Querying the Linked Data Graph using owl:sameAs Provenance

Sam Coppens, Ruben Verborgh Erik Mannens, and Rik Van de Walle

Ghent University - IBBT, Multimedia Lab, Gaston Crommenlaan 8 /201, Ghent, Belgium,
Firstname.Lastname@ugent.be

Abstract. Querying the Linked Open Data cloud as a whole still remains problematic. Prior knowledge is required to federate the queries to the appropriate datasets: each dataset provides its own SPARQL endpoint to query that dataset, and each dataset uses its own vocabulary to describe their information. In this paper, we propose a federated and asynchronous SPARQL framework to query the global data space. Our query federation is based on the fact that owl:sameAs relationships are symmetrical and transitive and, hence, that such linked resources are interchangeable. The provenance of the generated owl:sameAs links during the enrichment process is processed to resolve the SPARQL endpoints of linked datasets and their vocabulary mappings. This information is used during the query federation. Asynchronous SPARQL processing makes the query federation more scalable and exploits better web caching.

Keywords: Distributed, Linked Open Data, sameAs, SPARQL

1 Introduction

The main goal of Linked Open Data is to create a global data space, the Linked Open Data cloud\(^1\) \[1\], where end users can easily discover and consume data. Crucial for Linked Open Data is link generation. Link generation will discover relationships between the resources being published and the resources already being described in the Linked Open Data cloud using the SPARQL protocol\[7\]. This step will link the data being published with information of other data sources, turning the disparate set of data sources into one global data space. Bizer et al. \[2\] explain very clearly the concept and technical principles of Linked Data and why link discovery between datasets is crucial in creating a global data space.

At the moment, querying this global data space as a whole remains a challenge. If we want to get results from a query that ranges over multiple data sources, accessible through different SPARQL endpoints, we will have to split up the query into subqueries for each SPARQL endpoint and combine the results or we will have to fetch firstly all the different data sources into one data source and then query this aggregated data source. Both approaches have some limitations though:

\(^1\) http://linkeddata.org/
In the first approach, we will have to know which part of the query is meant for which SPARQL endpoint and we will need to know for each SPARQL endpoint which vocabulary it uses, so that appropriate mappings to the query can be applied.

In the second approach, we don’t need to know which SPARQL endpoints must be queried for receiving an appropriate answer, but we still need to know which vocabulary to use in the query so that the aggregated data sources can understand the query. This approach has the danger that in a highly interlinked environment, a lot of information has to be indexed first.

This paper proposes a solution to the problem of querying distributed data sources. In our approach, we will traverse the owl:sameAs links of the resources combined with the provenance of these owl:sameAs links so we know how to distribute the query to find more information on resources. The basic principle of our query federation framework is that owl:sameAs is a symmetrical and transitive property. This means that resources linked via owl:sameAs (directly or indirectly) are interchangeable. This forms the basis for the query distribution algorithm, such that all the information of a resource and all its owl:sameAs equivalents, distributed on the Web, becomes available at a single SPARQL endpoint and creates the impression that all the information of the owl:sameAs equivalents is also present in the local dataset. At the same time, mappings will be applied to the query distributions to support the external dataset’s vocabulary. This mapping information is retrieved implicitly by processing the provenance of the sameAs links, as will be shown in Section 3. Because of this feature, the mappings are always defined in terms of the local vocabulary. Thus, the local vocabulary forms the universal data model for information federation. In the end, this mapping strategy is something in between a point-to-point mapping strategy and a strategy using a universal data model.

