Processing Complex RDF Queries over P2P Networks

Felix Heine and Matthias Hovestadt and Odej Kao

2005
Processing Complex RDF Queries over P2P Networks

Felix Heine  
Paderborn Center for Parallel Computing (PC²)  
Paderborn University  
fh@upb.de

Matthias Hovestadt  
Paderborn Center for Parallel Computing (PC²)  
Paderborn University  
maho@upb.de

Odej Kao  
Paderborn Center for Parallel Computing (PC²)  
Paderborn University  
okao@upb.de

ABSTRACT
In large-scale distributed systems, information is typically generated decentralized. However, for many applications it is desirable to have a unified view on this knowledge, allowing to reason about it and to query it without regarding the heterogeneity of the underlying systems. In this context, two main requirements have to be fulfilled. On the one hand, a retrieval system has to be semantically rich, in order to be able to cope with and mediate between different schemas, and on the other hand it has to be scalable to large numbers of information sources. The dynamic nature of information makes the problem even worse.

Within this paper, we propose a solution to this problem. We describe a DHT-based peer-2-peer network storing knowledge in the form of RDF triples. The query evaluation algorithm allows to use arbitrary query patterns, and evaluates the query with respect to taxonomical reasoning. Thus the system combines expressivity and scalability. Although we describe the whole system, the focus of this paper is the query evaluation.

The system is generic by nature and suitable for numerous different applications. We describe an example application stemming from the Semantic Grid. ¹

Categories and Subject Descriptors
H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—Query formulation, Search process, Selection process; E.1 [Data Structures]: Distributed data structure

General Terms
Algorithms, Performance, Reliability

¹Partially supported by the EU within the 6th Framework Programme under contract 001907 “Dynamically Evolving, Large Scale Information Systems” (DELIS).

Keywords
P2P, RDF, RDFS, Query Processing, Semantic Web, Grid

1. INTRODUCTION
In large-scale distributed systems, like Grid systems, the Web and the forthcoming Semantic Web, but also in Services Computing, storage and evaluation of machine readable information is a key issue. Of special interest are methods to evaluate large, distributed collections of information, without downloading and storing them to stand-alone reasoners.

As an example, Grids use numerous distributed information services which monitor and announce the availability and the features of the Grid’s resources. A resource broker should be able to query these sources in a consistent, efficient, and transparent way. Another example is the Semantic Web [3]. Its vision is to make the huge information resources of the Web available for machine-driven evaluation. Currently, only humans are able to combine the information stored in the Web and draw conclusions from it. The first step of converting the information into a machine-readable format is already on its way. However, efficient means to query this information are missing.

Thus we are looking for a system which can dynamically integrate heterogeneous information sources and which allows to reason about the information. It should present the user a query mechanism which transparently retrieves information combined from various sources.

Two main challenges demand for a trade off: on the one hand, the system should provide a highly expressive logic with sophisticated reasoning features. On the other hand it should scale efficiently to large amounts of information and complex queries. We found the Resource Description Framework (RDF, [11]) in combination with RDF Schema (RDFS, [4]) to be a good basis to achieve this goal. RDF became a W3C recommendation in February 2004 and is expected to be widely used in the semantic web. RDF stores information in triples consisting of a subject, a predicate, and an object. The semantics underlying these triples are defined by a model theory and a set of entailment rules, which define how triples can be generated that follow logically from a set of existing ones [9]. RDFS provides a way to describe taxonomies of classes and properties, so that a user can refer to some vocabulary when encoding information via RDF triples.

In this paper, we describe a peer-2-peer (p2p) network which enables each participating node to publish information in the form of RDF triples. The vocabulary is described using RDFS. Each node can individually extend the vocab-
ularly as needed. The information is distributed in a well defined way over the network and a selected subset of the RDF entailment rules is used to compute conclusions from the information. A query algorithm is presented which allows to pose queries in the form of RDF graphs to ask the p2p network for the existence of a certain pattern of RDF triples. We show that even if the query contains some unspecified triples, efficient query evaluation is possible.

The system is general-purpose. It can be used as a storage for any kind of information, as long as it can be converted to RDF. The query mechanism already allows a huge degree of flexibility, making a wide range of applications possible. We give an example from Grid computing to motivate the usage of RDF and RDFS.

