Interactive Query Processing for Linked Data

Hannes Mühleisen

Freie Universität Berlin
Networked Information Systems Group
Königin-Luise-Str. 24/26, 14195 Berlin, Germany
muehleis@inf.fu-berlin.de

Abstract. In the recent years, the concept of publishing data on the Internet using the Linked Data principles has gained more and more widespread acceptance. A plethora of novel applications are conceivable, which share the need for retrieving data from the “Linked Data cloud”. However, though the process of publishing Linked Data is straightforward, data retrieval poses a significant challenge. The location of data relevant to a query is generally unknown. While one could cache the entire cloud on a single system, an on-demand query processing approach is necessary to handle dynamic data. We have designed, implemented, and evaluated such an approach using a breadth-first retrieval process to execute SPARQL queries over the Linked Data cloud. Special care was taken to support interactive applications, which depend on displaying results to users quickly.

1 Introduction and Motivation

In the recent years, the concept of publishing data on the Internet using the Linked Data principles has gained more and more widespread acceptance. Tool chains and best practices to do so have emerged, including various server and client implementations. In a nutshell, Linked Data calls for structured meta data to be published using the Resource Description Format (RDF) data model and the following conventions: Resource identifiers have to be URLs, their resolution yields more RDF data describing the resource identified by that URL further. Furthermore, RDF graphs are requested to include links to other RDF graphs [1]. A plethora of novel applications are conceivable, including for example distributed social applications such as the “Distributed Address Book” [7] or visualization mash-ups like “ResearchersMap” [5].

All these applications are in the need for retrieving data from the set of information published as Linked Data, coined the “Linked Data cloud”. This task, however, cannot be solved as straightforward as the mentioned data publishing process. An obvious solution is to use crawling techniques as employed by major search engines and to periodically retrieve and cache all RDF graphs published using Linked Data on a single system. This approach is used for example by the search engine “Sindice” [8]. While data retrieval can undoubtedly
be performed quickly using such a central system, on-demand query processing is often preferable, especially on data sets that are likely to change frequently.

Therefore, we argue that a on-demand approach at executing queries over the Linked Data cloud is due, with special focus on embracing the properties of both Linked Data, the requirements of interactive applications, and the general Internet architecture with all its limitations. The rest of the paper is structured as follows: Section 2 introduces previous and related work. In Section 3 we then introduce our concept of breadth-first search and the delivery of intermediate results. Section 4 presents our evaluation results using a prototypical implementation and example application. Finally, Section 5 concludes this paper and gives an overview over further work.

2 Related Work

The task of retrieving data from a distributed environment has been researched to great lengths in the context of query processing for distributed relational databases. Kossmann described a reference schema for distributed query processing in order to generate the query execution plan [6]. However, central operations such as site selection and cost estimation rely on a fixed network infrastructure and central storage indices.

Quilitz and Leser have applied this reference architecture to the distributed execution of SPARQL queries on RDF data. They make use of so-called “Service Descriptions”, which are available on every node evaluating queries [10]. These descriptions enable them to efficiently use site selection and cost estimation to converge on an efficient query execution plan.

Both the network infrastructure and data distribution are unknown in the case of Linked Data, they are encapsulated within the URL lookup system. Additionally, network structures are largely unknown, and no system-inherent indices are present to describe the location of data relevant to the current query. Peer-To-Peer solutions are also not applicable, as they depend on cooperation from the participating hosts, which cannot be assumed.

Hartig et al. have presented the concept of “link traversal based query execution”. Here, queries are evaluated on-demand by extracting and resolving all mentioned URLs from a SPARQL query, and then selectively and recursively resolving the URLs mentioned in the retrieved RDF graphs [3,4]. While the concept was shown to be able to achieve impressive results just by following references, response times were found unsuitable for interactive applications, mainly due to the fact that remote systems are notoriously sluggish in responding to requests.
3 Query Processing Concept

In this chapter, we first outline the RDF data model and the SPARQL query language shortly. Then we describe the main issue when executing queries over the Linked Data cloud. Based on that, our breadth-first query processing concept is introduced, covering design assumptions and a detailed algorithm description.

3.1 Data Model and Query Language

RDF describes a data model based on graphs where resources and literal values are the nodes, and labeled directed arcs connecting those nodes referred to as “properties”. Such a graph is given in Fig. 1. Here, the resource with the URI `ex:res1` is connected with the property `ex:p1` to the resource with the identifier `ex:res2`. For the purpose of storage or transmission RDF graphs are decomposed into single statements of subject, predicate and object entries. In this case, a single statement with `ex:res1` for subject, `ex:p1` as predicate, and `ex:res2` as object value would suffice to represent this graph.

