Beyond Classical SERVICE Clause in Federated SPARQL Queries: Leveraging the Full Potential of URI Parameters

Olivier Corby, Catherine Faron, Fabien Gandon, Damien Graux, Franck Michel

▶ **To cite this version:**

Olivier Corby, Catherine Faron, Fabien Gandon, Damien Graux, Franck Michel. Beyond Classical SERVICE Clause in Federated SPARQL Queries: Leveraging the Full Potential of URI Parameters. WEBIST 2021 - 17th International Conference on Web Information Systems and Technologies, Oct 2021, Online, Portugal. hal-03404125

HAL Id: hal-03404125
https://inria.hal.science/hal-03404125v1

Submitted on 26 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Beyond Classical SERVICE Clause in Federated SPARQL Queries: Leveraging the Full Potential of URI Parameters

Olivier Corby, Catherine Faron, Fabien Gandon, Damien Graux, and Franck Michel
Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis, France
firstname.lastname@inria.fr

Keywords: Semantic Web, URI parameters, federated querying, SPARQL, SPARQL federated query services, extended SPARQL-based services.

Abstract: Semantic Web applications integrating very different software and data sources have to face the heterogeneity of the quality and compliance to standards exhibited by each involved resource. In this paper we propose a uniform way of adapting and customizing the behavior of both the client and the server components of an `HTTP` exchange to cope with this diversity. We revisit the classical `SERVICE` clause in SPARQL federated queries in order to parameterize the behavior of both the SPARQL client and the SPARQL service. We propose mechanisms to identify and specify SPARQL federated query services and extended SPARQL-based services.

1 INTRODUCTION

Thanks to the W3C standards, the Web benefits from an unprecedented interoperability that propelled it beyond an "Information management proposal" (Berners-Lee, 1989) to a universal "application integration platform" (Fielding et al., 2017) and toward a platform linking all forms of intelligence (Berendt et al., 2021).

To parameterize the calls made between the applications on the Web, the RFC 3986 (Berners-Lee et al., 2005) defines the query component of a URI as indicated by the first question mark ("?" character) and terminated by a number sign ("#" character) or by the end of the URI. However, the exact structure of the query string is not standardized at the URI level. Although methods used to compose a query string may differ between websites, the original usage in HTML forms popularized the use of query strings composed of a series of field-value pairs where the field name and value are separated by an equals sign ("=") and the pairs are separated by the ampersand sign ("&") as in this example with two parameters `gname` and `fname` respectively set to "doe" and "john".

```
https://examp.le/search?gname=doe&fname=john
```

The query string of a URI is essentially perceived as a way to pass parameters to the server and to the logic invoked in producing a response. But the position we take in that paper is that, because they are at the frontier between the client and the servers of a Web application, URI parameters can be used to influence the behavior of both these components. Instead of resolving to adhoc hacks, we propose a uniform way, based on the URI query string, to adapt and customize the behavior of the client and the server components of an HTTP exchange. By relying on the standard URI parameters, the approach remains backward compatible as it is transparent for clients or servers that do not implement them and simply ignore them. Moreover, in the context of federated systems, for instance, the same software component may endorse the role of a client or a server in different interactions and this uniform way of parameterizing behaviours works seamlessly in that case too.

In particular, we propose to exploit the full potential of the query string of a URI in the context of Semantic Web applications (Gandon, 2018) where the communication between clients and servers is based on SPARQL, standing for SPARQL Protocol and RDF Query Language (Harris and Seaborne, 2013). It provides a fully declarative language for query and update operations (Allemang et al., 2020). Therefore, unlike with imperative programming languages, it is not possible to embed any instruction to tune the behavior of the SPARQL query processor that is evaluating a `SERVICE` clause, or to control the query plan that it will come up with. Yet, we may want to customize its behavior based on what we know about the remote service (quotas, supported features, etc.).

In this article, we report our return on experience on using URI parameters to parameterize the behavior

of both SPARQL clients and services. Our proposal is motivated by the fact that, although the W3C standards provide an excellent common ground for interoperability, in practice applications integrating very different software and data sources have to face the heterogeneity of the quality and compliance to standards exhibited by each involved resource. One application of great interest is to cope with heterogeneity in the context of federated querying. In particular, in real-world scenarios, a SPARQL query processor evaluating a federated query must be able, on one side, to parameterize the remote SPARQL services' behaviors (for which it is the client) according to their capabilities and specificity, and, on the other side, to adapt its own behavior to the limitations and characteristics of these remote services.

Nowadays, many national or international projects are facing such a situation, in particular any project targeting scientific data integration will have to address this heterogeneity, for example: the German DFG project *FAIR Data Infrastructure for Condensed-Matter Physics and the Chemical Physics of Solids* (FAIRmat[1]) aiming to integrate and make accessible and reusable the enormous amount of data on materials (basic and applied science and engineering) produced in recent years; the French ANR project *Data to Knowledge in Agronomy and Biodiversity* (D2KAB[2]) aiming to create a framework to turn agronomy and biodiversity data into an integrated, interoperable and open knowledge graph; the European ERIC project *Analysis and Experimentation on Ecosystems (AnaEE[3])* aiming to integrate all the steps of the scientific experimental methods, modelling and experimentation in order to understand the impact of change drivers on terrestrial and aquatic continental ecosystems across Europe; etc. A common challenge in the development of such research infrastructures (RI) is to provide a unique entry point to the databases provided by the project partners; these are most of the time hugely heterogeneous, not only regarding the data they store and their schemas, but also regarding the servers chosen to store and serve them and the practices adopted by the research teams in charge of configuring them and managing their creation and maintenance.

Thus, the main research question of this article is: *do URI parameters provide an effective way to improve and extend SPARQL-based interaction between Web clients and services?* It entails two specific subquestions : *can URI parameters allow SPARQL-based interactions that could not be done before?* and *can URI parameters make existing SPARQL-based interactions more efficient in time, space, etc.?*

To answer these questions, this article is organized as follows: Section 2 discusses related work on the parameterization of client/server exchanges in the context of SPARQL services. Section 3 presents the general principle of our approach to customize the behavior of SPARQL services and clients. The three next sections start from what is already possible to do in terms of parameterization with the standards, and gradually build on it to provide controls over the behaviors of both the client and the server components of a SPARQL exchange. Section 7 presents some experiments that we conducted to validate and demonstrate our approach. Section 8 draws some conclusions and gives directions for future work.