Following these sameAs links from the Linked Open Data cloud has several benefits. First of all, Following the sameAs links creates your own view on the Linked Open Data cloud. When publishing your information as Linked Open Data, you will only link with those resources from datasets you trust. By following the sameAs links for query federation, our query distribution framework only takes into account those datasets you trust and linked with. A second major advantage of following the sameAs links is that they provide an easy mechanism to do distributed joins. In RDF all resources are identified by a URI. the sameAs links will link two related resources to each other by means of interrelating their URIs through owl:sameAs. This means, the identifier of the remote resource, linked to the local resource is known and can be used to perform the distributed join operation. The paper is structured as follows: in the following section, we give an overview of existing related work. Section 3 will elaborate on our solution, where we also discuss in detail the two main components of our framework, i.e., the index builder, and the query distributor. In Section 4, we give an overview of the optimisations that are applied to enhance the query execution performance. We end the paper with an evaluation and a conclusion.
2 Related Work

At this moment, there are several approaches to distributed querying. Some approaches rely on prior crawling and caching of the data, e.g. Sindice [10], which crawls web pages embedding RDF and microformats and makes the crawled data available through a SPARQL endpoint and through an API. Actually, this is not distributed querying, but querying a semantic web index, built from crawling and caching. Another approach to distributed querying is relying on runtime link traversal to answer queries. This approach is followed by SQUIN [5]. Here, the index is built at query runtime, which avoids syncing problems. A third approach being used, is based on query federation, which is followed by frameworks like FedEx: a federation layer for distributed query processing on Linked Open Data[9] and DARQ [8]. FedEx provides a query distribution layer on top of the Sesame framework. It executes query federation and query optimisation. DARQ federates the queries using the predicates to decide where to send triple patterns to as an optimisation technique. Just like FedEx, is Splendid [3] an extension to Sesame, which employs VoID to distribute its incoming queries. Splendid will start with ASK SPARQL queries to each dataset for verification and later on statistical information is used to optimise the federated queries.

Our framework is a combination of the link traversal approach and the query federation approach to solve queries ranging multiple, disparate datasets. It is similar to what is being done in ‘Data summaries for on-demand queries over linked data’[4]. This latter will use an index structure for optimising the query distribution and then queries the data real time, so no syncing problems occur. Our framework will federate queries such that they follow the owl:sameAs links. Our approach actually exploits the fact that resources linked via a owl:sameAs link, are interchangeable, which automatically brings in the distributed information of these resources. At the same time, we will introduce property mappings and class mappings to solve our incoming queries and overcome interoperability issues between the linked datasets. The benefit of this approach is that a data provider gets more control over his data. He decides which datasets are used to enrich his data and, as a consequence, he controls the SPARQL endpoints to which incoming queries for his data are distributed to.

Our framework is integrated into ARQ\(^2\). ARQ is a query engine for Jena\(^3\) that supports the SPARQL RDF Query language. Many SPRQL endpoint implementations are based on ARQ. This allows for any SPARQL endpoint service provider relying on the ARQ library to put up easily its own distributed SPARQL endpoint. They just have to replace the ARQ library with our extended ARQ library and feed ARQ with the SPARQL construct queries, used for enriching their dataset, or with SILK configuration files, if they used SILK to enrich their dataset. Thus, it becomes very easy for a data publisher to set up a distributed SPARQL endpoint, which federates queries to those datasets it is linked with.


\(^3\) [http://jena.apache.org/](http://jena.apache.org/)
3 Solution

The main idea behind our distribution framework is to rewrite incoming queries in such a way that the symmetry and transitivity of owl:sameAs is exploited. To achieve this, our framework traverses the owl:sameAs links of the resources using SPARQL. Incoming queries are split up in subqueries, which can be evaluated independently. For each subquery, we look for possible owl:sameAs linked resources. These subqueries are refactored to target also the remote SPARQL endpoints of these owl:sameAs linked resources. At the same time, we apply vocabulary mappings to the refactored subqueries to match the remote datasets’ vocabularies. Thus, to distribute the queries, some prior knowledge is needed:

- SPARQL endpoints: for each discovered related resource, we have to know the SPARQL endpoint we can consult for retrieving information on the related resource.
- Vocabulary mappings: we cannot just distribute the same query to the different SPARQL endpoints, because every endpoint uses its own vocabulary to describe things, thus we need appropriate mappings for disparate classes and properties of the remote data sources. Some datasets use, e.g., rdfs:label for denoting the name of a person, other datasets use, e.g., foaf:name. These latter datasets needs to be queried using foaf:name instead of rdfs:label to retrieve the person’s name.

