specification (GTFS),[2] along with extensions like the Google Proto Buffer[3] based GTFS-Realtime,[4] has become a de-facto standard. For an emerging number of bicycle sharing stations, the comparable General Bikeshare Feed Specification (GBFS)[5] is gaining momentum in acceptance worldwide. Some work already motivates the use available open data for route planning (Nallur et al., 2015). Efforts have also been made to model traffic data in terms of Linked Data vocabularies (Colpaert et al., 2017; Colpaert et al., 2019), also tackling the problem of integrating both static datasets, and dynamic Linked Data feeds (Harth et al., 2013).

These approaches, however, often consider

[1]RDF 1.1 Primer document (Jul. 2020): https://www.w3.org/TR/rdf11-primer/

[2]GTFS Static Specification (Jul. 2020): https://developers.google.com/transit/gtfs/

[3]Google Protocol Buffer (Jul. 2020): https://developers.google.com/protocol-buffers

[4]GTFS Realtime Specification (Jul. 2020): https://developers.google.com/transit/gtfs-realtime

[5]GBFS Specification (Jul. 2020): https://developers.google.com/transit/gtfs-realtime

Spieldenner, T.
On the Fly SPARQL Execution for Structured Non-RDF Web APIs.
DOI: 10.5220/0010142102430252
In *Proceedings of the 16th International Conference on Web Information Systems and Technologies (WEBIST 2020)*, pages 243-252
ISBN: 978-989-758-478-7

Linked Data and traditional data services as two separate worlds. Either they do not consider Linked Data as suitable representation at all (as in the work by Nallur et al. (Nallur et al., 2015)), or in the case of Colpaert (Colpaert et al., 2017), or Harth (Harth et al., 2013), they assume readily available Linked Data representations of the data of interest, and may suggest translation steps to lift existing data to the required Linked Data representation. The problem of integrating live data that is not yet provided in a semantic Linked Data representation into a Linked Data application is mostly untackled. In fact, a lack of usable tools to make the transition between the traditional world of data, and the Linked Data world, has recently been identified as a main bottle neck that hinders the uptake of concepts from Semantic Web in application in industry and public sectors (Verborgh and Vander Sande, 2020).

As a remedy, we present in this paper a system, implemented as microservice, that provides a SPARQL 1.1 Query[6] service behind an API that is fully compliant to the SPARQL 1.1 protocol interface.[7] It allows to specify remote sources, perform a provided query against them, and return as result a SPARQL query result in RDF representation.

The remainder of this paper is structured as follows: In Section 2, we provide an overview of existing non-RDF to RDF lifting approaches, as well as approaches that allow to query structured data as Linked Data. We revisit the most relevant notions of the RDF data model and the SPARQL query language in Section 3. From these, we derive a formal definition of our SPARQL query service in Section 4, provide a thorough description of the resulting service implementation in Section 5, and give detailed examples of its usability in a selected use-case from the transportation domain in section 6. We finally conclude the paper and give an outlook over future work in Section 7.

2 RELATED WORK

From the beginning of the Semantic Web, the mapping of existing structural data to RDF is an active research area. The so called *lifting and lowering* of data is required to access and modify any non RDF datasource from Linked Data applications (Sect. 2.1).

[6]W3C SPARQL 1.1 Query Language Recommendation (Jul. 2020): https://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[7]W3C SPARQL 1.1 Protocol Recommendation (Jul. 2020): https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

In addition to lifting non-RDF data to an RDF representation, research has also carried out on how to generate RDF data from evaluation semantic queries on non-semantic data sources directly (Sect. 2.2).

2.1 RDF-lifting Approaches

A prominent example for lifting relational databases to RDF is *R2RML* (Das et al., 2012). R2RML specifies mappings from database schemas to RDF graphs in RDF turtle syntax. A respective mapping file describes table structure and content, and where to inject the data from these tables into a target RDF graph.

Several extensions to R2RML have been presented: The tool *Karma* by Gupta et al. (Gupta et al., 2012) is a visual tool to define R2RML mappings by annotating relational data. Karma supports the user by inference of suitable target mappings from previously annotated data. The resulting mappings can be published as service with REST API for online data conversion. Dimou et al. extended R2RML to *RML* (Dimou et al., 2013; Dimou et al., 2014a; Dimou et al., 2014b), a superset of the R2RML mapping language, that also allows for mapping of structural datasources that are not stored in relational databases. Source formats for RML include CSV, TSV, XML, and JSON.

Slepicka et al. present *KR2RML* (Slepicka et al., 2015), a different approach to extending R2RML to support heterogeneous sources, that keeps in mind extendability and scalability with respect to changes in the source data; aspects in which Slepicka et al. see shortcomings in RML. Finally, the implementation *CARML*,[8] an extension to RML, allows dynamic input streams as input to an RML mapping, instead of specifying to the to be mapped source directly in the mapping file. This is a crucial feature for re-using mappings for a variety of different structurally equivalent source files.