2 RELATED WORK

Nowadays, there are many SPARQL endpoints and most of them suffer from partial coverage of the SPARQL query language together with a lack of availability for a great majority of them, as reported by (Buil-Aranda et al., 2013). In addition, one recurring problem when dealing with SPARQL endpoints is the limitation of the server's resources allocated to each individual query, often leading to timeouts, errors if the queries were too complex, or even incomplete results. To circumvent these limitations some efforts have been made to set up alternative solutions to the "traditional" SPARQL endpoints such as the SaGe initiative by Minier *et al.* which relies on Web preemption (Minier et al., 2019). Others like C. Buil-Aranda *et al.* (Buil-Aranda et al., 2014) propose to split the query into pieces so as to respect the various quotas of the distant SPARQL server. Another approach tackles the challenge the other way around by providing specific interfaces, named *Linked Data Fragments* (Hartig et al., 2017) to access the datasets which can then be chosen by users depending on their needs. For example, in the Triple Pattern Fragments approach (Verborgh et al., 2016), the server only evaluates triple patterns. However, these *fragment* approaches generate a large number of subqueries and substantial data transfer.

Unfortunately, all these methods are applied after reaching these limitations. We aim at preventing these situations. We propose to consider that a wide set of URI parameters –to tune the server's behavior– could prevent these issues by providing SPARQL practitioners with the useful tools. Moreover, we also introduce URI parameters to be interpreted by the client too; which is, to the best of our knowledge, unprece-

[1] https://www.fair-di.eu/fairmat/proposal
[2] http://www.d2kab.org/
[3] https://www.anaee.eu/

dented. In particular, we focus our effort (and validate it) on the parameterization of federated querying using the SPARQL `SERVICE` clause[4].

Technically, SPARQL 1.1 Graph Store HTTP Protocol[5] defines a mean for updating and fetching RDF graph content from a Graph Store over HTTP in the REST style. For example the following:

```
DELETE /rdf-graph-store?graph=<graph_uri> HTTP/1.1
Host: examp.le
```

would be the equivalent of the following SPARQL 1.1. Update operation:

```
DROP GRAPH <graph_uri>
```

As a consequence, some SPARQL engines such as GraphDB[6], Fuseki[7] or BrightStarDB[8] rely on this to let the SPARQL practitioners slightly tune the queries with additional parameters. Typically, practitioners are *e.g.* able to select the return format of a query using `output=xml` to obtain RDF/XML. In addition, GraphDB also provides some additional features for *internal federation*[9] which unfortunately does not comply with the SPARQL 1.1 standard anymore, at the time of writing of this article, in June 2021. Other solutions like RDF4J[10] (Broekstra et al., 2002), Virtuoso[11] (Erling and Mikhailov, 2010), Anzo[12] or also BlazeGraph[13] extend the set of possible parameters. For example, RDF4J allows the practitioner to add `timeout` or `limit`; with `explain`, BlazeGraph enables a mode where the query results are extended with explanations of the query plan. When compared to these approaches, we notice that, while these SPARQL endpoint implementations support parameters, they remain specific to the server side and in particular the idea of using them in the context of SPARQL `SERVICE` clauses is not widespread yet.

Alternatively to using the `SERVICE` keyword to access additional endpoints, one may use federated SPARQL services which provide a unified access interface to a set of autonomous data sources. Their main objective is to transform a query posed on a federation of data sources into a query composed of the union of subqueries over individual data sources of the federation. In particular, Saleem *et al.* studied federated RDF query engines with web access interfaces (Saleem et al., 2016). So far, most of the efforts have been tackling challenges such as *source selection* or *query planning* and few implemented tools provide the practitioners with a way to configure the behaviour of the data sources' endpoints. For example, FedX (Schwarte et al., 2011) only provides `setMaxExecutionTime` to define the maximum time for the whole query.

Finally, regarding the technique we present about unifiying several endpoints under the same banner (see Section 6.1), it is worth mentioning that Comunica (Taelman et al., 2018) does also provide a similar concept of querying *several-at-once*. However, Comunica requires practitioners to enter all the wanted sources' URLs[14] from the beginning when our implementation allows the description of a set of endpoints through the use of a dedicated vocabulary. In addition, Comunica does not set up a process to configure the endpoints' behaviors as we propose.

3 GENERAL PRINCIPLE

We start with a reminder of the terms of the SPARQL 1.1 Protocol[15] that describes a means for *SPARQL protocol clients* to submit *SPARQL protocol operations* (either `query` or `update` operations) to a *SPARQL protocol service*. For brevity, from now on we will omit the *protocol* term, speaking simply of *SPARQL client*, *SPARQL operation* and *SPARQL service*. We will use the term *SPARQL query* to denote a SPARQL operation of type `query`. Furthermore, we will use the terms *SPARQL endpoint* and *SPARQL service* interchangeably. In the rest of the paper, we also use the term URL instead of URI because SPARQL endpoints are identified and accessed using URLs.