This paper is organized as follows: After mentioning related work, we describe a introductory example in section 3. Then we explain our architecture and how we distribute the information over the p2p network. This distributed knowledge is the basis for our query evaluation algorithm, which is described in section 5. After that, we present and discuss simulation-based results. We conclude after an outlook to further work.

2. RELATED WORK

The general idea of the semantic web has been described by Tim Berners-Lee et al. in [3]. They paint the vision of a web where information can be automatically processed by software. The Resource Description Framework (RDF) together with RDF Schema [11, 4] is one of the upcoming standards which will help to make this vision a reality.

In [9] model theoretic semantics have been defined for RDF and RDFS (in short RDF/S). A set of rules have been shown to implement these semantics in a forward-chaining manner. In [5], Broekstra and Kampman explained that forward-chaining is indeed a viable approach to RDFS inference. If the dependencies between the rules are regarded, a huge number of tests can be avoided, resulting in a scalable algorithm. Their work targets on non-distributed RDF data in a single database.

In [6], dynamic query execution for schema-based p2p networks in the context of the Edutella project [13] is described. This work focuses on dynamic query planning and execution. Queries are evaluated in a distributed fashion; the optimizer tries to evaluate operators local to the data. However, the data model is different from ours, so that the work is not directly comparable.

In [12], Kokkinidis and Christophides describe middleware based on P2P for evaluating queries in the RDF Query Language (RQL) using RDF Schema knowledge. They focus on the construction and optimization of query plans. Their basic approach is different from ours as they do not use a Distributed Hash Table (DHT) based P2P network. Instead, they use so-called semantic overlay networks, which are groups of peers sharing the same schema.

The idea of using URIs as the key to distribute information over an DHT-based P2P network has been described in several papers. We have used it in [10] to distribute knowledge based on Description Logics. It has been used in the GridVine project [1] and RDFPeers [7] to distribute RDF triples. The distribution of triples used in this paper is a combination of these ideas including the distribution of taxonomies over DHT networks.

3. EXAMPLE FROM GRIDS

In this section, we describe a sample application of matching resources in large scale Grid systems. One of the major aspects of Grids is the virtualization of resources. This means, that users do not have hard-coded links to the resources they use for their jobs. The job has a requirement profile describing the type of resources needed to run the job. Each provider describes its resources, in order to allow a resource discovery mechanism to find matches for the given requirements.

Although we focus on traditional Grid resources like high performance computers or large scale storage resources, the system is open to match anything which can be described using RDF.

We now describe how we use RDF in combination with RDFS to describe resources and background knowledge about resources. We introduce our usage of RDF by examples.

Consider a cluster named sfb having 4 nodes running the Debian distribution of the Linux operating system. The cluster nodes are equipped with Itanium2 processors. Part of the RDF graph describing this system is shown in figure 1. Both the operating system and the processor are described by blank nodes; we are not interested in these entities themselves but rather in their type and their properties.

![Figure 1: RDF graph for the sfb cluster.](image1)

A simple query graph is shown in figure 2. It looks similar to the first graph, but there are some differences. First, the entity which is queried is denoted with a variable. Second, the type of the operating system is specified as pc2:Linux compared to pc2:Debian in the resource description. Third, the type of the processor is specified as a combination of two types: Intel and 64Bit. Thus, the query is more generic than the description. The entire query graph should be read as a template which is to be matched with a sub-graph of the resource description graph.

![Figure 2: Query graph.](image2)

In order to be able to match this query with the resource description, we need additional schema knowledge, which we encode in RDF Schema. It encompasses information like
“pc2:Debian is a sub-class of pc2:Linux”. Figure 3 shows how this knowledge is encoded in RDF/S.

\[
\begin{align*}
\text{pc2:Debian} & \text{rdfs:subClassOf} \text{pc2:Linux} \\
\text{pc2:Itanium2} & \text{rdfs:subClassOf} \text{pc2:64Bit} \\
\text{pc2:Itanium2} & \text{rdfs:subClassOf} \text{pc2:Intel}
\end{align*}
\]

Figure 3: Graph for schema knowledge.

The schema knowledge is integrated in the query process by applying the RDFS entailment rules [9]. These rules are executed in a forward-chaining manner, thereby generating new triples. For instance, entailment rule “rdfs9” will generate new rdf:type edges. In the resulting graph, shown in figure 4, a sub-graph is searched which matches the query graph.

\[
\begin{align*}
\text{pc2:Cluster} & \text{rdf:type} \text{pc2:kernelVersion} \\
\text{pc2:hasProcessor} & \text{pc2:sfb} \\
\text{pc2:hasOS} & \text{pc2:Itanium2} \\
\text{pc2:Debian} & \text{pc2:64Bit} \\
\text{pc2:Linux}
\end{align*}
\]

Figure 4: Union of local and schema knowledge.