2.2 SPARQL Query Interfaces to Non-RDF Datasources

A different approach to make non-semantic datasets accessible for Linked Data applications is to provide a SPARQL query endpoint to clients, and transparently lift the queried non-RDF source upon receiving a query by a client.

In 2004, Bizer et al. presented *D2RQ* (Bizer and Seaborne, 2004). D2RQ provides a mapping language from relational database schemata to RDF, sim-

[8]CARML GitHub Repository: https://github.com/carml/carml

ilar to R2RML. Clients may send requests to the platform to perform SPARQL queries against a database, or explore a database as Linked Data, while the D2RQ platform performs the lifting of the database to RDF transparently, based on a previously defined mapping.

Similar approaches realize SPARQL-to-SQL mappings by employing R2RML based liftings (Rodríguez-Muro and Rezk, 2015; Priyatna et al., 2014; Calvanese et al., 2017). For non-relational datasources, Michel et al. present an approach to query the document-based MongoDB by employing *xR2RML* (Michel et al., 2017; Michel et al., 2016), an extension for R2RML for non-relational sources.

SPARQL-Generate (Lefrançois et al., 2017) integrates RDF generation from non-RDF datasources directly into the SPARQL-query itself. This removes the need of separately provided mapping files, however, it requires that the target SPARQL processor implements SPARQL-generate on top of SPARQL 1.1.

Finally, *SPARQL-Microservice* (Michel et al., 2018a; Michel et al., 2018b) provides a SPARQL query interface to wrap existing, JSON-based Web APIs.

The presented approaches in 2.1 cover the translation of legacy data to Linked Data data dumps. Most of them take offline approaches to lift data to RDF, a way we found impractical for live data.

From the approaches in 2.2, many target relational databases rather than Web APIs, with D2RQ as one chosen example. Our approach compares best to SPARQL Generate and SPARQL Microservice, which create the Linked Data representation from structured legacy data transparently as query response. However, SPARQL Generate requires an extended SPARQL implementation beyond the generally used specification. SPARQL Microservice requires cumbersome configuration during deploy time. It is moreover limited to JSON-LD as lifting result, which makes it difficult to impossible to use it in scenarios where the source data are not JSON, or when another result representation is needed.

Our approach overcomes these shortcomings by complying fully to the SPARQL 1.1 protocol specification. It uses RML as underlying mapping and thus supports a wide range of source- and target formats, making it more flexible and versatile than existing approaches. Lifting of target data is done transparently in an online step during query execution. A client can therefore use the service to directly query non-RDF legacy data, as if the queried source was a Linked Data source.

3 PRELIMINARIES

This section will handle in brief the basics of the RDF graph model and the SPARQL Query language. The RDF graph model will be described according to the contents of the RDF Primer document (see Footnote [1]). The basics established in this section will in the following be used to define our service.

3.1 RDF Graph Model

The Resource Description Framework (RDF) is the standard data model for the Semantic Web.

Let in the following $\mathbf{I}$, $\mathbf{L}$ and $\mathbf{B}$ be pairwise disjoint infinite sets of IRIs (Internationalized Resource Identifiers), literals and blank nodes, respectively. We will refer to elements in the union set in $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$ as RDF terms. The subset $\mathbf{T} = \mathbf{I} \cup \mathbf{L}$ of RDF terms denotes resources in some universe of discourse.

Blank nodes in $\mathbf{B}$ are to be understood that they indicate the existence of a resource, the content of which is described directly in the blank node itself, but they do not use an IRI to identify a particular resource.

The infinite set of all RDF triples is $\mathcal{T} = (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{T} \cup \mathbf{B})$. Asserting an RDF triple (s, p, o) says that some resource, denoted by p, establishes a binary relationship between the resources denoted by s and o.

An *RDF graph* $G \subset \mathcal{T}$ is thrn a finite set of RDF triples of the form (s, p, o).

3.2 SPARQL Query Language

The SPARQL Query Language (SPARQL) is the W3C recommended query language for RDF datasets (see footnote [6]). SPARQL queries are built around a graph pattern matching facility, i.e. triple patterns, which form the core of the language. In the following, we briefly present the syntax and semantics of SPARQL graph patterns.

Following the formal evaluation algebra as proposed by Perez et al. (Pérez et al., 2006), we use $\mathbf{V}$ to denote the infinite set of variables that is disjoint from $\mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$. The set of variables occurring in a syntax expression E is given by $var(E)$.