A SPARQL client submits a SPARQL operation over HTTP to a SPARQL service that handles the operation and sends the response back to the client. `query` operations may be submitted using either the HTTP `GET` or `POST` method, whereas `update` operations must be submitted using the `POST` method. When a SPARQL query is submitted using the `GET` method, the URL includes, among other possible parameters, the URL-encoded SPARQL query parameter. For instance, the following URL conveys a SPARQL query to a SPARQL service to get all its triples[16]:

```
http://e.g/sparql?query=select * where{?x ?p ?y}
```

We propose to use the standard mechanism of URL-

[4] SPARQL 1.1 Federated Query
[5] Graph Store HTTP Protocol
[6] GraphDB compliance with the Graph Store HTTP Protocol
[7] Jena Fuseki endpoint configuration
[8] BrightStarDB configuration
[9] GraphDB internal federation
[10] RDF4J repository queries
[11] Virtuoso documentation
[12] Anzo endpoint configuration
[13] BlazeGraph rest API
[14] Comunica endpoint over multiple sources
[15] https://www.w3.org/TR/sparql11-protocol/
[16] For readability, the SPARQL query is not URL-encoded.

encoded query string parameters as a common means to tune the behavior of SPARQL clients and servers involved in an HTTP exchange. Newly defined parameters may pertain to different aspects of the submitted SPARQL operation, such as query planning, authentication, output format, or execution traces. For instance, we can amend the previous example URL to instruct the service to provide some execution traces:

```
http://e.g/sparql?log_level=debug&
                  query=select * where{?x ?p ?y}
```

This use of parameters in the URL of an HTTP query is "natural" in the sense that the parameters are interpreted by the service being invoked. We call them "server-side" parameters and explore them in Section 4. We propose to complement this approach with a set of "client-side" parameters in the URL of an HTTP query, meant to tune the behavior of the SPARQL client that initiates the exchange. As far as we know, this is the first documented case of usage of URL parameters for tuning SPARQL clients and it is detailed in Section 5. The primary use case that we foresee is when the SPARQL client is processing a federated query. Within a federated SPARQL query, the `SERVICE` keyword instructs the query processor to invoke (a portion of) the SPARQL query against a remote SPARQL endpoint identified by its URL.[17] Our rationale here is to tune the behavior of the SPARQL federated query processor that takes the role of a SPARQL client with respect to the remote SPARQL services it federates, by adding query string parameters to the URL of the remote endpoint in a `SERVICE` clause. For example, if we know that a remote endpoint does not support the HTTP `POST` method, we can instruct the federated query processor to use only the `GET` method to communicate with it:

```
SERVICE <http://e.g/sparql?method=get>
  { SELECT * WHERE {?x ?p ?y} }
```

Other methods could be figured out to pass such parameters to the SPARQL federated query processor, in the form of SPARQL extensions, for instance using Java-like annotations (introduced with the '@' character). Yet, this would make the SPARQL query invalid for processors that do not support this extension. The main interest of using query string parameters in the URL of the `SERVICE` clause of a SPARQL federated query is that the query remains syntactically fully SPARQL-compliant.

Finally, in the continuation of this incremental enrichment of the SPARQL 1.1 Protocol, we propose to also leverage URL parameters as a means to control the behavior of what we call "extended SPARQL-based services". Such services are invoked with regular SPARQL query operations, but their behavior differs from regular SPARQL services in that their output may be different from regular SPARQL query results, or they may implement a custom service logic fulfilling specific needs. Typically, a SPARQL federated query processor can be such an extended SPARQL-based service: it takes as an input a SPARQL query, rewrites it into a federated query, communicates with remote SPARQL services to evaluate the query, then returns the SPARQL results. Although its interface complies with that of a SPARQL service, it implements a different logic and therefore may require additional parameters to properly tune its behavior. Another example is that of a service that evaluates an input SPARQL query, applies a custom transformation to the results (e.g. generates an HTML page), and returns the result of this transformation instead of the SPARQL query results. For instance, the following URL asks an extended and named SPARQL-based service to get all its triples and transforms the results in XML format into, by using an XSLT transformation.

```
http://e.g/transform/sparql?format=text/xml&
  transform=transfo.xsl&
  query=select * where{?x ?p ?y}
```

In the next three sections we will present in details the three patterns of URL parameters for SPARQL:

- Parameters that modify the behavior of a SPARQL service in Section 4;
- Parameters that modify the behavior of a SPARQL client, focusing on the case of federated querying using `SERVICE` clauses, in Section 5;
- Parameters that impact the behavior of a named extended SPARQL-based service in Section 6.

The list of all the URL parameters presented in the paper is given in Table 1. It is worth noticing 1) that this is not a closed list and that other URL parameters may be introduced following the same approach to answer new needs, and that 2) the same parameter can modify both the behavior of a service and the behavior of the client, for instance enforcing the format in which a result will be serialized on one side and parsed on the other side. This is notably the case every time a parameter concerns some aspect of the communication (HTTP method, representation format, etc.). As a result, such parameters will appear in both Section 4 and Section 5 where their impact on the behaviors will be explained respectively for the service side and the client side.

[17] https://www.w3.org/TR/sparql11-federated-query/

Table 1: Overview of URL parameters examples. Parameters in bold are part of the SPARQL protocol.

Parameter	Definition
default-graph-uri	specify the default named graph
named-graph-uri	specify the named graphs to use
format	specify expected result format
access	provide an access control key
log_level	require logs of the execution
method	force the HTTP method to use
header	add an HTTP header
limit	maximum number of results
timeout	maximum time for a response
binding	specify variable binding method
binding_focus	variables to pass in bindings
binding_skip	variables not to pass in bindings
binding_slice	size of bindings for each call
mode	generic behaviour customizing
accept	subset of federated services to be used
reject	subset of federated services not to be used
transform	apply a transformation to the data

4 URL PARAMETERS FOR SPARQL SERVICES

This first family of URL parameters is certainly the most natural, standard and common one. It aims at modifying the behavior of the SPARQL service being invoked: a SPARQL service receiving a parameterized HTTP request should adapt its behavior and the returned answer according to the query parameters. Yet it can be noticed that, while many SPARQL endpoint implementations support parameters (Virtuoso, BlazeGraph, etc.), the idea of using them in the context of SPARQL `SERVICE` clauses is not widespread yet. In this section, we first want to establish the interest of these parameters in the context of a `SERVICE` clause and then propose several new parameters for SPARQL `query` and `update` operations.

The `default-graph-uri` and the `named-graph-uri` parameters provide an immediate and standard example as they are defined in the SPARQL 1.1 Protocol to specify the dataset to be used by a SPARQL service in solving a query. In the context of a `SERVICE` clause, this ability addresses an important limitation of the SPARQL query language that does not support the declarative specification of the dataset for such a clause: `FROM` and `FROM NAMED` are not supported at the `SERVICE` clause level in SPARQL. Although this usage is neither documented in the standards nor widespread in online documentation, it is a perfect example from the standards of the impact of being able to parameterize the behavior of the invoked SPARQL service directly through the URL query string.

