Querying the Web of Data with SPARQL-LD

Pavlos Fafalios, Thanos Yannakis, and Yannis Tzitzikas

Computer Science Department, University of Crete, Greece, and
Institute of Computer Science, FORTH-ICS, Greece
{fafalios,yannakis,tzitzik}@ics.forth.gr

Abstract. A constantly increasing number of data providers publish their data on the Web in the RDF format as Linked Data. SPARQL is the standard query language for retrieving and manipulating RDF data. However, the majority of SPARQL implementations requires the data to be available in advance (in main memory or in a repository), not exploiting thereby the real-time and dynamic nature of Linked Data. In this paper we present `SPARQL-LD`, an extension of SPARQL 1.1 Federated Query that allows to directly fetch and query RDF data from any Web source. Using `SPARQL-LD`, one can even query a dataset coming from the partial results of a query (i.e., discovered at query execution time), or RDF data that is dynamically created by Web Services. Such a functionality motivates Web publishers to adopt the Linked Data principles and enrich their digital contents and services with RDF, since their data is made directly accessible and exploitable via SPARQL (without needing to set up and maintain an endpoint). In this paper, we showcase the benefits offered by `SPARQL-LD` through an example related to the Europeana digital library, we report experimental results that demonstrate the feasibility of `SPARQL-LD`, and we introduce optimizations that improve its efficiency.

1 Introduction

While more and more structured data are published on the Web following the Linked Data principles [6], an important question is how one can efficiently access and query this constantly increasing body of knowledge. SPARQL [4] is the standard query language for retrieving and manipulating RDF data. However, the majority of SPARQL implementations requires the data to be available in advance, i.e., to exist in main memory or in a RDF repository accessible through a SPARQL endpoint. Nonetheless, Linked Data exists in the Web in various forms; even an HTML Web page can contain RDF data through RDFa [3], or RDF data may be dynamically created by Web Services.

In this paper we present `SPARQL-LD`, an extension (actually a generalization) of SPARQL 1.1 Federated Query [5] that allows to *directly* and *flexibly* exploit this wealth of data. `SPARQL-LD` extends the applicability of the `SERVICE` operator enabling to query any HTTP Web source containing RDF data. This extension does not require the named graphs to have been declared, thus one can even fetch and query a dataset returned by a portion of the query (i.e., whose URI is derived at query execution time).

Such a functionality can motivate Web publishers to enrich their documents and digital libraries with RDF since it makes their data directly accessible via SPARQL without needing to set up and maintain an endpoint (e.g., they can just publish RDF dumps). Actually, *availability* is the main bottleneck towards the success of the Semantic Web as a reliable technology. Buil-Aranda et al. [8] tested 427 public endpoints and found that their performance can vary by up to 3-4 orders of magnitude, while only 32.2% of public endpoints can be expected to have monthly uptimes of 99-100%. Therefore, it may be more reliable to directly retrieve the triples of a dereferenceable URI than retrieving the same triples by invoking a query against a remote endpoint (considering of course that the query requirements are satisfied).

Fig. 1 shows a query that can be answered by `SPARQL-LD`. The query first accesses Europeana's [13, 15] SPARQL endpoint[1] for retrieving artists of works related to Renaissance (lines 2-3). Then, by querying the dereferenceable URI of each artist, the query retrieves and shows a description (in English) and an image of only those of Mannerist style (lines 4-6). Note that Europeana does not contain information about artist styles. Notice also that the artist URIs are derived at query execution time. One could also integrate in the same query data from any Web resource or Web Service that offers its data in a standard RDF format. As an example, consider that an online bookstore service exports its search results in RDF. Using `SPARQL-LD` one can directly access this service through SPARQL and find books about the artists returned by the two `SERVICE` patterns in the query of Fig. 1. Likewise, in the same query one could exploit a video service and find links of YouTube videos related to some of the artists.

Consequently, the functionality offered by `SPARQL-LD` can overcome the limitations of digital libraries (and information sources in general) related to information integration, enrichment and exploitation.

```
1 SELECT DISTINCT ?creator ?descr ?photo WHERE {
2   SERVICE <http://europeana.ontotext.com/sparql> {
3     ?work dc:subject dbr:Renaissance ; dc:creator ?creator }
4   SERVICE ?creator {
5     ?creator dct:subject dbc:Mannerist_painters ;
6              dbo:abstract ?descr ; foaf:depiction ?photo FILTER(lang(?descr)="en") } }
```

Fig. 1: An example of a SPARQL query that can be answered by `SPARQL-LD`.