The important point is that the schema knowledge and the knowledge about the resources can be located on different nodes, so that the query answer can only be computed by combining these distributed RDF graphs.

4. ARCHITECTURE

This section describes our scenario and introduces how we distribute the knowledge over the network. This will be the basis for the query evaluation algorithm described in the following section.

4.1 Scenario

We assume to have n nodes participating in the p2p network. All of them have some local knowledge stored as RDF triples. They also have local schema knowledge stored as RDF Schema triples. The schema knowledge does not need to be the same for every node. In fact, we are convinced that it is impossible to ensure synchronization between schema knowledge in large world-wide distributed environments or to restrict the schema to a single common standard. Moreover, it is desirable to allow each node to add locally needed schema information on the fly. If new entities need to be described, new classifications may become necessary. Waiting for a new version of some standard schema does not solve this problem.

However, we assume that there is an ontology which serves as a common schema, at least for some subsets of the nodes. This ontology will be the basis which can be extended locally. Additional schema knowledge may be stored to allow translation from one ontology to the other. Without such common understanding, no interoperability would be possible.

\[
\begin{align*}
\text{Node 1:} & \text{SK, } \text{LK,} \\
\text{Node n:} & \text{SK, } \text{LK,} \\
\text{DHT-based P2P network}
\end{align*}
\]

Figure 5: Architecture.

The general architecture is shown in figure 5. All nodes are connected via a p2p network, which runs a distributed hash table (DHT) algorithm [2]. Each node i has stored two sets of RDF triples, the schema knowledge SKi and the local knowledge LKi.

The desired result is to put all this knowledge virtually in one huge pool, apply RDFS entailment rules to this pool and evaluate queries with respect to the union of the knowledge. This approach is very beneficial, as overlaps in the schema knowledge are used to build bridges between different schemas used by different nodes. A query is formulated as a pattern consisting of multiple triples where parts of the URI references and labels are replaced by variables. We call the RDF graph resulting from the union of the local knowledge and appliance of the entailment rules the model graph, while we call the query pattern the query graph.

We support a selected subset of the RDFS entailment rules [9]. In particular, we support the rules rdfs7 and rdfs9, which state that instances of classes and properties are also instances of super-classes and super-properties, respectively. The rules rdfs5 and rdfs11, which implement the transitivity of the subClassOf and subPropertyOf predicates, are supported implicitly. The triples are not generated, but the rules rdfs7 and rdfs9 implicitly propagate instances or pairs of instances to every class / property in the transitive closure of the subClassOf / subPropertyOf relation.

Note that we assume blank node labels to be unique in the network. This can simply be achieved by adding a node identifier to the label. Thus we can assure that we can join the triple-sets without caring about the blank node labels.

Both the model and the query graph are directed graphs. The labels of the model graph can be URI references or XML literal values, or blank node labels. For the following discus-
sion, we do not have to differentiate between URI references and XML literals, so we define the set of labels \( \mathcal{L} \) which contains both types of entities. The set of blank labels is denoted by \( \mathcal{B} \). Thus each vertex is labelled with an element of either \( \mathcal{L} \) or \( \mathcal{B} \). An edge of an RDF graph is a blank label, so only elements of \( \mathcal{L} \) are allowed. RDF permits multi-edges, i.e., more than one edge between a pair of nodes, and no stand-alone vertices are allowed.

The graph can be described as a triple set

\[
T_M \subseteq (\mathcal{L} \cup \mathcal{B}) \times \mathcal{L} \times (\mathcal{L} \cup \mathcal{B})
\]

The query graph is defined very similar. However, if we use variables from a set \( \mathcal{V} \) of variables, and we allow edges to be labelled with variables, the query graph in triple representation is

\[
T_Q \subseteq (\mathcal{L} \cup \mathcal{V}) \times (\mathcal{L} \cup \mathcal{V}) \times (\mathcal{L} \cup \mathcal{V})
\]

For convenience, we denote the set of variables occurring in \( T_Q \) by \( \mathcal{V}_Q \), and the set of literals occurring in \( T_Q \) by \( \mathcal{L}_Q \). The sets \( \mathcal{L}_M \) and \( \mathcal{B}_M \) are defined analogously.