A tuple from $(\mathbf{T} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{T} \cup \mathbf{V})$ is called a *triple pattern*. Triple pattern components may be bound, i.e. from set $\mathbf{T}$, or unbound, i.e. from set $\mathbf{V}$.

The semantics of SPARQL graph patterns is defined in terms of an evaluation function $[\![\cdot]\!]_G^{\mathcal{D}}$ that evaluates a SPARQL graph pattern P over a dataset $\mathcal{D}$ with an active RDF graph G. The result of this evaluation is a set of mappings $var(E) \to \mathbf{T}$ in case of a

SPARQL SELECT query, or an RDF Graph $g \subset \mathcal{T}$ in case of a SPARQL CONSTRUCT query.

4 SERVICE DEFINITION

From the notions of the previous section, we will now derive a formal definition of the service in terms of RDF liftings and SPARQL query execution. We first define the query execution as function on a (mapped) data source. Second, we define a SPARQL 1.1 query interface with the notions of the defined formal query approach.

4.1 Formal Definition

With the notions from Sect.3, we define moreover the following concepts:

We define

$$\mathcal{Q} = (\mathbf{T} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{T} \cup \mathbf{V}) \quad (1)$$

as a shortcut for the set of all Triple Patterns.

Let Σ denote a *machine readable alphabet*, Σ^* the set of all words over alphabet Σ, and $\Gamma \subset \Sigma^*$ a set of *datagrams* in a given structured data format encoded in alphabet Σ. Such structured data formats could for example be comma separated value (CSV), JSON, or XML documents, as emitted by a Web resource, but also binary streams following a deterministic structure or protocol.

A datagram $\gamma \in \Gamma$ is then a valid structured piece of data that is encoded in a machine readable alphabet Σ.

We then define a ***mapping function***

$$\mathbf{m} : (\Gamma \subset \Sigma^*) \to (\mathcal{G} \subset \mathcal{T}), \mathcal{T} = (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{T} \cup \mathbf{B}) \quad (2)$$

as a function that translates a given structured datagram γ into an RDF Graph $\mathbf{g} \in \mathcal{G}$.

Let $\Omega_{\mathcal{Q}}$ denote the set of result mappings of a triple pattern $P \in \mathcal{Q}$ according to (Buil-Aranda et al., 2013). A *service call* $\mathbf{s}$ is then a function $\mathbf{s}$ with the following properties:

$$\mathbf{s} : (\Gamma \times \mathcal{Q}) \to \Omega_{\mathcal{Q}} \quad (3)$$

$$\mathbf{s}(\gamma, P) = [\![P]\!]^{\mathcal{D}}_{\mathbf{m}(\gamma)}, \; \mathbf{m} : (\Gamma \subset \Sigma^*) \to (\mathcal{G} \subset \mathcal{T}) \quad (4)$$

Means, a service accepts as call parameters a tuple that consists of a datagram γ from a datagram syntax Γ, and a triple Pattern $P \in \mathcal{Q}$ in some SPARQL syntax.

The result of the service call is an evaluation of the triple pattern P against the result of a lifting operation $\mathbf{m}$ on datagram γ.

4.2 Service SPARQL Query API

Following, we define the API to the SPARQL Wrapping Service as superset on the W3C SPARQL 1.1 Protocol[9] specification. We define parameters to specify a SPARQL query that is to be evaluated, as well as structured legacy data on which to evaluate the query, or URIs that point to endpoints from where to retrieve the data respectively. The subset of parameters that provides necessary information for the execution of the SPARQL query should completely comply with the SPARQL 1.1 Protocol specification.

4.2.1 Requests

The SPARQL 1.1 Protocol Recommendation specifies three modes of query requests: Query by `HTTP GET` request with Query String parameters, by query via `HTTP POST` request, either with message body included as URL encoded query parameters, or as direct `POST` operation with all information contained in the message payload. Accordingly, a service call **s** is performed by an HTTP request with the following methods and parameters (see also Table 1):

Query via `GET`: The request is sent by the client via `HTTP GET` request to the service endpoint with no Content-Type header set, as request body is empty. The endpoint accepts as parameters *query* and *source*, with *query* being the properly serialized and URL encoded triple pattern P according to SPARQL 1.1. protocol specification, and *source* a reference to a resource from which the datagram γ can be retrieved, or a datagram γ as url-encoded string respectively.

Query via `POST` with URL Encoded Parameters: The request is sent by clients via `HTTP POST` to the service endpoint with Content-Type header set to `application/x-www-form-urlencoded`. The service accepts as parameters `query` and `source`. Parameters are URL-encoded and ampersand-separated. `query` contains the SPARQL triple pattern P, and `source` the datagram γ (or an URI from which γ can be retrieved).

Query via Direct `POST`: Clients send `HTTP POST` requests with Content-Type header set to `application/sparql-query`. The source datagram γ is provided as encoded URL parameter, either inline, or as URI from which γ can be retrieved. The SPARQL query P is provided as unescaped string within the message payload.