`SPARQL-LD` was first demonstrated in a short (demo) paper [10]. With respect to that paper, in this paper: i) we provide examples that illustrate the benefits offered by `SPARQL-LD`, ii) we identify factors that affect efficiency and propose optimizations, and iii) we extensively evaluate the efficiency of `SPARQL-LD` and the effect of the proposed optimization techniques. In addition, this paper provides a more detailed related work. The rest of this paper is organized as follows: §2 discusses related work, §3 introduces `SPARQL-LD`, §4 details optimization techniques, §5 presents evaluation results, and finally §6 concludes the paper.

[1] `http://europeana.ontotext.com/sparql`

2 Related Work

The approach that we propose is considered a method to execute queries over the Web of Linked Data. Such approaches can be classified in three main categories: *query federation*, *data centralization*, and *link traversal*.

The idea of *query federation* is to provide integrated access to distributed sources on the Web. For example, the systems DARQ [23] and `SemWIQ` [17] provide access to distributed RDF data sources using a mediator service that transparently distributes the execution of queries to multiple SPARQL services. Given the need to address query federation, in 2013 the SPARQL W3C working group proposed a query federation extension for SPARQL 1.1 [5]. Buil-Aranda et al. [7] describe the syntax of that extension and formalize its semantics.

The idea of *data centralization* is to provide a query service over a collection of data copied (and probably transformed) from different sources on the Web. Such a collection is usually called "Warehouse". There are *domain independent* warehouses like SWSE [14], but also *domain specific* like the MarineTLO-based Warehouse [25]. In the same category falls the case of *digital libraries* containing descriptions and metadata about digital objects collected from multiple content providers (like Europeana [15, 22]). Although such approaches require the data to exist in a single repository, they can significantly benefit from the functionality offered by `SPARQL-LD`. For instance, a query service over such a repository can support `SPARQL-LD` and offer the ability to also integrate (during query execution) data coming from online RDF sources (like in the example of Fig. 1).

Link traversal approaches exploit the Linked Data principles for discovering data related to URIs given in the query. For instance, the work in [11] discovers data that might be relevant for answering a query by following RDF links between data sources based on URIs in the query and in partial results. `Diamond` [19] is a similar in spirit query engine to evaluate SPARQL queries on distributed RDF data where, as a query is being evaluated, additional Linked Data can be identified by exploiting dereferenceable URIs. Finally, LDQL [12] is a declarative language to query Linked Data which is also based on link traversal. LDQL separates query components for selecting query-relevant regions of Linked Data, from components for specifying the query result.

`SPARQL-LD` actually *complements* the aforementioned approaches on query federation, data centralization and link traversal; it can be used in combination to such approaches. Works that focus on optimizing the execution of SPARQL federated queries, and that can be also applied in our case, are discussed in §4.

3 SPARQL-LD: Functionality and Examples

Motivation. Although the majority of SPARQL implementations requires the data to be available in advance (in main memory or in a repository), the specification of SPARQL allows to directly query a RDF dataset accessible on the Web (in a standard format) and identifiable by an URI through the operators `FROM`/`FROM NAMED` and `GRAPH`. However, this has an important limitation: it requires knowing *in advance* the URI of the dataset and having declared it in the

FROM NAMED clause. Thus, a URI coming from partial results (that get bound after executing an initial query fragment) cannot be used in the GRAPH operator as the dataset to run a portion of the query. Furthermore, although RDFa [3] and JSON-LD [1] are W3C standards that are exploited by an ever-increasing number of publishers, we have not managed to find a SPARQL implementation that can directly query such RDF data. In addition, using the SERVICE operator of SPARQL 1.1 Federated Query [5], we can invoke a portion of a query against a remote RDF repository. However, SERVICE requires the URI to be the address of a SPARQL endpoint, thus one cannot exploit this operator for querying RDF data accessible on the Web but not available through an endpoint.