The desired semantics for our query evaluation are as follows: given a model graph \( T_M \) and a query graph \( T_Q \), find every matching for the variables occurring in \( T_Q \) to the set of blank nodes and literals occurring in \( T_M \), such that for each triple in \( T_Q \) there is a matching triple in \( T_M \). Thus we search for mappings

\[
R: \mathcal{V}_Q \rightarrow \mathcal{L}_M \cup \mathcal{B}_M
\]

such that for every triple \( (s, p, o) \in T_Q \) there is a triple \( (s', p', o') \in T_M \) such that:

\[
\begin{align*}
    s \in \mathcal{V}_Q & \Rightarrow s' = R(s) & s \in \mathcal{L}_Q & \Rightarrow s' = s \\
    p \in \mathcal{V}_Q & \Rightarrow p' = R(p) & p \in \mathcal{L}_Q & \Rightarrow p' = p \\
    o \in \mathcal{V}_Q & \Rightarrow o' = R(o) & o \in \mathcal{L}_Q & \Rightarrow o' = o
\end{align*}
\]

Note that this definition includes the possibility to match two different variables to the same value, as we do not insist on \( R \) being an injective function, which makes the problem a bit different to the subgraph isomorphism problem (see [16]). We impose two restrictions upon \( T_Q \). First, we expect it to be connected (not strongly connected). This is natural, as we can break the query evaluation of an unconnected query graph in multiple evaluations of the connected components. The result set of the whole query is determined by enumerating every combination of the results for the connected components. Second, we expect that there exists at least one triple in \( T_Q \) having at least one labelled element. One of these labels will serve as a starting point for the query evaluation.

### 4.2 Node Architecture

Within this section, we briefly look at the building blocks of each node of the network. See figure 6 for an overview. Each node takes care of several local information sources, which can be simple RDF or RDFS files. However, these sources can also be interfaces to other systems like Grid information providers. The node collects the RDF triples in a local storage. No reasoning is done with this knowledge.

The triples are directly sent to the relevant nodes in the network, as described in the next section.

The triples which are received via the p2p network are stored in a separate container. The evaluation of the RDFS rules is done with respect to the triples stored in this container, which again leads to the generation and distribution of new triples.

The query evaluation algorithm uses the network knowledge of the local node, and queries the network for additional triples needed to answer this query.

### 4.3 Distribution of the Triples

In order to be able to query the knowledge, we have to have a way to find relevant triples for the query, as we do not want to query every node in the triple. For this purpose, we use a structured peer-to-peer network which implements a distributed hash table. There are different DHT approaches available, all having some kind of lookup mechanism in common. This lookup mechanism enables the user to determine a specific node which is responsible to store data for a certain key.

In our scenario, we use the URI references respectively the XML literals as keys. We store each triple three times, indexing by the subject, predicate, and object. Thus each node sends out its own triples to the responsible nodes. The target nodes store the triples and possibly generate new triples by applying the entailment rules. These new triples have to be sent out to other nodes.

Thus, after finishing this process, the whole model graph is accessible in a well-defined way over the DHT network. See [10] for a discussion about the length of this process and the message load generated by it. Although this discussion focussed on taxonomies generated from DL reasoners, it applies as well to the RDFS taxonomies we are using here.

Now there are several ways to retrieve triples from the network. To retrieve a set of triples, at least one part of the triples must be fixed. We use this part as a key to the DHT network, retrieving all triples with this value.

We define three functions, **getBySubject**, **getByPredicate** and **getByObject**, which we use to retrieve sets of triples. As an example we describe the **getBySubject** function. It gets a label as input and retrieves all triples from the network where the subject equals this label. It calls the lookup operation of the DHT network to retrieve the network node which stores these triples. So the execution time of these functions is determined by the time the lookup operation takes and the transfer time of the result set.

Additionally, we define functions to retrieve statistical information. They work similar, however they retrieve only...
5. QUERY EVALUATION

Our algorithm for evaluating the queries works in two phases. In the first phase, we determine candidate sets for each of the triples in the query graph, as well as for the variables. The candidate sets for the variables and for the triples are mutually dependent, thus we have a refinement procedure which successively removes candidates from both sets which are not suitable. In the second phase, the remaining candidates for the variables are tested in several combinations to reveal the matches. In general, there are exponentially many combinations. However, we employ again a refinement procedure which reduces the number of possible combinations.