Obviously, the above definition satisfies the SPARQL 1.1 protocol specification with respect to

[9]https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

Table 1: Expected parameters for a SPARQL 1.1 Query service call, based on the original SPARQL 1.1 query protocol specification. The structured datagram γ takes the role of both Dataset $\mathcal{D}$ and (default) graph $\mathcal{G}$.

Method	**Query Parameters**	**Content-Type**	**Message Body**
`GET`	`query`=P (exactly 1), `source`=γ (ex. 1)	None	None
`POST` (URL enc. Parameters)	None	`application/` `x-www-form-urlencoded`	URL-enc., &-separated: `query`=P (exactly 1), `source`=γ (exactly 1)
`POST` (direct)	`source`=γ (ex. 1)	`application/` `sparql-query`	Unencoded SPARQL query string

necessary parameters. Our API does not yet consider specification of an RDF dataset $\mathcal{D}$ against which the query should be executed in terms of `default-graph-uri` or `named-graph-uri`. However, according to the SPARQL 1.1 protocol recommendation, these parameters are optional, and the specification states that, *"if an RDF Dataset is not specified in either the protocol request or the SPARQL query string, then implementations may execute the query against an implementation-defined default RDF dataset"*.[10] This default dataset is in our case the result of the mapping operation $\mathbf{m}(\gamma)$.

4.2.2 Responses

Following the SPARQL 1.1 protocol specification, a query request to a service **s** returns the SPARQL query result with a success status code *(2xx)*.

The service moreover returns codes *400 (Bad Request)* and *500 (Internal Server Error)* in case of a malformed query, or a failure to execute the query respectively, in accordance with the SPARQL 1.1 Protocol specification. The service moreover returns a 400 error code if the supplied datagram γ is syntactically incorrect. The service may moreover return:

422 (Unprocessable Entity), if γ, either provided directly via `HTTP POST`, or as URI reference for download, is syntactically correct, but the mapping **m** returns an error for some reason, or any parameter specifying γ is missing.

502 (Bad Gateway), if γ is provided by URI reference, and the service under `source-uri` returns an error.

[10]https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/#dataset

5 IMPLEMENTATION

This section will describe in detail the actual implementation of the previously defined service as a microservice. The overall architecture, and the components it is composed from, is provided in Section 5.1. 5.2 details out the service API beyond the SPARQL interface. We will give an overview of employed frameworks and libraries in our prototype implementation in Section 5.3.

5.1 Service Architecture

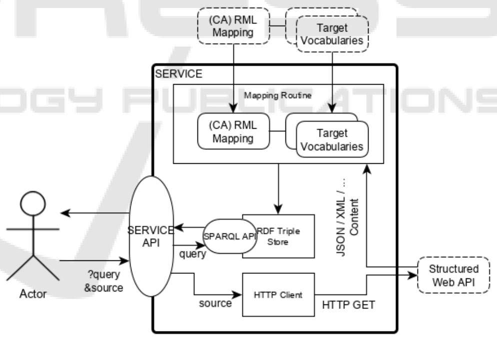

Figure 1: Architecture of the SPARQL API service.

We build the query service around the components as shown in Figure 1. Client requests are received by an *HTTP API* endpoint. This API accepts HTTP GET and POST Requests with parameters for `query` and `source` according to the specification in Section 4.2. Upon receiving a client request, an *HTTP Client* component sends HTTP GET request to the endpoint as specified by the `source` parameter. If this request returns an error, this error is forwarded to the requesting client according to the error handling routine as described in Section 4.2.

In case the remote source returns valid data, it is used as input for an *RML Mapping* component. The respective mapping is provided (for example in terms of an RML mapping file) by the service itself, and can be inspected by clients via an HTTP GET request to the respective resource according to Section 5.2.

The result of the mapping is stored in an *in-memory RDF Triple Store* that provides a SPARQL query API to the service application. If the mapping was successful, the query as provided by the client as parameter is executed against the triple store that contains the mapping result. Otherwise, an error according to Sec. 4.2 is returned.

Finally, the result of the query is returned to the client as result of its request (or an error, if execution of the query was not successful).

5.2 Service Self Information

```
{ "definitions": {
  "Bike": {
    "type": ["object"],
    "properties": {
      "bike_id": {"type": "string"},
      "lat": {"type": "number"},
      "lon": {"type": "number"},
      "is_reserved": {"type": "integer"},
      "is_disabled": {"type": "integer"}
  }},

  "BikeData": {
    "type": "object",
    "properties": {
      "bikes": {
        "type": "array",
        "items": {"$ref":
          "#/definitions/Bike"}
  }}} # end of BikeData

}, # end of definitions

  "type": "object",
  "properties": {
    "data": {"$ref":
      "#/definitions/BikeData"}
  },
  "required": ["data"]
}
```

Listing 1: Example of a JSON-Schema description of information conveyed by a `free_bike_status.json` datagram according to NABSA/GBFS General Bike Feed Specification.