Extended SERVICE definition. The SPARQL 1.1's SERVICE operator (SERVICE a P) is defined (in [7]) as a graph pattern P evaluated in the SPARQL endpoint specified by the URI a, while (SERVICE $?X$ P) is defined by assigning to the variable $?X$ all the URIs (of endpoints) coming from partial results, i.e. that get bound after executing an initial query fragment. The idea behind SPARQL-LD is to enable the evaluation of a graph pattern P not absolutely in a SPARQL endpoint a, but generally in a RDF graph G_r specified by a Web Resource r. Thus, now a URI given to the SERVICE operator can also be the dereferenceable URI of a resource, the Web page of an entity (e.g., of a person), an ontology (OWL), Turtle, or N3 file, etc. In case the URI is not the address of a SPARQL endpoint, the RDF data that may exist in the resource are fetched at real-time and queried for the graph pattern P.

SPARQL-LD is a generalization of SPARQL in the sense that every query that can be answered by the original SPARQL can be also answered by SPARQL-LD. Specifically, if the URI given to the SERVICE operator corresponds to a SPARQL endpoint, then it works exactly as the original SPARQL (the remote endpoint evaluates the query and returns the result). Otherwise, instead of returning an error (and no bindings), it tries to fetch and query the triples that may exist in the given resource.

Implementation. SPARQL-LD has been implemented using Apache Jena [2]. Jena is an open source Java framework for building Semantic Web applications. Specifically, we have extended Jena 2.13 ARQ component. ARQ is a query engine for Jena that supports SPARQL 1.1. The implementation is available as open source[2]. An endpoint that supports SPARQL-LD is publicly available[3].

The implementation can be described through the following process: we first check if the URI corresponds to a SPARQL endpoint by submitting the ASK query "ASK {?x ?y ?z}". In case we get a valid answer, we continue just like the default query federation approach, i.e. the corresponding graph pattern (query) is submitted to the endpoint. In case we do not get a valid answer, it means that the URI is not the address of an endpoint. Then, we read the *content type* header field of the URI by opening an HTTP connection and setting the value application/rdf+xml to the ACCEPT request property (we do that for handling also the case of RDFa). Now, according to the returned content type, we fetch

[2] https://github.com/fafalios/sparql-ld

[3] http://users.ics.forth.gr/~fafalios/sparql-ld-endpoint

and query the corresponding triples. For the case of HTML Web pages (the content type is `text/html` or `application/xhtml+xml`), we try to fetch and query the RDF triples that may be embedded in the Web page as RDFa. If the Web page does not contain any RDF data, the query returns no bindings. For reading possible RDF triples in a Web page, we exploit the `Semargl` framework (`https://github.com/levkhomich/semargl`) which also offers an integration with Jena. The implementation allows also reading and querying JSON-LD files.

Query Examples. Here we give two example queries that demonstrate the functionality offered by `SPARQL-LD`. More examples are available at the endpoint given in Footnote 3.

Querying dynamically-created RDF data. `X-Link` [9] is a Linked Data-based Named Entity Extraction (NEE) framework which can export the result of the NEE process in RDF using the Open NEE model [9]. An `X-Link` Web service configured for the artistic domain is publicly available at `http://83.212.107.202/x-link-art`. This service can identify names of several types of entities in a given Web document and link them to Web resources (URIs). For instance, we can request to perform NEE with *painters* and *countries* as the entities of interest at the Web page "`https://en.wikipedia.org/wiki/Mannerism`" and get the results in the default RDF/XML format, with the following request:
`http://83.212.107.202/x-link-art/api?categories=painter;country&url=https://en.wikipedia.org/wiki/Mannerism`