This prior knowledge can be retrieved from the provenance of the owl:sameAs links. The provenance of these owl:sameAs links can be expressed as rules in the form of SPARQL construct queries, which will be explained in detail in Section 4. This allows us to build a lookup table, called distribution index which stores the service endpoints and for each service endpoint the property mappings and the class mappings. This information can be extracted from the SPARQL construct queries, representing the owl:sameAs links’ provenance. This index building process takes place prior to the queries and not during the queries. It is a preprocessing step. This lookup table is part of the distributed SPARQL processor, which will use this information to answer its incoming queries and to distribute the subqueries accordingly.

Our distribution framework is implemented as an extension to ARQ. It has two main building blocks. The index builder processes the owl:sameAs links’ provenance to build up the distribution index. This is explained in Section 4. The query distributor is the second main building block, responsible for federating and mapping the incoming queries, using the distribution index. This block is explained in Section 5.

4 Index Builder

The Index Builder is responsible for building the distribution index. This index must be built before any queries are being fired. The index builder will take provenance information of the owl:sameAs links to extract the services to which it will distribute incoming queries and for each service it will extract possible mappings.
Many owl:sameAs links are generated based on some rule, representing the provenance of a owl:sameAs link. Figure 1 shows how a owl:sameAs link, relating two persons, can be represented as a SPARQL construct query. The index builder actually processes such rules to feed the distribution index. Our framework implements two instances of this index builder as an extension to ARQ. There are, thus, two ways of feeding the distribution index:

- A SPARQL index builder that is fed with SPARQL construct queries, representing the provenance of the owl:sameAs links present in the dataset. In fact, these rules (SPARQL construct queries) are the queries used for interlinking the dataset. The SPARQL queries are compiled into SPARQL algebra and the index builder algorithm, described below, is implemented such that it operates on SPARQL algebraic expressions.

- A SILK index builder that takes as input a configuration file of SILK [6], the link discovery framework. This configuration file has all the information available to represent the provenance of the discovered links as SPARQL construct queries. Hence, an index builder was also implemented for this sort of input. This SILK index builder just parses the SILK configuration file and directly fills up the index, because the SPARQL endpoint, class mappings and property mappings for this SPARQL endpoint are directly available from this configuration file.

---

4 http://www4.wiwiss.fu-berlin.de/bizer/silk/
4.1 Index Builder Algorithm

As explained, the provenance information of all the owl:sameAs links of a dataset can be expressed as a number of SPARQL queries. The index builder iterates through all these queries and for each query it does the following:

1. Extract the remote SPARQL endpoint and store it in the *distribution index*. In our example this is <http://dbpedia.org/sparql>.
2. Extract the query variables from the CONSTRUCT clause, expressing the local resource (e.g., ?resource) and remote resource (e.g., ?remoteresource).
3. Extract the triple pattern for the local resource and the triple pattern for the remote resource.
4. Extract possible FILTER expressions for the local resource and the possible FILTER expressions for the remote resource.
5. Extract all different paths from both the local and remote triple patterns. These paths can include FILTER expressions, thus for building the paths also the extracted FILTER expressions are taken into account.

**Local path 1:** ?resource a owl:Thing.
**Local path 2:** ?resource lons:type "person".
**Local path 3:** ?resource lons:name ?concept.
   ?concept skos:prefLabel ?name

**Remote path 1:** ?remoteresource a foaf:Person.
**Remote path 2:** ?remoteresource foaf:name ?remotename.
   FILTER (str(?remotename) = ?name)

6. For every path, extracted from the remote triple pattern and FILTER expressions, that ends in a query variable, find the corresponding path, extracted from the local triple pattern and FILTER expressions, that ends with the same query variable. The part of the remote path, starting from the remote query variable to the query variable, and the part of the local path, starting from the local query variable, are property mappings of each other, as shown in the example below. This mapping is stored in the *distribution index* for this SPARQL endpoint.