In the current version, the service provides, listed as result of an HTTP `OPTIONS` request, routes to the following resources:

Under the route `/sourceformat/`, clients may retrieve the expected structure of source data. The source format is specified in JSON- or XML-Schema format, depending on the format that the service maps (see also Listing 1). The provided description may be used by clients to validate whether the service is capable of querying the intended legacy API according to JSON-/XML-Schema documentation.[11]

Moreover, the employed RML mapping file is provided under the route `/mapping/` for reasons of documentation. From the mapping file, client developers may learn employed ontologies or vocabularies in the resulting RDF SPARQL response, as returned by the service.

For future versions, we moreover plan to provide a definition of the output RDF format, for example in SHACL[12], to help clients to validate their local RDF representation automatically against the output that is generated by the service.

5.3 Prototype Implementation

Our prototype implementation is based on the Java Spring Framework[13] for the Web Service HTTP interface (*"SERVICE API"* in Fig. 1). RDF features are provided by the RDF4j[14] RDF library, using the RDF4j Repository API[15] for SPARQL Queries. The RDF4j Sail API[16] serves as temporal in-memory triple store to contain RDF mapping results against which the SPARQL Queries are executed (*"SPARQL API"* and *"RDF Triple Store"* in Fig. 1 respectively). The mappings are performed by the CARML[17] mapping framework. CARML extends RML mapping routines by the capability of defining a dynamic input stream as input the mapping, unlike RML, which expects a route to a fixed source.

Figure 2 shows the function call sequence between components of the service as chosen for our implementation: The HTTP interface provided by the Java Spring application server API receives a client request that specifies parameters `source` and `query` as either URL parameters or payload of a HTTP POST query. A `DataBuilder` class checks whether the supplied `source` argument is an URI, or already the datagram γ itself. In case of a it being an URI, a `WebAPIProxy` is used to retrieve γ from the provided API. γ then serves as `Structured Data` input for the mapping process.

[11]https://json-schema.org/, https://www.w3.org/XML/Schema

[12]https://www.w3.org/TR/shacl/

[13]Spring Framework Website (Feb. 2020): ttps://spring.io/

[14]RDF4j Website (Feb. 2020): https://rdf4j.org/

[15]RDF4j Repo. API (Feb. 2020): https://rdf4j.org/documentation/programming/repository/

[16]RDF4jSailAPI(Jul.2020):https://rdf4j.org/documentation/sail/

[17]CARML GitHub Repository: https://github.com/carml/carml

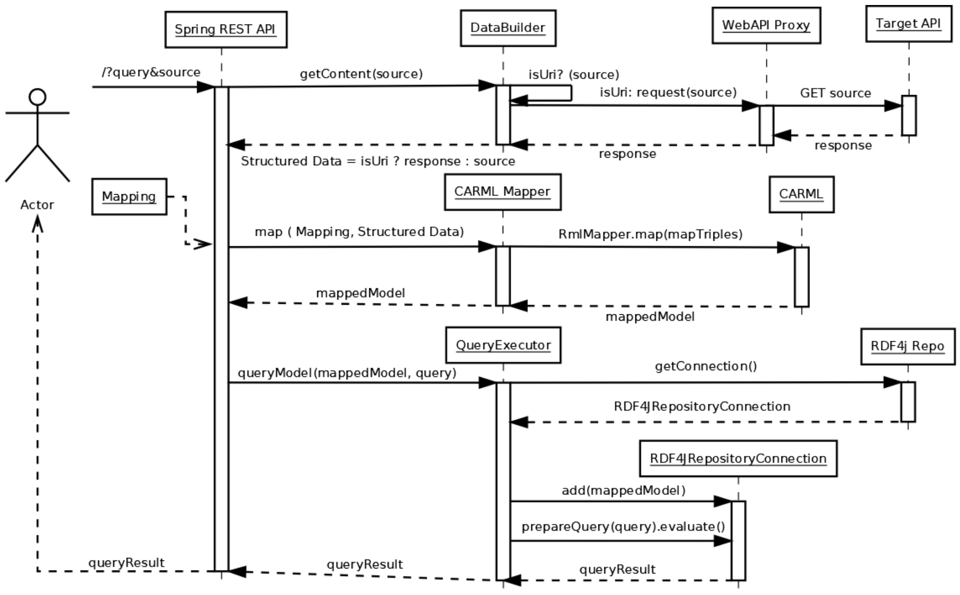

Figure 2: Call sequence between the different service components upon a client request to the API as defined in Sect. 4.2.