Using the proposed extension, one can exploit the APIs of such services directly through SPARQL. For instance, Fig. 2 depicts a query that *parameterizes* and *calls* the above annotation service *at query execution time* (the namespaces have been omitted to save space). The query first retrieves Web pages related to *Mona Lisa* by querying its dereferenceable DBpedia URI (lines 2-3). Then, it calls the `X-Link` service for identifying names of *painters* and *countries* in the retrieved Web pages (lines 4-6), and for each detected entity the query retrieves (and shows) its name, its category and its number of occurrences in the Web pages (lines 7-9). Finally, the entities are ordered by the number of occurrences in descending order (line 10).

```
 1 SELECT DISTINCT ?detectedEntity ?categoryName (count(?position) as ?NumOfOccurrences) WHERE {
 2   SERVICE <http://dbpedia.org/resource/Mona_Lisa> {
 3     dbr:Mona_Lisa dbo:wikiPageExternalLink ?page }
 4   VALUES ?templ { <http://83.212.107.202/x-link-art/api?categories=painter;country&url=PAGE> }
 5   BIND(REPLACE(str(?templ), "PAGE", str(?page), "i") as ?x) BIND(URI(?x) as ?service)
 6   SERVICE ?service {
 7     ?annot oa:hasBody ?ent .
 8     ?ent oae:regardsEntityName ?detectedEntity ; oae:position ?position .
 9     ?ent oae:belongsTo ?category . ?category rdfs:label ?categoryName }
10 } GROUP BY ?detectedEntity ?categoryName ORDER BY DESC(?NumOfOccurrences)
```

Fig. 2: Example of a SPARQL query that parameterizes and calls an annotation service at query execution time.

Querying RDFa. The first author of this paper has enriched his personal Web page (`http://users.ics.forth.gr/~fafalios`) with RDFa describing information about his publications. Using the proposed extension, such RDF data embedded in Web pages is directly available through SPARQL. For example, the query in Fig. 3 returns all his co-authors together with their publications. The list of co-authors is obtained by querying the RDF data that is embedded in his personal Web page (lines 2-4), while their names and publications are obtained by querying the dereferenceable URI of each co-author (lines 5-7). Notice that the author URIs are derived at *query execution time.*

```
1 SELECT DISTINCT ?authorName ?paper WHERE {
2   SERVICE <http://users.ics.forth.gr/~fafalios/> {
3     ?p <http://purl.org/dc/terms/creator> ?author
4     FILTER(?author != <http://dblp.l3s.de/d2r/resource/authors/Pavlos_Fafalios>) }
5   SERVICE ?author {
6     ?author <http://xmlns.com/foaf/0.1/name> ?authorName .
7     ?paper <http://purl.org/dc/elements/1.1/creator> ?author } }
```

Fig. 3: Example of a SPARQL query that reads and queries RDF data embedded in a Web page (as RDFa) at query execution time.

4 Optimizations

Several approaches have been proposed in the literature that aim at optimizing the execution of SPARQL federated queries, e.g., by reordering triple patterns based on cost estimation [24], by optimizing the evaluation of the `OPTIONAL` operator [7] (which is the most costly operator in SPARQL), by planning `SERVICE` queries against multiple endpoints based on the expected number of returned triples [20], or by parallelizing the execution of joins and union operators [24]. In addition, several caching approaches have been proposed that aim to improve the performance on answering SPARQL queries [16, 18]. Obviously, all of them are beneficial for `SPARQL-LD` too. However, `SPARQL-LD` has the following extra requirements (points that need attention) that are not satisfied by existing works: (a) to *reduce the* `ASK` *queries that check whether a URI corresponds or not to a SPARQL endpoint*, and (b) to *avoid fetching remote resources that have been already fetched in the context of a single query execution.*

Below we describe optimizations that cope with the above requirements.

4.1 Index of Known SPARQL Endpoints

We have seen that, compared to the original `SERVICE` operator, the only additional cost is the time to run an `ASK` query (as we will see in §5, this cost is about 200 ms in average). To eliminate this cost, we can keep a small index with the URIs of known endpoints (like DBpedia's and Europeana's) as well as the URIs of endpoints that have been already checked. Thereby, if the `SERVICE` URI exists in the index, the query is directly forwarded to the endpoint, otherwise an `ASK` query is first submitted.