5.1 Determination of Candidate Sets

The task of this phase is to identify parts of the model graph which are relevant for the query. The main focus of the algorithm is to reduce network load during this phase. This means, that as less nodes as possible are contacted, and as less data as possible is sent around. We evaluate and compare different strategies.

The main idea behind the algorithm is to determine how much candidates are expected for each triple and to iteratively choose the triple with the smallest expected candidate set. At each step, the algorithm maintains a set of candidates for each triple, denoted \( C_T(t) \), \( t \in T_Q \) and a set of candidates for each variable denoted \( C_V(v) \), \( v \in V_Q \). Candidate sets may be undefined. As a short-cut, we will write \( C_V(v) = \Delta \) if the candidate set for \( v \) is not defined. We furthermore define \( |C_V(v)| := \infty \) iff \( C_V(v) = \Delta \). As it will simplify the algorithms presented later, we further define the candidate set of a fixed value (either literal or URI reference) to be the one-element set containing that value: \( C_V(x) = \{ x \} \) iff \( x \in L \).

Then some network communication will be used to retrieve the candidate sets, leading to new estimates for the other triples. We use the notion of the specification grade of a triple to see where we expect the smallest communication overhead. If you look at the way we distribute the triples, we can either use the subject, the predicate, or the object to retrieve the candidates. Each of these can either be a variable or a fixed value. If it is a fixed value, we have to do a single lookup to retrieve the candidate set. If it is a variable, the number of lookups is determined by the current number of candidates for this variable. If there are no candidates for the variable so far, we cannot use it to retrieve a candidate set.

Thus we define the specification grade of a triple’s element as follows:

\[
sgI(x) = \begin{cases} 
|C_V(x)| & : C_V(x) \neq \Delta \\
\infty & : C_V(x) = \Delta 
\end{cases}
\]

Due to our above definition, this can be written short-hand as \( sgI(x) = |C_V(x)| \). The specification grade for a triple is the minimum specification grade of its elements:

\[
sgI(s, p, o) = \min(sgI(s), sgI(p), sgI(o))
\]

The idea behind this definition is that the specification grade determines the number of lookup operations needed. Thus we can write down the algorithm:

```plaintext
function candidates(T_Q, T_M)
  set each \( C_T(t) \) and \( C_V(v) \) to \( \Delta \)
  while there is some undefined \( C_T(t) \)
    determine some triple \( t = (s, p, o) \) where
    • \( C_T(t) = \Delta \), and
    • \( sgI(t) \leq sgI(t') \) ∀′ with \( C_T(t) \) undefined.
    if \( sgI(t) = sgI(s) \)
      \( C_T(t) := \cup_{t \in C_V(v)} \) getBySubject(x)
      \( C_T(t) := \{ (s, p, o) \in C_T(t) : p \in C_V(p), o \in C_V(o) \} \)
    else
      similar code for predicate and object
      return error
    end if
  end while
  return ok
end function
```

The heart of the algorithm is the refinement procedure. There are two ways of refinement. First, we can look at a variable’s candidate set. We compare it with the candidate sets for each triple where this variable occurs. If a candidate does not occur within the triple candidate set, it has to be removed from the variable candidate set. The other way around, we look at the candidate set for a triple and remove any candidates where there is some value not within the matching variable’s candidate set. We always keep track of the set of changed variables \( V' \) and changed triples \( T' \), so that we do not have to check every set.

```plaintext
function refine(C_T, C_V, T, V)
  while \( V \neq \emptyset \) or \( T \neq \emptyset \)
    for each \( t = (s, p, o) \in T \)
      if \( s \in V \)
        \( C_V(s) := C_V(s) \cap \text{subject}(C_T(t)) \)
        if \( C_V(s) \) has been changed
          \( V := V \cup \{ s \} \)
        end if
      end if
      similar code for predicate and object
      \( T := T \setminus \{ t \} \)
    end for
    for each \( v \in V \)
      for each \( t \in T_Q \)
        if subject(t) = v
          \( C_T(t) := \{ (s', p', o') \in C_T(t) : s' \in C_V(v) \} \)
        end if
        if \( C_T(t) \) has been changed
          \( T := T \cup \{ t \} \)
        end if
      end if
      similar code for predicate and object
    end for
    \( V := V \setminus \{ v \} \)
  end while
  return ok
end function
```
The crucial question is how we can further reduce the network load. The refinement procedure is uncritical, as it works completely local. Thus we have to look at the order in which the triple candidates are retrieved from the network.