![Fig. 1. Simple RDF graph](image)

RDF graphs can be queried using the query language SPARQL, which in essence enables the user to formulate graph patterns to be matched to parts of a graph [9]. For the most common type of queries requesting tuples generated from the matching RDF graph, variables are bound to a set of concrete values referred to as bindings. Bindings can be handled like tuples retrieved from a relational database, for example by applying filtering, ordering, or limiter functions. Fig. 2 shows a simple RDF graph at the top and a visual representation of a matching graph pattern on the bottom. The corresponding SPARQL query is given in Listing 1.1 if it is evaluated on the graph, the result will be a binding of the variable `?r` to the URI `ex:res2`.

3.2 Problem Description

Using Linked Data, the graph could be stored on two different servers, one graph containing `ex:res1` and its properties, the other `ex:res2`, and the third `ex:res3`. To evaluate the query in this scenario, the reference to `ex:res1`
has to be resolved, a process referred to as “dereferencing”. To create result bindings, \textit{ex:res2} has to be resolved. In some cases, the pattern in the query contains hints which links to resolve. This is the main concept behind “link traversal based query execution” [3,4], where resources are only resolved if they are linked using a predicate contained in the query. However, this is not possible in all cases, as can be seen from this example: Results can only be bound by “following” the property \textit{ex:p1} to \textit{ex:res2}, which is not contained in the query, leaving the query processor no other choice but to resolve all URIs present in the graph.

![RDF graph (top) and matching pattern (bottom)](image)

**SELECT ?r WHERE { ?r ex:p2 ex:res1. }**

**Listing 1.1. SPARQL query**

### 3.3 Query Processing

For our approach to query processing, the main goal is to support interactive applications that utilize data from the Linked Data cloud. To achieve this, we have made two observations: First, the query processing should be able to deliver as many results as possible from the immediate neighborhood of the requested resources. This is due to the structure of the Linked Data cloud, where related graphs are more likely to reference to each other. Second, query results should be pushed to the client as quickly as possible, in order to support uninterrupted user interaction.

We made the following assumptions: (1) Applications do not make use of queries requiring a large number of subsequent dereferencing operations, as
these would be rather impossible to process using only on-demand retrieval. (2) The amount of links to other resources in an average RDF graph is limited. We have confirmed the latter to be reasonable by analyzing a partial crawl of the Linked Data cloud that was created for the Billion Triple Challenge 2009[1]. In this crawl, each of 49330 RDF graphs contained on average 123 links to remote resources. (3) The effort to evaluate a query on local in-memory RDF data is far less expensive than the effort to retrieve RDF data from a remote server using a HTTP request.

From these assumptions, our algorithm is designed as follows. From every URL mentioned either in the prefixes section or the pattern section of a SPARQL query, a breadth first link traversal is performed. This traversal makes use of a multilevel queue, where the URLs contained in the query represent level zero. As RDF graphs are being retrieved from the queue level zero, the URLs of resources contained in those graphs are queued for retrieval on queue level one. The URLs referenced in the graphs retrieving the URLs on queue level one are then added to queue level two and so on. Queue entries are marked, if an attempt has been made to retrieve RDF from them, and the retrieval process progresses through the queue starting on level zero, only continuing to the higher subsequent levels if all entries on the current level have been marked.

Applications specify the maximum amount of results to be found, and the maximum amount of time they are willing to wait for these results. The retrieval process periodically checks, whether the specified amount of results has already been found, or if the maximum execution time has been reached. The retrieval process is terminated in either case, sending all results to the client. Also, the partial results found so far are sent to the client periodically to support user interaction.

Algorithm 1 describes this algorithm in pseudo code: After the URIs mentioned in the query $Q$ have been queued on level zero (line 1), a loop is executed terminating only after a sufficient amount of results have been found or if the time limit was reached (lines 3-24). In each iteration, the URI to retrieve data from next is read from the queue, starting on level zero, until an URI is found that has not been retrieved before (lines 4-9). If no more URIs are in the queue, request processing is cancelled (lines 10-12). Otherwise, the data retrieved from the current URI is parsed into a set of RDF statements (line 13), the URI is marked visited as well. For every statement found, its subject is added to the queue on the subsequent level and the object only if it is also an URI (lines 16-18). Now the query $Q$ is evaluated on the entirety of the statements retrieved so far, and the results are sent to the client (lines 20-23). The algorithms iterates over the queue until either no more URIs queued on any level, or until time

or result limits have been reached. It should be noted, that an implementation should use multiple concurrent processes in order to retrieve RDF from more than one server at a time, this has been omitted for brevity.