For example, consider the query of Fig. 4. The query first retrieves Greek painters from the dereferenceable URI of the corresponding DBpedia category (lines 2-3), and then it queries Europeana' SPARQL endpoint for retrieving works of these painters (lines 4-6). However, if the number of painter URIs returned by the first `SERVICE` invocation is n, the query will call the remote endpoint n times (one for each painter URI), which in turn requires to run n ASK queries. Thus, in case we do not use the proposed index of known endpoints, the expected cost for running n ASK queries is about $n \times 200$ ms.

```
SELECT DISTINCT ?painter ?work  WHERE {
  SERVICE <http://dbpedia.org/resource/Category:Greek_painters> {
    ?painter <http://purl.org/dc/terms/subject> ?greekPainter }
  SERVICE <http://europeana.ontotext.com/sparql> {
    ?objectInfo <http://purl.org/dc/elements/1.1/creator> ?painter .
    ?objectInfo <http://www.openarchives.org/ore/terms/proxyFor> ?work } }
```

Fig. 4: Example of a SPARQL query that calls the same remote SPARQL endpoint multiple times.

4.2 Request-scope Caching of Fetched Datasets

A SPARQL query may contain multiple SERVICE invocations against the same Web resource. Consider for example the query of Fig. 3. In case the same co-author exists in more than one publications, the corresponding RDF triples (of co-author's URI) will be redundantly fetched multiple times.

In such cases, fetching and loading repeatedly the same resource triples costs both in time, computer resources and traffic load. To avoid this, for a submitted query we can use a *request-scope* cache (usable only in the context of a submitted query) of datasets that have been already fetched. Thereby, in each new SERVICE invocation, we first check if the corresponding URI exists in the cache in order to avoid re-fetching its triples. The cache can be cleared after query execution. Of course, one could instead apply a caching policy that will keep the fetched resources in cache after query execution for serving future queries (for a period of time and according to the available main memory), e.g., a combination of static and dynamic caching as it is used by web search engines [21].

5 Evaluation

We have seen that using SPARQL-LD, one can run queries which are more expressive than those supported by SPARQL 1.1. Nevertheless, here we evaluate the efficiency of the extended SERVICE operator for several querying scenarios, examining also the cost of each task of query execution. We first evaluate the time for retrieving the properties of several randomly selected resources (URIs) using different access methods (§5.1). This can also reveal which RDF format offers the lower average query time. Then, we examine the case of querying very large Web resources, i.e. resources containing even millions of triples (§5.2). This allows us to inspect the scalability of our implementation. Finally, we evaluate the effect of the proposed optimizations (§5.3).

The experiments were carried out using an ordinary computer with processor Intel Core i7 @ 3.4Ghz CPU, 8GB RAM and running Windows 7 (64 bit). The implementation is in Java 1.7. All data used in the experiments (URIs, queries, etc.), as well as the full results, are publicly available at http://users.ics.forth.gr/~fafalios/sparql-ld/Eval.zip.

5.1 Query Execution Time

We run experiments for 1,000 randomly selected DBpedia URIs belonging to the following 10 (randomly selected) DBpedia resource classes: *Artist, Painter, Scientist, Region, River, Fish, Athlete, BasketballPlayer, SportTeam, ChemicalCompound*. Notice that DBpedia publishes its data following the Linked Data principles (i.e., the URIs are dereferenceable) and also offers the data of a URI (properties and related entities) online in various formats including N3 and RDF/XML. Moreover, DBpedia has a publicly available SPARQL endpoint (`http://dbpedia.org/sparql`). We measured the total time that is required for retrieving the *outgoing properties* of each URI using the following 4 access methods: (1) by querying its dereferenceable URI, (2) by querying its RDF/XML file, (3) by querying its N3 file, and (4) by querying DBpedia's SPARQL endpoint.

Fig. 5 depicts a boxplot of the results. We notice that in all cases the average query time is in the scale of milliseconds. Specifically, the average time is: 654 ms when querying the dereferenceable URIs (and the mean 636 ms), 335 ms when accessing the RDF/XML files (and the mean 297 ms), 323 ms when accessing the N3 files (and the mean 293 ms), and 305 ms when querying DBpedia's endpoint (and the mean 288 ms). We notice that querying the RDF/XML or N3 files has almost the same performance as querying the endpoint (the difference is only a few milliseconds), although querying the endpoint does not require checking the URI content type as well as reading and loading the corresponding RDF triples (since the query is directly evaluated by the remote endpoint and the result is returned). Moreover, we see that querying the dereferenceable URIs is more costly compared to the other approaches. This is maybe due to the fact that DBpedia first checks the value of the `ACCEPT` request property for returning the URI contents in the RDF/XML format (and not in HTML).