The (CA)RML `Mapping` file is provided by the service itself. It is fed to a `CARML Mapper` class that, after some preprocessing steps, uses the `CARML` mapping library to translate γ to a `mappedModel`, which will be in RDF Graph form.

The `mappedModel`, and the `query` as provided by the client, are used as input for an `QueryExecutor`. The `QueryExecutor` first opens a connection to a temporary `RDF4j Repo`, loads the `mappedModel` into it, and executes the `query` via the RDF4j Sail API.

The `queryResult` of this operation is finally returned to the client as result of the client's initial query request.

6 IN-USE EXAMPLES

Following, we demonstrate the usage of the service using examples from public transport and bike rental domain, the main application domain of the funding project SmartMaaS.

Second, we show how the presented service can be used to infer additional information from distributed data sets by employing distributed SPARQL queries over several SPARQL Wrapper services with the `SERVICE` keyword.

6.1 SELECT Query on JSON Data

```
SELECT ?name ?lat ?lon WHERE {
  ?station a gbfs:Station ;
  gbfs:name ?name ;
  wgs84_pos:lat ?lat ;
  wgs84_pos:long ?lon .
}
```

Listing 2: A SPARQL SELECT query that reads location information for bike sharing station from a GBFS service endpoint.

The following example demonstrates a simple SELECT query against a JSON data endpoint. The query as shown in Listing 2 is sent as `query` parameter to the service endpoint, using the JSON data as shown in Listing 3 as input. The RDF result is shown in Listing 4.

The overall execution time of the query in the example was about 180ms (milliseconds) for a dataset of 63 bike sharing station items. Of these 180ms, 30ms were spent on the CARML lifting process, and 10ms on the SPARQL query execution (measured on a Intel Core i7-4770k, 3.5GHz). The remaining time was spent to retrieve the source data from the provided URI as `source` parameter.

6.2 Federated Queries

The design of the Service API over parameterized, SPARQL 1.1 compliant request URLs also allows for federated SPARQL queries using the `SERVICE` keyword, as described in the respective W3C recommen-

```
{"last_updated": 1595835393,
 "ttl": 60,
  "data": {
   "stations": [
   {
     "station_id": "10044279",
     "name": "Bahnhof Beuel",
     "short_name": "4741",
     "lat": 50.739211,
     "lon": 7.126598,
     "region_id": "547"
   },
   {
     "station_id": "10044287",
     "name": "Haltepunkt Bonn-West",
     "short_name": "4742",
     "lat": 50.7367675,
     "lon": 7.0809567,
     "region_id": "547"
   }, ... ]
 }}
```

Listing 3: Input provided as example for a simple SELECT query (excerpt; source: https://gbfs.nextbike.net/maps/gbfs/v1/nextbike_bf/de/station_information.json).

dation document.[18]

In the formal W3C SPARQL 1.1 Grammar Recommendation,[19] a `ServiceGraphPattern` (entry `59` in the respective grammar document[20]) is defined as

`ServiceGraphPattern := 'SERVICE' 'SILENT'? VarOrIri GroupGraphPattern`

Accordingly, a federated query against a SPARQL Wrapper Service endpoint can be performed by employing as `VarOrIri` a valid URI against the SPARQL Wrapper Service API according to Section 4.2, and as *ServiceGraphPattern* the query that is to be executed against the dataset γ that is referred to as `source` (acc. to API definition in Section 4.2), employing the mapping $\mathbf{m}(\gamma)$ that is provided by the service under the URI that is provided as `VarOrIri` element in the query. The parameter `query` can be omitted in this case, as the Triple Pattern P that describes the query is already provided in terms of the `GroupGraphPattern` that follows the `VarOrIri` element of the federated query.

Listing 5 shows an example of a federated query. For brevity, the complete route to the service endpoint, as well source datagram γ are omitted, and given as `#srvpath` and `#srcpath` respectively.[21]

[18]W3C SPARQL 1.1 Federated Query recommendation: https://www.w3.org/TR/sparql11-federated-query/.

[19]SPARQL 1.1 Grammar: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#grammar

[20]As at current date, June 2020

[21]The complete URI used for the given example was http://sparql-wrapper.service/?source=https://gbfs.nextbike.net/maps/gbfs/v1/nextbike_bf/de/station_information.json

```
<results>
  <result>
   <binding name='name'>
    <literal>Bahnhof Beuel</literal>
   </binding>
  <binding name='lon'>
   <literal datatype='xsd:double'>
     7.126598
   </literal>
  </binding>
  <binding name='lat'>
   <literal datatype='xsd:double'>
    50.739211
   </literal>
  </binding>
  </result>
 <result>
  ....
</results>
```

Listing 4: Query result of the Query in Listing 2 against the data in Listing 3.