Algorithm 1 Retrieval

**Require:** Query $Q$, Time limit $L_t$, Result limit $L_r$

1: $queue_0 \leftarrow \text{mentionedUris}(Q)$, $resultSetSize \leftarrow 0$, $level \leftarrow 0$
2: $localStatements \leftarrow \{\}$
3: repeat
4: repeat
5: $curUri \leftarrow \text{poll}(queue_{level})$
6: if $empty(curUri)$ then
7: $level \leftarrow level + 1$
8: end if
9: until ($not(\text{empty}(curUri)) \land \text{not}(\text{retrieved}(curUri))) \lor empty(queue)$
10: if $empty(curUri)$ then
11: return
12: end if
13: $statementSet \leftarrow \text{parseRdf}(\text{retrieveUri}(curUri))$
14: for all $statement$ as $statementSet$ do
15: $queue_{level+1} \leftarrow queue_{level+1} \cup \text{subject}(statement)$
16: if $\text{isUri}(\text{object}(statement))$ then
17: $queue_{level+1} \leftarrow queue_{level+1} \cup \text{object}(statement)$
18: end if
19: end for
20: $localStatements \leftarrow localStatements \cup statementSet$
21: $resultSet \leftarrow \text{evaluateQuery}(Q, localStatements)$
22: $resultSetSize \leftarrow \text{count}(resultSet)$
23: print $resultSet$
24: until $resultSetSize \geq L_r \lor \text{elapsedTime()} \geq L_t$

4 Evaluation

We have implemented the aforementioned approach for processing queries on Linked Data as a prototype using the Java programming language. The OpenRDF Sesame framework handles RDF parsing and local SPARQL query processing. The prototype provides an HTTP API, over which a client can send SPARQL queries, and as well as result and processing time limits. Our algorithm is then executed with a configurable number of concurrent threads. For this evaluation, the implementation was extended to provide status information at regular time intervals. In particular, the following status variables were recorded:

- **Processing Time** – The time since query processing was started
– **Search Level** – The level of the queue on which URLs are resolved at the moment
– **Result Count** – The number of bindings available so far from the union of the retrieved RDF graphs.
– **Resolved URIs** – The number of URLs resolved so far

Following the assumptions made when designing the query processing algorithm, we have selected two queries for evaluation to be performed on graphs currently available in the Linked Data cloud\(^2\). For all evaluation queries, the time limit was set to 60 seconds, and the result limit to 100 bindings. The first query is given in Listing 1.2, it retrieves the URIs and the names of all people that claim to know a particular person in their FOAF profiles. A FOAF profile is a person’s entry for the Friend-Of-A-Friend social semantic web \(^2\). The query was evaluated using our prototype, and the status variables are plotted in Fig. 3:

The x-axis denotes the time passed since the query was issued. The y-axis on the left shows the amount of results found as well as the queue level that is currently being processed. The right y-axis scales to the amount of URIs resolved. We made the following observations: Within the first five seconds, 12 bindings were already available and sent to the client. After 15 seconds, 15 bindings were available, and even though 150 more URIs were resolved in the following time, no additional bindings were found, even though the second queue level was fully dereferenced after 33 seconds. This could be due to the limited number of statements within the FOAF files, and the fact that this query explicitly queries the conceptual neighborhood of a particular profile.

```sparql
SELECT DISTINCT ?p ?n WHERE {
  ?p foaf:knows <http://bbfish.net/people/henry/card#me> .
} ORDER BY ?n
```

**Listing 1.2. SPARQL query ‘who’ (prefix declarations omitted)**

The second query given in Listing 1.3 (from the SQUIN example queries\(^3\)) retrieves the unemployment rates for all cities with an university in the German federal state of Lower Saxony, using the DBpedia project data, which is also published as Linked Data\(^4\). The progress status plot shown (see Fig. 4) shows very different results to those of the first query: No result bindings at all

\(^{2}\) As of August 12th, 2010
\(^{3}\) http://www.squin.org
\(^{4}\) http://www.dbpedia.org
have been produced, even though over 350 URIs were resolved, and the breadth
search arrived on the second queue level after only four seconds. This is con-
sidered to be mainly due to the fact that many DBpedia graphs contain huge
amounts of references, which cannot be resolved in reasonable time.