**Property mapping 1:**
   FILTER (str(?remotename) = ?name)

7. The remaining paths from the remote and local triple patterns and FILTER expressions, make up the class mapping. This is also stored in the *distribution index* for this SPARQL endpoint.

**Class mapping 1:**
   ?resource a owl:Thing. ?resource lons:type "person" =
   ?remoteresource a foaf:Person.
To summarise, we list here the information to be stored in the distribution index using our index builder algorithm:

- Sparql endpoint = http://dbpedia.org/sparql

5 Query Distributor

The query distributor refactors the incoming query to become a distributed query using the information of the distribution index built by our index builder. The query will be transformed in such a way that all linked resources via owl:sameAs become interchangeable, which aggregates all the information that directly or indirectly is available on these resources. Thus, our framework actually makes use of the fact that owl:sameAs is symmetrical and transitive.

This distribution of the incoming query is done during the query optimisation in ARQ, thus after the incoming query has been compiled to a SPARQL algebraic expression. The query distributor is implemented as a query transformation that uses the distribution index to perform its transformations. It will search for triple patterns that are not part of a SPARQL SERVICE element. Triple patterns belonging to a SPARQL SERVICE operator are not altered, only the part of the incoming query that is meant to be evaluated against the local data is affected by the query distributor. The query distribution algorithm will distribute incoming queries in two phases. First, the basic graph patterns (BGPs) of the query are being distributed during the Transform BGP phase. Later on, these distributed BGPs will be merged appropriately during the Merge BGP phase. The details of the query distribution algorithm are discussed hereafter.

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpediaont: <http://dbpedia.org/ontology/>

WHERE {
  { ?book dbpediaont:writer ?author. | BGP 1
    ?book rdfs:comment ?comment. | BGP 2
  } OPTIONAL
  {
  }
}

Fig. 2. Example Query Distribution
5.1 Transform BGPs Algorithm

This algorithm operates on a SPARQL algebraic expression and will, as explained, transform the BGPs first individually. This distribution is illustrated with an example. In this example, the index is built from the query rule, shown in Figure 1, which was discussed in detail in Section 4. The incoming query, this algorithm is illustrated with, is shown in Figure 2.

The query distribution algorithm does for every BGP of the incoming query the following:

1. Extract the triples of the BGP to be evaluated against the local data (i.e., triples from triple patterns not part of a SPARQL SERVICE element).
2. Extract possible FILTER expressions affecting these extracted triple patterns.
3. Extract all nodes from the BGP. These nodes can be URIs or query variables (e.g. ?book, ?booktitle, ?author, and ?comment for BGP 1 from our example depicted in Figure 2).
4. For every node, extract all possible paths (i.e. query paths) from the extracted triples and FILTER expressions. Thus, these query paths can include FILTER expressions, apart from the triples. They can be seen as property paths, extended with FILTER expressions.

Query paths extracted with starting node ?book from BGP 1:

5. Every extracted path from the previous step has a local variant (e.g. ?book dbpediaont:writer ?author) and a remote variant (e.g. ?book owl:sameAs ?DQV0. SERVICE <http://dbpedia.org/sparql> ?DQV0 dbpediaont:writer ?author), which are tied together using the UNION operator, as shown in Figure 3. For the remote variant of the query path, the start node of that query path is decoupled into the owl:sameAs equivalent. The owl:sameAs equivalent is distributed to the SPARQL endpoints from the distribution index and at the same time mappings from the distribution index for that SPARQL endpoint are applied. The example depicted in Figure 3 shows the query paths extracted from node ?book for the example query depicted in Figure 2.