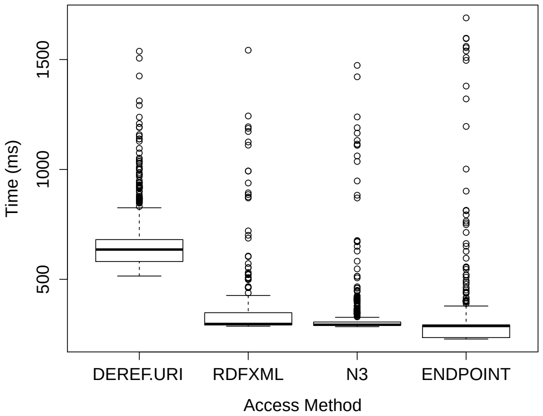

Fig. 5: Query execution time for several access methods.

We should stress here that the results are highly affected by the network status at the time of query execution and by the status of the server hosting the remote resource. This is evident by noticing the several outlier cases in the boxplot. For example, for the case of N3 files, although the query execution time was about 300 ms for the majority of URIs, some of them required more that one second. By inspecting the contents of these URIs, we noticed that their number of triples does not differ compared to the average case, thus either the network or the remote server (DBpedia's server) is responsible for this delay.

We also examined the time required by the main subtasks of query execution, specifically: (a) the time to check if the URI given to the `SERVICE` operator corresponds to an endpoint, (b) the time to get the URI content type (in case it is not an endpoint), and (c) the time to fetch and load the RDF statements that correspond to the given URI (in case it is not an endpoint). Fig. 6 depicts the results. The average time is 194 ms for (a) (and the mean 160 ms) and 174 ms for (b) (and the mean 146 ms). Here also we notice some outliers due to network delay. As regards (c), the average time is 289 ms (and the mean 283 ms) for the case of dereferenceable URIs, 148 ms (and the mean 121 ms) for the case of RDF/XML, and 138 ms (and the mean 118 ms) for the case of N3. We notice that accessing the N3 files is slightly more efficient. This can be justified by the fact that the N3 format is more compact and smaller in size than RDF/XML.

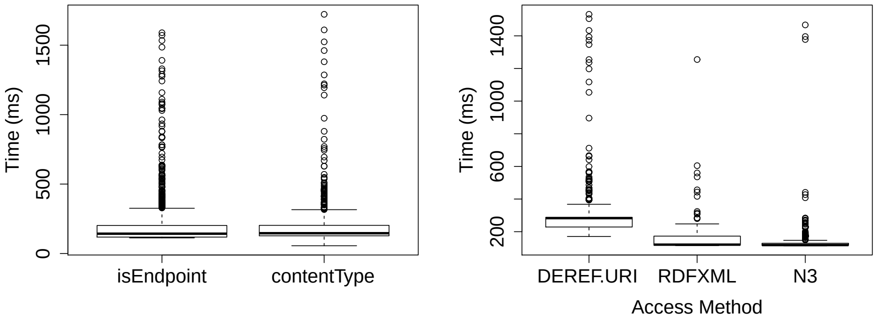

Fig. 6: **Left:** Time for (a) checking if the URI corresponds to a SPARQL endpoint, and (b) checking the URI content type. **Right:** Time for (c) fetching and loading the RDF statements corresponding to the given URI using different access methods.

5.2 Accessing Very Large Web Resources

We ran experiments for testing the case of accessing very large Web resources, i.e., resources containing a big number of RDF triples. We created four N3 files using real data coming from DBpedia. Specifically, we downloaded the triples of the English titles from the canonicalized 2014 dataset (single file containing about 11 millions triples). From this file, we created four files of 10^4, 10^5, 10^6, and 10^7 triples respectively, and we uploaded them in a Web accessible server. For each URI, we submitted a query that requests the properties of a particular resource (that exists in all files as subject in the triple).

Table 1 depicts the results. We notice that the total query execution time is less than 1 sec in the case of 10^4 triples, while for 1m triples the time is about 30 secs. For bigger files, the time can be in the scale of minutes. However, we should stress here that, usually, the online RDF files are not very big in size because large files cannot be easily handled/exchanged. For instance, DBpedia publishes one file (of small size) for each named-entity. As regards the main subtasks of query execution, we notice that, as expected, fetching and loading the triples is the most time consuming task.