The `format` parameter was introduced in our implementation to enable us to specify the format of the SPARQL query result expected from the SPARQL service (the SPARQL `update` operation is not concerned). The motivating scenario is that, in a perfect Web, SPARQL services all support content negotiation and provide correctly formatted results, but in the World *Wild* Web, SPARQL services may have limitations or bugs in their content negotiation or serializer implementations. The value of a `format` parameter is a media type[18]. If set, this parameter overrides values provided by the optional HTTP `Accept` header, if any, and thus bypasses any content negotiation process. For instance, the following URL asks a SPARQL service to provide all its triples in the `JSON-LD` format.

```
http://e.g/sparql?format=application/ld+json
                 &query=select * where{?x ?p ?y}
```

The `access` parameter introduced in our implementation enables us to enforce a key-based access control policy on SPARQL services. The motivating scenario is that, in a perfect Web, SPARQL services should serve open data at will to all users, but in the World *Wild* Web, some private or critical SPARQL services need to limit their access to authorized users for security and privacy concerns or to avoid saturation. Typically, a SPARQL service running in protected mode will not process an HTTP `query` operation unless this request is parameterized with an authentication key that was delivered by the SPARQL service manager to the authorized users. For instance, the following URL can be used by an authorized user to submit a SPARQL query to a SPARQL service while passing the authentication key `a6h3fb58Fbd3`:

```
http://e.g/sparql?access=a6h3fb58Fbd3
                 &query=select * where{?x ?p ?y}
```

Note that other authentication and/or authorization mechanisms may be required. For instance, Section 5 describes how a `header` parameter can pass a security token via the standard `Authorization` HTTP header.

The `log_level` parameter was introduced in our implementation to enable us to instruct SPARQL services to turn on the generation of execution traces. The motivating scenario is that, in a perfect Web, SPARQL services should always receive SPARQL queries implementing the actual user need and serve the expected data, but in the World *Wild* Web, some unexpected or missing data may be delivered by a SPARQL service, and like for any program, execution traces are highly valuable to understand abnormal behavior, detect bugs in the service or modify the SPARQL query sent to it. An example URL containing a `log_level=debug` parameter is provided in

[18] https://www.iana.org/assignments/media-types/media-types.xhtml

Section 3. The execution traces are produced in the SPARQL service's logging system. Another implementation could choose to provide the client with such traces by means of the header's `link` field within XML or JSON SPARQL results.

5 URL PARAMETERS FOR SPARQL CLIENTS

In a perfect Web, all SPARQL services implement the full set of SPARQL recommendations, have no timeout nor limited number of results, and perfectly comply with serialization specifications. In the World *Wild* Web, a SPARQL service may support the HTTP `GET` method but not the `POST` method, or may be able to return a valid JSON document but an invalid XML document. Furthermore, public SPARQL endpoints often limit the maximum number of results with quotas, such that a client would need to add the `LIMIT` and `OFFSET` modifiers to its SPARQL query in order to incrementally get a complete result set. This is even more striking when federated querying. In the worst-case scenario, to evaluate the `SERVICE` clauses in a SPARQL federated query, the SPARQL query processor may need to adapt its behavior to the issues or limitations of each one of the SPARQL remote services that the query involves. Unfortunately, SPARQL does not allow specifying such a fine-tuning in a declarative manner.

Therefore, in this section we consider the case where URL parameters are not only meant for the SPARQL service being invoked but also, or rather, meant to modify the behavior of the SPARQL client sending an HTTP request to a SPARQL service or receiving its response. More precisely, in our implementation we have introduced several query string parameters in the URL of the `SERVICE` clause, that are interpreted by the SPARQL federated query processor (the client) to tune its behavior – and that may also be interpreted by the remote SPARQL service or simply be ignored.

The `method` parameter specifies the HTTP method (`GET` or `POST`) to be used by the federated query processor to submit a SPARQL query to a remote SPARQL service. For instance, the URL in the following `SERVICE` clause can be used to ask the federated query processor (the client) to use the HTTP `POST` method to submit its query to the remote SPARQL service – and to pass the authentication key `a6h3fb58Fbd3`. The remote SPARQL service may use or ignore any of the parameters, in particular it can ignore the `method` parameter but enforce the authentication key:

```
SERVICE <http://e.g/sparql?method=post&
                           access=a6h3fb58Fbd3>
    { ?x ?p ?y }
```

The `format` parameter specifies the media type of the query results returned by the SPARQL service. It may be used when the SPARQL service does not support content negotiation and returns results in a fixed format. It may also be the counterpart of the server-side `format` parameter described in Section 4, that skips the content negotiation process and forces the SPARQL service to return results in the specified media type. Accordingly, on the client side, this parameter instructs the SPARQL federated query processor to select an appropriate parser for the results of a remote service without considering any content negotiation.

The `log_level` parameter is the counterpart of the server-side `log_level` parameter presented in Section 4. When federated querying, it instructs the client SPARQL federated query processor to turn on the generation of execution traces. This is particularly handy when drafting a SPARQL federated query. Indeed, how a federated query processor rewrites each `SERVICE` clause, for instance by adding variable bindings, is implementation-dependent. The `log_level` parameter can help query debugging by providing insight into the strategy adopted by the federated query processor when rewriting a certain `SERVICE` clause.