The definition of the specification grade as given above ensures a minimal number of lookup operations in the current step. However, it can lead to a large number of candidate sets for the triple, leading to both a high bandwidth consumption and a large number of lookups in further steps. Thus we evaluated an alternative definition of \( s_g1 \), called \( s_g2 \). The idea is to take into account the number of candidates retrieved when using some triple.

We have to use the \( \text{cntBySubject} \), \( \text{cntByPredicate} \), and \( \text{cntByObject} \) functions to retrieve the expected number of candidates for some query. Thus we have more lookup operations. However, these lead to smaller candidate sets as a better sequence of triples is chosen. The new definition of the specification grade is as follows. We have to define three different functions, referencing to the three elements of a triple. We only describe the subject function \( s_g1, s_g2 \) and \( s_g0 \) are analogous.

\[
sg(x) = \begin{cases} 
\sum_{v \in C_V(x)} \text{cntBySubject}(s) : C_V(x) \neq \Delta \\
\infty : C_V(x) = \Delta 
\end{cases}
\]

The specification grade of a triple is now defined as

\[
sg2((s, p, o)) = \min(sg1(s), sg2(p), sg2(o))
\]

Experiments have shown, that this version results in much faster query evaluation when used for stand-alone RDF querying. In section 6 we present simulation-based results for the network traffic generated by the two versions. In addition, we evaluated a third version \( s_g3 \) which enhances \( s_g2 \) by caching the results of the \( \text{cntBySubject} \) etc. functions, which greatly reduces the number of lookups.

### 5.2 Final Evaluation

After having retrieved candidate sets for all triples and variables, we have to do the final evaluation to retrieve matches for the query. This is done completely local. However, it can be computationally expensive. In general, every combination of candidates for the variables has to be considered and tested, which are exponentially many. In order to reduce the amount of combinations, we apply the refinement procedure each time we pick a candidate for some variable. This will strike out any candidates for the other variables which are in contradiction with the current pick. In this way, we avoid completely to run into unsuccessful branches. So here is the algorithm:

```java
function evaluate\( (C_T, C_V) \)
if there is some \( v \) with \( C_V(v) \geq 1 \)
pick some \( v \) with \( C_V(v) > 1 \)
for each \( x \in C_V(v) \)
\( C'_V := C_V, C'_T := C_T \)
\( C'_V(v) := \{x\} \)
refine\( (C'_T, C'_V, \emptyset, \{v\}) \)
evaluate\( (C'_T, C'_V) \)
end for
else
the candidate sets for the variables form a match
end if
end function
```

The run time of this algorithm is determined by the run time of the refinement procedure, which is applied after each choice we make for some variable. Thus it is mandatory to have an efficient way to store the candidate sets. Additionally, searching for a candidate and the removal operation have to be highly efficient. Finally, a fast copy operation for the candidate sets is needed. We employ hash sets for the candidate sets which allow an efficient implementation of these operations.

### 6. RESULTS

In this section, we present results in order to evaluate the performance of our algorithm. We are mainly interested in two different questions, related to the two phases of the algorithm. First, we want to know how efficient the first phase works with respect to network traffic. Second, we are interested in the performance of the local part of the algorithm.

To answer the first question, we have simulated the algorithm and counted two different metrics. The first is the number of lookup operations needed during the algorithm. Each time we retrieve information related to a DHT key, we have to perform a lookup operation, which is typically costly. E.g., when using the Chord DHT [15], a lookup operation is resolved by forwarding a message O(\( \log n \)) hops through the network. Thus we aim to reduce the number of lookups.

On the other hand, large result sets will lead to a slow transmission over the network, depending on the bandwidth. In order to evaluate the three version of the \( s_g1 \) function described in section 5, we generated random taxonomies of classes and properties, filled them with instances and properties, and generated random query graphs. We investigated different sizes of the model graph, in combination with various query graphs. The queries had an increasing number of variables leading to larger candidate sets.