Table 1: Querying large online N3 files.

Num of triples	Total Query Time	Is endpoint	Get content type	Fetch triples
10^4	900 ms	318 ms	7 ms	449 ms
10^5	3.2 sec	1,3 sec	8 ms	1.8 sec
10^6	31 sec	11 sec	8 ms	19 sec
10^7	546 sec	111 sec	51 ms	433 sec

5.3 Effect of Optimizations

We run experiments with and without the proposed optimizations. As regards the first optimization *(index of known endpoints)*, the expected speedup depends on the number of `SERVICE` calls to endpoints that exist in the index. As regards the second optimization *(caching of fetched datasets)*, the expected speedup depends on both the number of `SERVICE` calls to already-fetched resources and on the size (number of triples) of these resources. The queries used in this evaluation are available at `http://users.ics.forth.gr/~fafalios/sparql-ld/Eval.zip`. We run each query 3 times and here we report the average values.

Regarding the first optimization, we run experiments for different number of calls to "known" remote endpoints. Table 2 shows the speedup for each case. The speedup is calculated as the query execution time when the optimization is not applied divided by the optimized time. We notice that, using the proposed optimization method, the query execution time can be significantly improved (in our experiments, it is from 1.6 to 3.9 times faster).

Table 2: Effect of first optimization (*index of known endpoints*).

Query	Num of calls to indexed endpoints	Time without Opt.	Time with Opt.	Speedup
Q1	10	3.5 sec	1.8 sec	1.9×
Q2	10^2	27.2 sec	16.5 sec	1.6×
Q3	10^3	9.6 min	2.5 min	3.9×
Q4	10^4	44.8 min	24 min	1.9×

As regards the second optimization, we run experiments for different number of calls to already-fetched resources and for different number of triples in these resources. Table 3 shows the results. As expected, this optimization can highly improve the efficiency of query execution (in our experiments, it is from 1.2 to 24.6 times faster), while it also reduces the transfer of data between local server and remote sources.

6 Conclusion

We have presented `SPARQL-LD`, a generalization of SPARQL 1.1 Federated Query that allows to directly fetch and query RDF data from any HTTP Web source.

Table 3: Effect of second optimization (*caching of fetched datasets)*

Query	Num of calls to cached datasets	Num of triples	Time without Opt.	Time with Opt.	Speedup
Q5	**16**	10^3	11.9 sec	1.4 sec	8.5×
Q6	**16**	10^4	72.9 sec	5.7 sec	12.7×
Q7	**16**	10^5	10.3 min	39.8 sec	15.5×
Q8	10	$\mathbf{10^2}$	11.8 sec	9.6 sec	1.2×
Q9	10^2	$\mathbf{10^2}$	35.9 sec	10.7 sec	3.4×
Q10	10^3	$\mathbf{10^2}$	4.8 min	11.6 sec	24.6×

Using `SPARQL-LD` one can exploit and combine in the same SPARQL query: i) data stored in the (local) repository, ii) data coming from online RDF or JSON-LD files, iii) data embedded in Web pages as RDFa, iv) data coming from dereferenceable URIs, v) data that is dynamically created by Web Services, and vi) data coming by querying other SPARQL endpoints. A distinctive characteristic of this extension is that it enables to also query datasets coming from the partial results of a query (i.e., discovered at query execution time). We also identified factors that can affect the efficiency of `SPARQL-LD` and we proposed optimizations that manipulate such cases.

The functionality offered by `SPARQL-LD` motivates Web publishers to follow the Linked Data principles and expose their data in RDF without needing to set up and maintain a costly SPARQL endpoint. For instance, a museum can enrich its Web page with RDFa, or just put online a RDF dump, and thereby make its data directly accessible via SPARQL.

The conducted experiments showed that, as expected, the time for querying the triples of online RDF resources highly depends on the number of triples existing in the resource, on the status of the network between the local server and the remote server hosting the resource, and on the status of the remote server itself. Nevertheless, we saw that for common Web resources of normal size (less than 10^4 triples), and without using any caching method, the total query time is very low. We also saw that the performance of querying RDF/XML or N3 files is almost the same as querying an endpoint. Finally, as regards the proposed optimizations, experimental results showed that they can highly improve the query execution time. For instance, in our experiments, using a request-scope cache of fetched datasets, the execution time of a query reduced from about 5 minutes to only 12 seconds.