The `header` parameter specifies an HTTP header to be sent by the client SPARQL federated query processor to a remote SPARQL service along with the SPARQL operation. Each parameter value, formatted as `name:value`, will be passed as a header of the HTTP request. For instance, let us consider the following `SERVICE` clause:

```
SERVICE <http://e.g/sparql?
        header=Authorization: Basic YvcGVuc2VzY>
    { ?s ?p ?o }
```

In this example, the value of parameter `header` is the HTTP `Authorization` request header with the `Basic` type and `YvcGVuc2VzY` for the credentials value. If the federated query processor uses the HTTP `POST` method to send the query to the remote SPARQL service, the HTTP request that it will send will look like this:

```
POST /sparql HTTP/1.1
Host: e.g
Content-Type: application/sparql-query
```

```
Authorization: Basic YvcGVuc2VzYW

SELECT * WHERE { ?s ?p ?o }
```

The `limit` parameter specifies a limit for the number of results returned by one service call. It instructs the client SPARQL federated query processor to add a `LIMIT` solution modifier to the query it submits to the remote SPARQL service. For example, the URL in the following `SERVICE` clause tells the query processor to query the remote service for only its 100 first results for the graph pattern in the clause.

```
SERVICE <http://e.g/sparql?limit=100> { ?s ?p ?o }
```

The `timeout` parameter specifies a time quota (in milliseconds) during which the client SPARQL federated query processor will wait for the remote SPARQL service to respond. Once this timeout expires without receiving a response, the federated query processor deems the query as failed. Example:

```
SERVICE <http://e.g/sparql?timeout=5000> {?s ?p ?o}
```

For simplicity, we consider the `connect` and `read` timeouts at once, however they could be distinguished by introducing two different `timeout` parameters.

Binding-related parameters. The query plan and strategy adopted by a SPARQL federated query processor to evaluate a SPARQL federated query is implementation-dependent. We introduced in our implementation a set of binding-related, client-side URL parameters to enable a SPARQL practitioner to leverage prior knowledge about a SPARQL service to help the client SPARQL federated query processor to come up with a more efficient query plan.

The `binding` parameter specifies the way the client SPARQL federated query processor should pass variable bindings to a remote SPARQL service. Bindings coming from the evaluation of some graph pattern in the federated query are passed to a remote SPARQL service in case the graph pattern it should evaluate shares in-scope[19] variables with the previously evaluated ones. The method used by a client SPARQL federated query processor to pass variable bindings to a remote SPARQL service is implementation-dependent. In a perfect Web, the "natural" way to pass bindings would be to use a `VALUES` block in the `SERVICE` clause. But, in the World *Wild* Web, some SPARQL services do not implement the `VALUES` clause. Therefore, for the sake of interoperability, by default our implementation uses filters to pass bindings, and we defined a `binding` parameter to choose among both possible ways. For instance, let us consider the following federated query:

```
SELECT * WHERE {
 ?s ?p ?o.
 SERVICE <http://e.g/sparql?> {?s ?q ?v. ?o ?q ?w}}
```

and the intermediate variable binding coming from the evaluation of triple pattern `?s ?p ?o`:

```
{(?s, s1), (?p, p1), (?o, o1)}
```

Adding parameter `binding=filter` to the URL of the `SERVICE` clause (the default behavior) will cause the federated query processor to generate the SPARQL code:

```
FILTER (?s = s1 && ?o = o1)
```

Conversely, adding parameter `binding=values` will generate the SPARQL code:

```
VALUES (?s ?o) { (s1 o1) }
```

The `binding_focus` and `binding_skip` parameters specify a subset of variables for which variable bindings should or should not be passed to the service. In the example below, the bindings consider variable `s` and not variable `o`.

```
SELECT * WHERE {
 ?s ?p ?o.
 SERVICE <http://e.g/sparql?binding_focus=s>
  { ?s ?q ?o }
}
```

Alternatively, in the example below the bindings do not consider `s` and consider `o` only.

```
SELECT * WHERE {
 ?s ?p ?o.
 SERVICE <http://e.g/sparql?binding_skip=s>
  { ?s ?q ?o }
}
```

The `binding_slice` parameter enables to specify the number of distinct variable bindings sent within one service call. The set of variable bindings is split into subsets of size less than or equal to this number. The service is called once for each such subset of variable bindings. In our implementation, the default value of `binding_slice` is 20. But one could, for instance, force one call per binding:

```
SELECT * WHERE {
 ?s ?p ?o.
 SERVICE <http://e.g/sparql?binding_slice=1>
  { ?s ?q ?o }
}
```

[19] https://www.w3.org/TR/sparql11-query/#variableScope

6 NAMED AND EXTENDED SPARQL-BASED SERVICES

In this section, we generalize on the server-side parameters approach of Section 4. We now propose to mint[20] URLs that **identify** and **specify** "extended SPARQL-based services", a kind of services invoked with regular SPARQL query operations, but whose behavior differs from that of regular SPARQL services in that their output may be different from regular SPARQL query results, or they may implement a custom service logic fulfilling specific needs. Their URLs are minted to unambiguously name these services. We put a specific stress on this naming concern because what is at stake here is not just to figure out the URL to access such a service. We mean to use those URLs as URIs, free from any optional query component, identifying these services as first-class resources, so that we can not only invoke these services, but also annotate them in RDF descriptions, and configure them with server-side URL parameters.

6.1 The common case of SPARQL Federated Query Services

As federated querying over multiple sources is a very common and important use case in applications such as scientific data integration, this section makes a focus on SPARQL federated query `services`. In Section 5 we considered the client role of a SPARQL federated query processor communicating with remote SPARQL services to evaluate a SPARQL federated query. Here we consider the service role of a SPARQL federated query processor, serving its results to SPARQL clients that ignore everything about the federation. We denote by "SPARQL federated query service" a set of approaches implementing the federation of several SPARQL services. These can be seen as extended SPARQL-based services insofar as they take as input a regular SPARQL query, but implement a service logic that differs from that of a regular SPARQL service.

We defined a succinct vocabulary to declare in RDF a SPARQL federated query service. For instance the following `RDF/Turtle` configuration describes the `X` SPARQL federated query service, identified by `http://e.g/X/sparql`, that federates three SPARQL services.

```
prefix st: <http://e.g/sparql-template/>
<http://e.g./X/sparql> a st:Federation ;
 st:definition (
  <http://a.b/blazegraph/Y/sparql>
  <https://c.d/fuseki/annotation/sparql>
  <http://i.j/repositories/sparql>
 ) .
```

[20] Minting a URI is the act of establishing the association between the URI and the resource it denotes (Allemang et al., 2020).