SELECT DISTINCT ?cityName ?ur WHERE {
  ?u skos:subject <dbpedia:Category:Unive (...) Saxony> ;
  dbpedia:city ?c .
  ?c owl:sameAs [ rdfs:label ?cityName ;
    eurostat:unemployment_rate_total ?ur ]
}

Listing 1.3. SPARQL query 'unemp' (prefix declarations omitted)

4.1 Complexity Considerations

As any remote lookup operations are sure to be orders of magnitude more ex-
pensive than in-memory operations, we assign the cost model of our query pro-
cessing approach directly to the amount of URIs resolved. In the average case,
we denote the average number of references to unknown URLs with \( r \). If a query
contains only \( u \) URIs to start the resolution process on, the expected amount of
URLs that have to be resolved (denoted by the function $a(u, r, l)$) for a breadth search level $l$ is:

$$a(u, r, l) := u \times r^l$$

Thus, for the entire query processing operation, the amount of URLs to resolve sums up to:

$$\sum_{l=0}^{L} a(u, r, l)$$

As we have established before, the average RDF graph published using Linked Data can be expected to contain around 100 references to other URIs. If we assume the query to contain three references to an URI, and two levels are to be resolved in order to generate all available bindings, this evaluates to 300303 retrieval operations.

Obviously this amount of resolution operations cannot be performed by current computers using current Internet connections in the time required to support an interactive application. However, if the first levels of resolution already generate sufficient bindings to display a satisfactory result to the user, the vast amount of possible leads may be discarded without resolving them. Alternatively, if the environment allows the assumption of a far lesser average number of references by RDF graph, more levels of the breadth search can be resolved in a limited time frame.
4.2 Demonstration Application

In order to test the applicability of our approach to the interactive Linked Data applications it was designed for, we have developed a small demonstration application, the “Inverse FOAF lookup”, which is available online. The application displays a list of profile documents, in which their creators expressed to know a particular person. This application is implemented as a HTML-/JavaScript AJAX-Application and is executed within the environment of a web browser. Users can enter a Web ID, which serves as both an identifier and access path for a person publishing a personal profile as Linked Data. Using this identifier, the query given in Listing 1.4 is constructed by replacing $REQ\_URI$ with the user’s Web ID. The constructed query is then sent over an asynchronous HTTP request to our query processing service, where evaluation starts, and intermediate results start being sent back to the browser. Contrary to common AJAX development, the application retrieves intermediate results from the service as they become available, and then immediately displays them to the user, enabling quick feedback and immediate continuation of browsing, should an interesting result be found.

Listing 1.4. SPARQL query used in the demonstration application (prefix declarations omitted)

```
SELECT DISTINCT ?uri ?name WHERE {
  ?uri foaf:knows $REQ\_URI$ .
  ?uri foaf:name ?name .
} ORDER BY ?name LIMIT 100
```

5 Conclusion and Further Work

We have identified the need for a quick and thorough retrieval mechanism for the Linked Data cloud. We then introduced our approach of a query processing mechanism using breadth-first search and the serving of intermediate results to the client system. This idea was mainly based on the assumption of only relatively simple queries being used, and that the average amount of links to other graphs would be limited. Our evaluation of a prototypical implementation of the described concept showed promising results for simple queries on Linked Data graphs with limited connectivity. We have used our prototype to support a simple demonstration application crawling the FOAF network for back-references to a particular profile.

http://idqp.u0d.de
The fitness of our approach for general-purpose query processing on Linked Data graphs can therefore not be confirmed, even if the performance on the social Semantic Web was acceptable. A simple extension to the approach presented here would be the introduction of a Least-Recently-Used cache, as some graphs such as the FOAF vocabulary are referenced in nearly every graph, and have to be retrieved again for every query processing run.

The challenge of quickly and efficiently retrieving data from the Linked Data cloud can be largely reduced to the problem of identifying the graphs, that contain triples which contribute query result. This could be achieved by using an heuristic reducing the amount of graphs to be reduced while sacrificing thoroughness. Another possible method would be a hybrid combination of breadth-first and depth-first search methods, starting with breadth-first to achieve local thoroughness, and continuing with selective depth-first execution should the queue size exceed a certain threshold.

Acknowledgments

The author would like to thank Ralf Heese and Olaf Hartig for their very helpful feedback and discussions.

This work has been partially supported by the “DigiPolis” project funded by the German Federal Ministry of Education and Research (BMBF) under the grant number 03WK07B.

References