6. The outcome of the previous step are only the distributed query paths starting from a certain node (e.g. ?book in our example). These blocks (i.e. sequence operators) still need to be tied together. This is done during this last step using the UNION operator. This way, a BGP is split up in all possible query paths. Figure 4 gives a schematic overview of how a BGP 1 from the example depicted in Figure 2 is distributed. Each of the three right blocks represent the outcome from the previous step, but each using a different starting node.
Distributed query paths extracted with starting node ?book from BGP 1:
(sequence
(union
(bgp (triple ?book dbpediaont:writer ?author)) |Local QP 1
(sequence
(bgp (triple ?book owl:sameAs ?DQV0)) |Remote QP 1
(service <http://dbpedia.org/sparql> |
(bgp (triple ?DQV0 dbpediaont:writer ?author)))) |)
(union
(bgp (triple ?book rdfs:label ?booktitle)) |Local QP 2
(sequence
(bgp (triple ?book owl:sameAs ?DQV1)) |Remote QP 2
(service <http://dbpedia.org/sparql> |
(bgp (triple ?DQV1 rdfs:label ?booktitle)))) |)
(union
(bgp (triple ?book rdfs:comment ?comment)) |Local QP 3
(sequence
(bgp (triple ?book owl:sameAs ?DQV2)) |Remote QP 3
(service <http://dbpedia.org/sparql> |
(bgp (triple ?DQV2 rdfs:comment ?comment)))) |)

Fig. 3. Part of the outcome from step 5 of the Transform BGPs algorithm

5.2 Merge BGP Algorithm

The previous phase only distributed the BGPs. These distributed BGPs cannot
be merged as such. If we take back the example depicted in Figure 2 and consider
BGP 2, the nodes extracted from this BGP are only ?book and ?producer. If the
local dataset only has information on authors (which are linked to their DBpedia
owl:sameAs equivalent) and we want to have the producer of a book, there needs
to be a query path from ?author to ?producer. This query path is not extracted
from BGP 1, nor from BGP 2. For this, we need a special merging algorithm.
The result of such a merge is shown in Figure 5. This example only shows the
merged operator with query path starting with node ?book. The result is an
extra query path is merged using a LEFTJOIN operator. To achieve this, our
merging algorithm works as follows:
1. The triples from BGP 1 and BGP 2 are merged.
2. From the merged triples all nodes are extracted (e.g. ?book, ?booktitle, ?comment, and ?producer).
3. For every node, all query paths are extracted.
4. If a query path from a certain node is not yet present in the distributed BGP 1, then the query path is merged with the already extracted query paths for that node in the distributed BGP 1. This merge is done using the operator that binds BGP 1 and BGP 2, i.e. LeftJoin operator in this case, which is the algebraic equivalent of an OPTIONAL operator in SPARQL, as shown in Figure 5.

```
Merged (BGP 1 and BGP 2), distributed query paths extracted with starting node ?book:
(leftjoin
(sequence operator shown in step 5 of the transform BGP algorithm )
(union
 (sequence
 (bgp (triple ?book owl:sameAs ?DQV0))
 (service <http://dbpedia.org/sparql>
 (bgp (triple ?DQV0 dbpediaont:writer ?producer)))))
```

**Fig. 5.** Merging of distributed BGP 1 and BGP 2 - only that part with query paths starting with ?book node

6 Optimisations

**Asynchronous, Distributed Query Executor** Until now, SPARQL query processing in ARQ is synchronous, but non-blocking. This means communication with the client can be made asynchronous, but this is only effective if the processing of the query also happens asynchronously. By introducing a SPARQL operator, which is processed asynchronously, the SPARQL processing becomes asynchronous. By doing this, results become available as soon as the SPARQL processor has found some answers, but there are also some benefits regarding the performance of the SPARQL processing. Looking at the schematic overview of a distributed query, shown in Figure 6, a query is split up in different versions of the query from the perspective of a node that needs to be decoupled into its owl:sameAs equivalents (UNION operator in Figure 6). All these different query versions, of which one is depicted in Figure 5.2, need to be joined. This join can be replaced by our asynchronous operator, which will evaluate all the different versions of the query asynchronously and concurrently. Our operator, is in fact an asynchronous UNION operator, merging all its incoming results.
Query Paths  The basic entity of our distribution algorithm operates on is a query path. A query path is actually a property path extended with FILTER expressions and GRAPH expressions. These query paths are distributed to the remote SPARQL endpoints and, hence, FILTER and GRAPH expressions are evaluated at the remote SPARQL endpoint.