In future, we will study query planning approaches and more optimization and caching techniques appropriate for `SPARQL-LD`.

Acknowledgements. This project / work has received funding from the European Union's Horizon 2020 research and innovation programme under the BlueBRIDGE project (Grant agreement No 675680).

References

1. A JSON-based Serialization for Linked Data. `http://www.w3.org/TR/json-ld/`.
2. Apache Jena. `http://jena.apache.org/`.

3. RDFa Core 1.1. `http://www.w3.org/TR/2015/REC-rdfa-core-20150317/`.
4. SPARQL 1.1 Query Language (W3C). `http://www.w3.org/TR/sparql11-query/`.
5. SPARQL Federat. Query. `http://www.w3.org/TR/sparql11-federated-query/`.
6. C. Bizer, T. Heath, and T. Berners-Lee. Linked Data-The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.
7. C. Buil-Aranda, M. Arenas, O. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *Web Semantics: Science, Services and Agents on the World Wide Web*, 18(1), 2013.
8. C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *ISWC 2013*. Springer, 2013.
9. P. Fafalios, M. Baritakis, and Y. Tzitzikas. Exploiting Linked Data for Open and Configurable Named Entity Extraction. *International Journal on Artificial Intelligence Tools*, 24(02), 2015.
10. P. Fafalios and Y. Tzitzikas. SPARQL-LD: A SPARQL Extension for Fetching and Querying Linked Data. In *The Semantic Web–ISWC 2015 (Posters & Demonstrations Track)*, Bethlehem, Pennsylvania, USA, 2015.
11. O. Hartig. SPARQL for a Web of Linked Data: Semantics and Computability. In *9th ISWC*. Springer-Verlag, 2012.
12. O. Hartig and J. Pérez. LDQL: A Query Language for the Web of Linked Data. In *The Semantic Web-ISWC 2015*, pages 73–91. Springer, 2015.
13. B. Haslhofer, E. Momeni Roochi, B. Schandl, and S. Zander. Europeana RDF store report. 2011.
14. A. Hogan, A. Harth, J. Umbrich, S. Kinsella, A. Polleres, and S. Decker. Searching and Browsing Linked Data with SWSE: The Semantic Web Search Engine. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(4), 2011.
15. A. Isaac and B. Haslhofer. Europeana linked open data–data. europeana. eu. *Semantic Web*, 4(3):291–297, 2013.
16. K. Kjernsmo. A survey of HTTP caching implementations on the open Semantic Web. In *The Semantic Web. Latest Advances and New Domains*. Springer, 2015.
17. A. Langegger, W. Wöß, and M. Blöchl. A Semantic Web Middleware for Virtual Data Integration on the Web. In *5th ESWC*. Springer-Verlag, 2008.
18. M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with SPARQL query caching. In *The Semantic Web: Research and Applications*, pages 304–318. Springer, 2010.
19. D. Miranker, R. Depena, H. Jung, J. Sequeda, and C. Reyna. Diamond: A SPARQL query engine, for linked data based on the rete match. *AImWD 2012*, 2012.
20. G. Montoya, M.-E. Vidal, and M. Acosta. A Heuristic-Based Approach for Planning Federated SPARQL Queries. *COLD*, 905, 2012.
21. M. Papadakis and Y. Tzitzikas. Answering keyword queries through cached subqueries in best match retrieval models. *Journal of Intelligent Information Systems*, 44(1):67–106, 2015.
22. J. Purday. Think culture: Europeana. eu from concept to construction. *The Electronic Library*, 27(6):919–937, 2009.
23. B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *5th ESWC*. Springer, 2008.
24. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on Linked Data. In *The Semantic Web–ISWC 2011*. Springer, 2011.
25. Y. Tzitzikas, C. Alloca, C. Bekiari, Y. Marketakis, P. Fafalios, M. Doerr, N. Minadakis, T. Patkos, and L. Candela. Integrating Heterogeneous and Distributed Information about Marine Species through a Top Level Ontology. In *MTSR*, 2013.