In our evaluation, the query given in Listing 5 was performed against an endpoint that emits information about the location of rentable bikes in GBFS JSON format. The `GraphGroupPattern` in the `SERVICE` clause merges that information with information about the location of rental bike stations of the same provider. The respective query response is shown in Listing 6. Note that the displayed information (*Which bike is currently parked at which station?*), is originally not provided by how the GBFS datamodel is defined. Deriving this information via semantic queries over the originally not semantically enriched GBFS data is a direct benefit from lifting queries against the GBFS data to a semantic representation.

The total execution time of the construct query was 560ms for a dataset of 63 bike station entries, and 649 items for free bikes respectively. Of these 560ms, approx. 10ms each were spent for the lifting process of both the datasets, and another 30ms to perform the query against the station information data in the SERVICE clause. The remaining times were spent to retrieve the source data from the provided URIs.

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel service that allows to perform SPARQL queries against non-RDF datasets. Unlike existing solutions, the presented service is not limited to a certain source format, as long

```
CONSTRUCT {
  ?station a gbfs:Station ,
          wgs84_pos:SpatialThing;
    gbfs:name ?station_name ;
    wgs84_pos:lat_lon ?lat_lon_pos.

  ?bike_id a gbfs:Bike ;
    wgs84_pos:location ?station .
}
WHERE {
  ?bike wgs84_pos:lat ?lat ;
        wgs84_pos:long ?lon .

  SERVICE <#srvpath/?source=#srcpath
      > {
    ?station gbfs:name ?name ;
    wgs84_pos:lat ?station_lat;
    wgs84_pos:long ?station_lon .
  }
  FILTER (
    ABS(?lat-?station_lat)<0.001 &&
    ABS(?lon-?station_lon)<0.001)
  BIND
    (CONCAT(str(?lat),",",str(?lon))
      as ?lat_lon_pos)
}
```

Listing 5: Example of federated SPARQL query.

as the source to be queried is structured. The service offers a SPARQL 1.1 protocol HTTP query API, that is also suitable to be used as endpoint for federated SPARQL queries.

Based on the original SPARQL 1.1 Protocol, we have derived a formal query API, and provided a formal design of the presented solution. We have moreover presented a proposal for an actual service architecture, based on the CARML non-RDF-to-RDF mapping engine.

Finally, we outlined our protocol implementation, and concluded with an evaluation of the prototype implementation. We moreover demonstrated the applicability of the service in the scope of federated SPARQL queries.

The presented implementation is published on Github under https://github.com/SmartMaaS/sparql-api-wrapper.

The current design and implementation so far neglect named graphs. How to include this concept in the presented service design is subject to future work.

So far, the service supports SPARQL SELECT, ASK and CONSTRUCT queries. As future work, we also plan to investigate how SPARQL UPDATE queries against a non-RDF endpoint may be used to modify a remote non-RDF dataset, given that the remote endpoint allows data modification.

We have discussed our solution with respect to use-cases from the domain of traffic and public transport. We see however, and plan to evaluate, a clear

```
<http://foo.bar/stations/10044540>
  a gbfs:Station,
    wgs84_pos:SpatialThing;
  gbfs:name "Juridicum";
  wgs84_pos:lat_lon
      "50.73009232899485,
      7.108277678489685" .

<http://foo.bar/bikes/44608> a gbfs:
    Bike;
  wgs84_pos:location
    <http://foo.bar/stations
        /10044540> .

<http://foo.bar/bikes/45448> a gbfs:
    Bike;
  wgs84_pos:location
    <http://foo.bar/stations
        /10044540> .

<http://foo.bar/bikes/45337> a gbfs:
    Bike;
  wgs84_pos:location
    <http://foo.bar/stations
        /10044540> .
```

Listing 6: Query result (excerpt) as returned by the federated SPARQL query example.

applicability in use-cases from industrial domains, as well as Smart City, Smart Grid, and Smart Living scenarios.

ACKNOWLEDGEMENTS

This work has been supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) (FZK 01MD18014 C) as part of the project Smart MaaS, and by the German Federal Ministry of Education and Research through the MOSAIK project (grant no. 01IS18070-C). The project Smart MaaS is part of the technology program "Smart Service Welt II", which is funded by the German Federal Ministry for Economic Affairs and Energy (BMWi).

REFERENCES

Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific american*, 284(5):28–37.

Bizer, C., Heath, T., and Berners-Lee, T. (2011). Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global.

Bizer, C. and Seaborne, A. (2004). D2RQ-treating non-RDF databases as virtual RDF graphs. In *Proceedings of the 3rd international semantic web con-*

ference (ISWC2004), volume 2004. Proceedings of ISWC2004.