Below we present two examples of alternative implementations of such SPARQL federated query services that we developed and tested.

The first one receives a SPARQL query and sends it as is to the remote SPARQL services that it federates, performs the union of partial results and aggregates, if any, and returns the results. The URLs of this first implementation continue to use the `/sparql` that appears in the URL of standard SPARQL services to suggest that they do not change the query itself.

The second implementation is what is commonly referred to as a "federation engine". It starts by asking remote SPARQL services which properties and classes occur in their datasets and builds an index from the results. Then, using this index, it rewrites the SPARQL query it received with `SERVICE` clauses. Each clause is targeting a SPARQL service that, according to the index, may contain results for the triple patterns it specifies. To distinguish this second implementation from the first one, the endpoint URL is minted with `/federate` in its name:

```
http://e.g/X/federate?
    query=select * where {?x ?p ?y}
```

Similarly to what we demonstrated in Section 4, like for any SPARQL service, we can tune the behavior of such a named SPARQL federated query service by adding query string parameters to its URL:

The `mode=provenance` parameter enables to track the provenance of each result. For instance, the query below asks the D2KAB federated service to retrieve arboriculture sub-concepts from any of the federated endpoints.

```
SELECT DISTINCT * WHERE {
  [] skos:prefLabel "arboriculture"@fr;
     skos:narrower+ [ skos:prefLabel ?lbl ].
}
```

We can submit it with the URL below, where the URL-encoded query is ommitted for clarity:

```
http://corese.inria.fr/d2kab/sparql?
   mode=provenance&query=...
```

Table 2 shows a short subset of the results returned, where a pseudo-variable `server` is added to output the provenance of the variable bindings.

The `accept` and `reject` parameters specify a subset of the SPARQL services to be involved in the evaluation of the query submitted to the SPARQL federated query service. The value of these parameters is

Table 2: Example query result when using the `mode=provenance` parameter. Variable `server` provides the URL of the service that yielded this variable binding.

server	lbl
http://ontology.irstea.fr/bsv/sparql	vigne de cuve
http://ontology.inrae.fr/frenchcropusage/sparql	abricotier

a string pattern. The SPARQL federated query service should skip the services in its federation whose URL matches the pattern value of the `reject` parameter and/or consider those whose URL matches the pattern value of the `accept` parameter. For instance, assuming that the federation configuration names the endpoints of several DBpedia chapters, to answer the SPARQL query encoded in the following URL, the `X` SPARQL federated query service should consider any DBpedia chapter except the French one.

```
http://e.g/X/sparql?accept=dbpedia&reject=dbpedia.fr
                    &query=select * where {?x ?p ?y}
```

6.2 Extended SPARQL-based services for application integration

Generalizing on the previous example of SPARQL federated query services and data integration, we now report on a number of other extended services we experimented with in the more general context of Web application integration. The approach remains the same but is no longer focused on the problem of query federation: we mint URLs to unambiguously name a new service, to annotate it in RDF descriptions, to call it and to configure it with URL parameters.

RDF transformation services. The `transform` parameter enables us to turn a SPARQL service into an RDF transformation service. In an ideal Web, everyone would be using the same transformation and rendering techniques and pipelines. In the World *Wild* Web, the loose coupling of applications and the variety of formats and presentation solutions requires flexible transformation means. A parameterized RDF transformation service returns the result of a transformation that could come from any existing approach such as XSLT (Kay, 2021), FRESNEL (Pietriga et al., 2006) or STTL (Corby et al., 2015). For instance, the following parameterized service outputs the transformation of the query results into an HTML page.

```
http://e.g/X/sparql?transform=st:rdf2html
                    &query=select * where {?x ?p ?y}
```

Another example is the generation of widgets exploiting dimensions of the data (time, location, etc.), for instance, a map with plots when the SPARQL query results contain longitude and latitude values.

SPARQL query result annotation services. The `mode=link` parameter together with the `transform` parameter enable us to turn a SPARQL service into a SPARQL query result annotation service. The result of the transformation is stored in a document on the server, a URL is provided for this document and is returned as the value of the `link` element in the head of the SPARQL query result. For example, the following parameterized service will generate a map with location longitude and latitude, and a `link` mode that adds the URL of the generated map in the head of the SPARQL query result.

```
http://e.g/X/sparql?transform=st:map&mode=link
        &query=select * where {?x ?p ?y}
```

It is possible to generate several annotations by specifying several transformations, in which case the service generates several `link` elements.

SPIN services. The `mode=spin` parameter enables to turn a SPARQL service into a SPARQL2SPIN service that generates the SPIN format (Knublauch et al., 2011) of the SPARQL query sent to it. For example, the following parameterized service

```
http://e.g/X/sparql?mode=spin
        &query=select * where {?x ?p ?y}
```

will output the following RDF/Turtle SPIN graph:

```
@prefix sp: <http://spinrdf.org/sp#> .
[a sp:Select ;
  sp:star true ;
  sp:where ([sp:object [sp:varName "y"] ;
    sp:predicate [sp:varName "p"] ;
    sp:subject [sp:varName "x"]])] .
```

SHACL services. In a perfect Web, the data sources we consume would be complete and would meet the level of quality we need. In the World *Wild* Web, data sources do not always match our criteria and we need means to check if the constraints we have are met. The `mode=shacl` parameter used in combination with the `uri` parameter enables to turn a SPARQL service into a SHACL service that evaluate the shapes in the SHACL document, the URL of which is given as value of parameter `uri`, on the RDF graph it serves. Moreover, the query passed as parameter in the URL is executed on the SHACL validation report graph. For example, the following parameterized service will output the conformity boolean value.

```
http://e.g/X/sparql?mode=shacl&uri=shape.rdf
    &query=select * where {?report sh:conforms ?b }
```