Mappings  To the query path that are being federated, mappings are applied to refactor the query path to the vocabularies the remote dataset supports. This enhances interoperability of the query federation framework and makes our framework more robust. The mappings are retrieved implicitly, by processing the provenance of the sameAs links. As a consequence, the mappings are always expressed in terms of the local vocabularies used. It can be seen either as a point-to-point mapping strategy or a strategy using a universal data model, where the universal data model is always the local data model from which the distributions start.

7 Evaluation

For evaluating our distribution framework, we make use of the Berlin SPARQL Benchmark. The dataset of this benchmark, consisting of 10 000 product descriptions, is published over two SPARQL endpoints, i.e., a local and a remote SPARQL endpoint. We have set up a local SPARQL endpoint, whose dataset only contains product URIs, linked via owl:sameAs with the product resources of the remote BSBM dataset. The BSBM querymix is fired at the local SPARQL endpoint, which distributes the queries using our distribution framework to solve the queries. For this, it contacts the remote SPARQL endpoint. The execution times of the results are used for evaluation. These result times are evaluated against the result times of the 'perfect' distributed query. This 'perfect' distributed query just forwards the whole query of the BSBM querymix to the remote SPARQL endpoint using the SERVICE operator. This way, we evaluate the result times of a query once using our distribution algorithm against the result times of a query using a 'perfect' query distribution. By doing this kind of evaluation, our evaluation is also independent of the used hardware/software and the size of the BSBM dataset and evaluate the distribution algorithm itself.

The table below shows the results for only a part of the queries of the BSBM querymix. The other query results showed similar performance.

<table>
<thead>
<tr>
<th>perfect' distribution algorithm</th>
<th>Result time first result</th>
<th>Result time last result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>780 ms</td>
<td>3763 ms</td>
</tr>
<tr>
<td>Query 3</td>
<td>2701 ms</td>
<td>4076 ms</td>
</tr>
<tr>
<td>Query 7</td>
<td>517 ms</td>
<td>603 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>our distribution algorithm</th>
<th>Result time first result</th>
<th>Result time last result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>967 ms (+23%)</td>
<td>4480 ms (+19%)</td>
</tr>
<tr>
<td>Query 3</td>
<td>3594 ms (+22%)</td>
<td>4929 ms (+21%)</td>
</tr>
<tr>
<td>Query 7</td>
<td>631 ms (+22%)</td>
<td>862 ms (+42%)</td>
</tr>
</tbody>
</table>
8 Conclusion

In this paper, we described our framework for distributing queries relying on the symmetry, transitivity, and the provenance information of owl:sameAs. This way, our framework acts like a window on the Linked Open Data cloud, where all information is available on resources that are directly (via owl:sameAs link) linked to one of your resources, or indirectly (via a owl:sameAs link of a linked resource). For this, the framework relies on the provenance of the owl:sameAs links, which can be expressed by a rule or SPARQL construct query. The rules or queries give information on the SPARQL endpoint to use for querying that linked dataset and some vocabulary mappings to use for querying that dataset.

Our framework is implemented as an extension of ARQ and consists of two main components: the index builder, and the query distributor. The index builder will build an index based on the provenance of the owl:sameAs links. It supports building an index based on SPARQL construct queries or rules, which express the provenance of the generated owl:sameAs links, and it supports also building an index based on a configuration file for the SILK enrichment framework. The query distributor will refactor the incoming queries to distributed subqueries, targeting the SPARQL endpoints of the linked datasets, during the query optimisation phase. During this refactoring the appropriate mappings are applied on the triple patterns, mapping the triple patterns to vocabularies the remote dataset supports. This approach gives data providers more control. They can easily set up their own distributed SPARQL endpoint, which will distribute incoming queries to those datasets they trust, because these are the datasets they use for interlinking. We evaluated our platform using the BSBM benchmark, but in a distributed environment. For this the queries from the benchmark had to be distributed to answer the queries.

9 Acknowledgments

The research activities that have been described in this paper were funded by Ghent University, Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), and the European Union.

References