Buil-Aranda, C., Arenas, M., Corcho, O., and Polleres, A. (2013). Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *Web Semantics: Science, Services and Agents on the World Wide Web*, 18(1):1 – 17. Special Section on the Semantic and Social Web.

Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., and Xiao, G. (2017). Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487.

Colpaert, P., Abelshausen, B., Andrés, J., Meléndez, R., and Delva, H. (2019). Republishing Open Street Map's roads as Linked Routable Tiles. In *European Semantic Web Conference*, pages 13—-17.

Colpaert, P., Verborgh, R., and Mannens, E. (2017). Public transit route planning through lightweight linked data interfaces. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10360 LNCS, pages 403–411.

Das, S., Sundara, S., and Cyganiak, R. (2012). R2RML: RDB to RDF Mapping Language. *W3C Recommendation*, (September 2012):1–34.

Dimou, A., Sande, M. V., Colpaert, P., Verborgh, R., Mannens, E., and Van De Walle, R. (2014a). RML: A generic language for integrated RDF mappings of heterogeneous data. In *CEUR Workshop Proceedings*, volume 1184.

Dimou, A., Sande, M. V., Slepicka, J., Szekely, P., Mannens, E., Knoblock, C., and Van De Walle, R. (2014b). Mapping Hierarchical Sources into RDF using the RML Mapping Language.

Dimou, A., Vander Sande, M., Colpaert, P., Mannens, E., and Van De Walle, R. (2013). Extending R2RML to a source-independent mapping language for RDF. In *CEUR Workshop Proceedings*, volume 1035, pages 237–240.

Gupta, S., Szekely, P., Knoblock, C. A., Goel, A., Taheriyan, M., and Muslea, M. (2012). Karma: A system for mapping structured sources into the semantic web. In *Extended Semantic Web Conference*, volume 7540, pages 430–434. Springer, Berlin, Heidelberg.

Harth, A., Knoblock, C. A., Stadtmüller, S., Studer, R., and Szekely, P. (2013). On-the-fly integration of static and dynamic linked data. In *CEUR Workshop Proceedings*, volume 1034.

Lefrançois, M., Zimmermann, A., and Bakerally, N. (2017). A SPARQL extension for generating RDF from heterogeneous formats. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10249 LNCS, pages 35–50.

Mayer, S., Verborgh, R., Kovatsch, M., and Mattern, F. (2016). Smart configuration of smart environments. *IEEE Transactions on Automation Science and Engineering*, 13(3):1247–1255.

Michel, F., Djimenou, L., Zucker, C. F., and Montagnat, J. (2017). *xR2RML: Relational and non-relational databases to RDF mapping language*. PhD thesis, CNRS.

Michel, F., Faron-Zucker, C., and Montagnat, J. (2016). A mapping-based method to query mongodb documents with sparql. In *International Conference on Database and Expert Systems Applications*, pages 52–67. Springer.

Michel, F., Zucker, C. F., Gandon, F., Fabien, G., and Faron-Zucker, C. (2018a). Bridging Web APIs and Linked Data with SPARQL Micro-Services. pages 187–191.

Michel, F., Zucker, C. F., Gandon, F., Fabien, G., and Faron-Zucker, C. (2018b). SPARQL Micro-Services: Lightweight Integration of Web APIs and Linked Data. Technical report.

Nallur, V., Elgammal, A., and Clarke, S. (2015). Smart Route Planning Using Open Data and Participatory Sensing. In *IFIP Advances in Information and Communication Technology*, volume 451, pages 91–100. Springer New York LLC.

Pérez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer.

Priyatna, F., Corcho, O., and Sequeda, J. (2014). Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph. In *WWW 2014 - Proceedings of the 23rd International Conference on World Wide Web*, pages 479–489.

Rodríguez-Muro, M. and Rezk, M. (2015). Efficient SPARQL-to-SQL with R2RML mappings. *Journal of Web Semantics*, 33:141–169.

Scharffe, F., Bihanic, L., Képéklian, G., and Atemezing (2012). Enabling Linked Data Publication with the Datalift Platform. *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Slepicka, J., Yin, C., Szekely, P., and Knoblock, C. A. (2015). KR2RML: An alternative interpretation of R2RML for heterogeneous sources. In *CEUR Workshop Proceedings*, volume 1426.

Verborgh, R., Steiner, T., Van Deursen, D., Van de Walle, R., and Vallés, J. G. (2011). Efficient runtime service discovery and consumption with hyperlinked restdesc. In *2011 7th International Conference on Next Generation Web Services Practices*, pages 373–379. IEEE.

Verborgh, R. and Vander Sande, M. (2020). The semantic web identity crisis: in search of the trivialities that never were. *Semantic Web*, (Preprint):1–9.