In figure 7 the number of lookup operations performed when using the different \( s_g1 \) versions is shown. On the x axis, two different model graph sizes are shown, each in combination with 10 different query graphs. The query graphs are ordered by an increasing number of variables. The sum of the sizes of all messages (number of triples) is shown in figure 8. One can see, that the first version of \( s_g1 \) leads to large messages while keeping a low number of lookup operations, while the second version performs exactly opposite. It reduces the message sizes. However, the number of lookup operations is increased significantly. The third version is an ideal combination resulting both in small message sizes and few lookup operations.

Regarding the performance of the second phase of the query evaluation, we ran different queries over a RDF database consisting of roughly 350,000 triples. We have tried several queries with different numbers of query triples and variables, see figure 9.

<table>
<thead>
<tr>
<th>Query</th>
<th>Triples</th>
<th>Variables</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>8</td>
<td>1417</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>11</td>
<td>1417</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>10</td>
<td>9879</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>13</td>
<td>9879</td>
</tr>
</tbody>
</table>

Figure 9: Characteristics of test queries.
Figure 7: Number of lookup operations.

Figure 8: Message size.

We compared our algorithm with Sesame², a well-known semantic web toolkit which allows local in-memory and database grounded RDF processing. We used the in-memory representation to get comparable results. Table 10 lists our results. The times are pure query processing without time to load the data into memory.

<table>
<thead>
<tr>
<th>Query</th>
<th>time</th>
<th>Sesame time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt; 1 sec</td>
<td>5 sec</td>
</tr>
<tr>
<td>2</td>
<td>&lt; 1 sec</td>
<td>7 sec</td>
</tr>
<tr>
<td>3</td>
<td>2 sec</td>
<td>47 sec</td>
</tr>
<tr>
<td>4</td>
<td>3 sec</td>
<td>59 sec</td>
</tr>
</tbody>
</table>

Figure 10: Local query evaluation time.

These initial results show that our algorithm is quite fast, which is mainly due to the speed of the refinement procedure and the data structure used here.

²see www.open-rdf.org, we used version 1.1.3

7. FUTURE WORK

The described algorithm is a good basis to do distributed RDF/S entailment in large scenarios. However, it can be improved in many ways. We will list here some of the topics:

- **Full RDFS Semantics.** We support the most important RDFS rules which are concerned with the class and property hierarchies. RDFS also supports reasoning over the domain and range of a property, and other topics like ContainerMembershipProperties etc. We aim to support full RDFS entailment in future versions.

- **Beyond RDFS Semantics.** For some applications, RDF Schema is sometimes too restrictive. E.g., it would be very interesting to be able to express transitive properties. An expressivity somewhere between OWL-DL and RDFS would be desirable.

- **Better Load Balancing.** As we distribute the knowledge according to the URIs of the classes and proper-
ties, and we propagate the information along the subclass / sub-property relationship, the quality of the load balancing between the peers depends on the ratio of the number of nodes and number of concepts. A small number of concepts respectively properties together with a large number of nodes lead to imbalances w.r.t. the storage load. Note that this problem goes beyond the one described in [7] as we also employ forward-chaining of RDFS rules, which leads to a huge triple load for the nodes which store triples about a very generic concept or property.

8. CONCLUSION

In this paper, we tackled the problem of querying large distributed collections of knowledge in an efficient and scalable way. We have described a novel algorithm to harness large scale collections of RDF data, where different schemas, encoded in RDFS, are used to structure the data. The triples are distributed over a structured DHT-style p2p network.

A selected subset of the RDFS entailment rules is applied to the original RDF data from the nodes, generating derived triples. The client of the system can formulate queries in terms of arbitrary RDF graphs. The queries are matched against the original RDF data together with the derived triples. The RDF data is split up into local knowledge and global schema knowledge. A match for a specific query is a logical implication from the schema knowledge from all nodes together with the local knowledge from all nodes.

The presented algorithm is a step towards the ability to combine and harness large-scale distributed collections of semantically enhanced information. It goes beyond what is possible with former systems, as basic inference support is a way to take advantage of RDFS semantics without loading semantically enhanced information. It goes beyond what is RDFS data from many different sources into a single reasoner, which would limit the overall scalability. P2P networks have proven to be a suitable basic infrastructure for achieving our goals.

Besides the simulation-based results presented in this paper, we already have a working prototype of our system. It is based on the FreePastry implementation of the Pastry DHT network [14]. However, as we use the common API for p2p networks [8], we are flexible with respect to the underlying DHT.

9. REFERENCES