Service Definition Specifying a list of parameter values to define extended SPARQL-based services may be cumbersome for some developers and one

may want to synthesize a specific list of parameter values within a specific mode. For this purpose, we defined a small vocabulary to enable the user to define in RDF new modes as the combination of several parameters that can be used to parameterize a service just like any predefined mode. For example, the following RDF description defines a `map` mode as the combination of the `st:map` transformation that generates a map with location longitude and latitude, and a `link` mode that adds the URL of the generated map in the head of the SPARQL query result.

```
[] st:mode "map" ;
st:param (("mode" "link")("transform" st:map)) .
```

Developers can then use such defined modes to define an extended service. For example, the following parameterized service is equivalent to the example service illustrating the annotation services.

```
http://e.g/X/sparql?mode=map
        &query=select * where {?x ?p ?y}
```

A generic mode, named "*", enables us to specify parameters that are shared by all the SPARQL-based services offered at a given SPARQL service. For example, the following RDF description states that all the services will run in `debug` mode.

```
[] st:mode "*" ;
st:param (("mode" "debug")) .
```

Services can also be described to associate them a parameter setting, so that the URL that should be used to invoke them will not have any query string parameter. For example, the following RDF description defines a service as parameterized by the (user-defined) `map` mode:

```
<http://e.g/map/sparql> st:param (("mode" "map")) .
```

Once defined like this it can be invoked without any other parameter than the standard `query` one:

```
http://e.g/map/sparql?query=select * where {?x ?p ?y}
```

7 EXPERIMENTS & EVALUATION

The URL query string parameters presented in the previous sections have been implemented in the Corese Semantic Web factory.[21] In this section, we demonstrate their interest in the context of two example queries that originate from on-going scientific data integration projects. In particular, we show the impact of this parameterization against baseline queries solving the same issues in standard conditions.

[21] https://project.inria.fr/corese/

7.1 Binding variables: `FILTER`/`VALUES`

The following query searches the URIs of some spatial entities in a local dataset, and tries to fetch additional information about each of them from Wikidata using a `SERVICE` clause.

```
SELECT ?e ?label WHERE {
  [] dct:spatial ?e.
  SERVICE <https://query.wikidata.org/sparql> {
    ?e rdfs:label ?label.
    FILTER (lang(?label) = "en")
}}
```

When a SPARQL engine evaluates this query, a possible strategy is to first retrieve the values of variable `e` (91 distinct values in our case), then pass them to the `SERVICE` clause as variable bindings. The SPARQL federated query recommendation does not specify the way to pass such bindings. A possible way is to pass them using a filter, i.e. the following query is submitted to Wikidata:

```
SELECT * WHERE {
  ?e rdfs:label ?label.
  FILTER (lang(?label) = "en")
  FILTER ((?e=wd:Q6730) || (?e=wd:Q12589) || ...)
}}
```

Unfortunately, Wikidata systematically times out when evaluating such a query.[22] However, we can work around this issue using the `binding` URL parameter described in Section 5. When we add it to the Wikidata URL as depicted below,

```
<https://query.wikidata.org/sparql?binding=values>
```

we instruct our SPARQL engine to use a `VALUES` block instead of a `FILTER`. The rewritten query sent to Wikidata now completes in less than one second:

```
SELECT * WHERE {
  VALUES (?entity) { (wd:Q6730) (wd:Q12589) ... }
  ?e rdfs:label ?label.
  FILTER (lang(?label) = "en")
}}
```

7.2 Slicing variables bindings

In the same on-going projects, we identified a second use case that involves the following SPARQL federated query to match resources from two RDF sources using their names:

```
SELECT * WHERE {
  SERVICE <http://e1.g/sparql> {
      [] schema:name ?name.
  }
  SERVICE <http://e2.g/sparql?binding=values> {
    SELECT distinct ?name ?fullname WHERE {
```

[22] The query only completes when there is one single value for `?e` in the `FILTER` clause. With two or more values, Wikidata times out.

```
      VALUES ?name {undef}
      [] rdfs:label ?fullname.
      FILTER(strstarts(?fullname, ?name))
}}}
```

In the first SERVICE clause, variable ?name is matched with 166 values. In the second SERVICE clause, variable ?fullname is matched with over 600,000 values. The second SERVICE clause cannot retrieve values for the ?name variable which is only used in the FILTER. Therefore, our implementation will start with the evaluation of the first SERVICE clause, and then pass the values of variable ?name as bindings to the second SERVICE clause. Furthermore, building on the experience of the use case described in Section 7.1, we added the parameter binding=values to perform the binding using a VALUES block rather than a FILTER.

The join between the two SERVICE clauses must be done on the ?name variable. Note however that, in the second SERVICE clause, ?name is not directly used as the term of a triple pattern but instead used in a string comparison (strstarts). This makes the query optimization much harder and, consequently, during our experimentations, the Virtuoso OS server at http://e2.g/sparql took in the order of 45 minutes to return the results.

We made further experiments to assess the number of values that can be passed in the VALUES block while keeping the query processing time below one minute. With one value, Virtuoso consistently took between 4s and 8s to respond. With two values, it took between 2min 30s and 3min. The more values, the longer it took to respond. Therefore, we added parameter binding_slice=1 to pass the values of variable ?name one by one. The URL of the second SERVICE clause is as follows:

```
<http://e2.g/sparql?binding=values&binding_slice=1>
```

The whole query processing completed in approximately 21 minutes, which is roughly half of the time it took to process the query without the binding_slice. The improvement may be deemed modest. It is however important to remind that with a public SPARQL endpoint, where the time quota is typically in the order of one minute, the initial query would never complete, whereas, with our method, it will take time but will complete eventually.

Without the URL query string parameters introduced in this paper and demonstrated in this section, it is hardly possible to obtain the results solely within SPARQL. Instead, the SPARQL practitioner would have to code a hack in another language to circumvent the limitations. As a comparison, we developed a Python application to achieve the same goal. Based on the SPARQLWrapper library, the program evaluates the first SERVICE clause, stores the temporary results in a DataFrame, and submits the second SERVICE clause one value at a time. This does exactly the same that what we achieve declaratively with just one parameter: binding_slice. While the SPARQL query is 10 lines long, the Python program is approximately 100 lines long. Besides, both solutions complete in a similar time since they submit the same number of queries.

Ultimately, this example also shows that beyond the federated querying use cases we used to demonstrate the interest of URL parameters in SPARQL, this technique is also a way to provide customized proxies masking the limitations and specificities of SPARQL service implementations and supporting the loose-coupling that make the Web an attractive application integration platform (Fielding et al., 2017).

8 CONCLUSION

In a perfect Web, following standards would be enough. In the World *Wild* Web, the ability to tune, customize, parameterize the applications may make the difference between a running application and a frozen one. In this paper, we proposed a uniform way of adapting and customizing the behavior of both the client and the server components of an HTTP exchange to cope with the heterogeneity of implementations we find in Web applications. More specifically we designed and experimented the use of URL parameters to parameterize the behaviors of both the SPARQL client and the SPARQL service, in particular in the context of SERVICE clauses used in SPARQL federated queries. We believe that this method has the potential to make federated querying in SPARQL more flexible, actionable, and suited to World *Wild* Web contexts, far beyond the examples that are generally demonstrated in ideal, controlled environments.

Another interesting general use case we did not develop here is the debugging of federated queries in the case where the query result is empty. It may be interesting to provide parameters such as mode=explain to obtain additional data such as intermediate query results that can be provided using linked results (mode=link). Hence, URL parameters could be used to tune a debugger.

In our future work, we also intend to study the relation of the extension we proposed with other parts of the SPARQL standard such as the SPARQL Service Description that "provides a mechanism by which a client or end user can discover information about the SPARQL service" (Williams, 2013).

In the longer term, the annotation technique in-

troduced by the `mode=link` parameter holds the huge potential to turn what are essentially one-off queries into entry points of a hypermedia network of queries. For instance, by providing links to reformulated or relaxed queries, expanded queries, suggested follow-up queries, etc., we would allow non SPARQL clients with classical Web client capabilities to still navigate and discover the content of endpoints as a classical Web hypermedia space just by applying the "follow your nose" principle.

ACKNOWLEDGEMENTS

This work was partially supported by the French National Research Agency under grant ANR-18-CE23-0017 (D2KAB project).

REFERENCES

Allemang, D., Hendler, J. A., and Gandon, F. (2020). *Semantic Web for the Working Ontologist*. ACM Books.

Berendt, B., Gandon, F., Halford, S., Hall, W., Hendler, J., Kinder-Kurlanda, K. E., Ntoutsi, E., and Staab, S. (2021). Web Futures: Inclusive, Intelligent, Sustainable The 2020 Manifesto for Web Science. *Dagstuhl Manifestos*.

Berners-Lee, T., Fielding, R. T., and Masinter, L. (2005). Uniform resource identifier (URI): Generic syntax. STD 66, RFC Editor. http://www.rfc-editor.org/rfc/rfc3986.txt.

Berners-Lee, T. J. (1989). Information management: A proposal. Technical report, CERN.

Broekstra, J., Kampman, A., and Van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying RDF and RDF schema. In *International semantic web conference*, pages 54–68. Springer.

Buil-Aranda, C., Hogan, A., Umbrich, J., and Vandenbussche, P.-Y. (2013). SPARQL web-querying infrastructure: Ready for action? In *International Semantic Web Conference*, pages 277–293. Springer.

Buil-Aranda, C., Polleres, A., and Umbrich, J. (2014). Strategies for executing federated queries in SPARQL 1.1. In *International Semantic Web Conference*, pages 390–405. Springer.

Corby, O., Faron-Zucker, C., and Gandon, F. (2015). A Generic RDF Transformation Software and its Application to an Online Translation Service for Common Languages of Linked Data. In *Proc. 14th International Semantic Web Conference, ISWC*, Bethlehem, Pennsylvania, USA.

Erling, O. and Mikhailov, I. (2010). Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*, pages 501–519. Springer.

Fielding, R. T., Taylor, R. N., Erenkrantz, J. R., Gorlick, M. M., Whitehead, J., Khare, R., and Oreizy, P. (2017). Reflections on the rest architectural style and" principled design of the modern web architecture"(impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 4–14.

Gandon, F. (2018). A Survey of the First 20 Years of Research on Semantic Web and Linked Data. *Revue des Sciences et Technologies de l'Information - Série ISI : Ingénierie des Systèmes d'Information*.

Harris, S. and Seaborne, A. (2013). SPARQL 1.1 Query Language. Recommendation, W3C. http://www.w3.org/TR/sparql11-query/.

Hartig, O., Letter, I., and Pérez, J. (2017). A formal framework for comparing linked data fragments. In *International semantic web conference*, pages 364–382. Springer.

Kay, M. (2021). XSL transformations (XSLT) version 2.0 (second edition). W3C recommendation, W3C. https://www.w3.org/TR/2021/REC-xslt20-20210330/.

Knublauch, H., Hendler, J. A., and Idehen, K. (2011). SPIN-overview and motivation. W3C member submission, W3C. https://www.w3.org/Submission/spin-overview/.

Minier, T., Skaf-Molli, H., and Molli, P. (2019). SaGe: Web preemption for public SPARQL query services. In *The World Wide Web Conference*, pages 1268–1278.

Pietriga, E., Bizer, C., Karger, D., and Lee, R. (2006). Fresnel: A browser-independent presentation vocabulary for rdf. In *International Semantic Web Conference*, pages 158–171. Springer.

Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., and Ngonga Ngomo, A.-C. (2016). A fine-grained evaluation of SPARQL endpoint federation systems. *Semantic Web*, 7(5):493–518.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: Optimization techniques for federated query processing on linked data. In *International semantic web conference*, pages 601–616. Springer.

Taelman, R., Van Herwegen, J., Vander Sande, M., and Verborgh, R. (2018). Comunica: a modular SPARQL query engine for the Web. In *International Semantic Web Conference*, pages 239–255. Springer.

Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., and Colpaert, P. (2016). Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37:184–206.

Williams, G. (2013). SPARQL 1.1 service description. W3C recommendation, W3C. https://www.w3.org/TR/2013/REC-sparql11-service-description-20130321/.