Supporting DL Semantics in BabelPeers

Dominic Battré, Felix Heine, André Höing, and Odej Kao

2007
Supporting DL Semantics in BabelPeers

Dominic Battré, Felix Heine, André Höing, and Odej Kao

Technical University of Berlin, Berlin, Germany,
Complex and Distributed IT Systems,
{dominic.battre,felix.heine,andre.hoeing,odej.kao}@tu-berlin.de,
WWW home page: http://www.cit.tu-berlin.de

Abstract. Scalable storage and retrieval of semantically rich data is a necessary prerequisite for the envisioned Semantic Web as well as other large-scale distributed systems. The full potential of these systems can only be reached when using semantics like description logics that allow to infer implicit knowledge from the explicitly stated facts. However, the combination of these complex reasoning tasks with scalability has been difficult to achieve so far.

In this paper, we show how to use query rewriting techniques that incorporate TBox knowledge into the queries in combination with the query evaluation methods of our BabelPeers P2P based RDF datastore. The resulting system can evaluate queries with respect to semantics including transitive roles, inverse roles, and complex concept descriptions. It is yet scalable through the distribution of the load over the P2P network. Thus it achieves the goal to combine complex semantics with scalability.¹

1 Introduction

The vision of the Semantic Web [6] foresees huge amounts of machine-processable data to be available on the web. This opens up new possibilities for querying and mining the wealth of information available today on the web. In particular, the machine-processable nature of the data allows to automatically derive new information contained only implicitly in stated facts.

However, the question of how to store and query this data in an efficient and scalable way is still open. In this paper, we propose a new approach to this problem. It bases on our BabelPeers RDF store [12] that allows efficient storage and retrieval of RDF triples based on a DHT p2p network. However, we advance the BabelPeers system with a new reasoning approach.

This approach builds on the common assumption that the instance data (ABox) is much larger than the metadata (TBox). It involves rewriting the queries given as conjunctive queries into a set of multiple queries. In this approach, we were inspired by DL-Lite [10], however we go beyond what is possible with DL-Lite. Additionally, we target the question how to build good query

¹ Partially supported by the EU within the 6th Framework Programme under contract 001907 “Dynamically Evolving, Large Scale Information Systems” (DELIS).
plans for the resulting set of rewritten queries, and how to execute these query plans in the scope of a p2p RDF store.

Our specific goal is to show that query rewriting is an efficient solution for p2p based RDF stores even when more complicated semantics like transitive properties oder full existential quantification are involved (see [3]). We target an OWL sublanguage that is rich enough to support the Lehigh University Benchmark [11].

In the following section, we describe our BabelPeers system. After that, we define the supported logic $DL_{LH}$ in section 3 and review the query rewriting technique. The following section focusses on the efficient execution of rewritten queries in the context of BabelPeers. Performance evaluations are given in section 6. Finally, we describe related work and conclude.

2 The BabelPeers System

The goal of the BabelPeers system is to integrate knowledge originating from different sources, i.e. different nodes in the p2p network. Thus it is possible to get results from the union of all knowledge in the network, including knowledge derived via logical reasoning using pieces that originate from different peers. Even broadcasting a query to all nodes in the network and collecting the results could not deliver the same quality of results one gets if the knowledge is integrated. Thus we need to disseminate the knowledge to well-defined nodes in order to be able to find and access it efficiently during query processing.

We use the W3C standard resource description framework (RDF) for knowledge representation. RDF is a very flexible framework. Each piece of information is encoded as a triple consisting of subject, predicate, and object. These triples can be linked together in a world-wide unique manner, as URIs are used to identify resources. Thus putting together large amounts of RDF triples stemming from different sources typically results in a large graph that describes the overall knowledge.

For storing and integrating the triples, we use a p2p network based on distributed hash tables (DHT). In a DHT, each data item is associated with an identifier from an identifier space, e.g. $0, \ldots, 2^{128} - 1$. Each node in the network is responsible for a certain range of this identifier space. Every item to be stored is pre-distributed to the node responsible for the identifier of the item. For fault tolerance, items are additionally replicated over multiple nodes. In our case where data items are RDF triples, we disseminate each triple to three different nodes based on the hash values of its components subject, predicate, and object. Thus we can later access the triples even if only one of the components is known.

The basic form of RDF reasoning is RDF Schema (RDFS) reasoning. RDFS allows to build hierarchies of classes and properties. The reasoning rules essentially state that instances are propagated along these hierarchies towards more generic classes or properties.
For the RDF Schema reasoning, BabelPeers follows a forward chaining approach. This means that we generate and store instances of every new triple which follows from the RDF Schema rules. Our dissemination scheme has the advantage, that all triples needed to do this forward chaining will be located on the same node. Thus, after dissemination, we can run the reasoning process on each node locally, generating the implicit knowledge. However, these newly generated triples are then disseminated to the network to be accessible via the standard indices over subject, predicate, and object.

Updates and deletions are handled via a soft-state process. Thus, every triple carries an expiration time and is automatically removed from the network when it does not get refreshed. This means that the dissemination process runs periodically to keep the knowledge in the network up-to-date.

3 Foundations of $DL_{LH}$

In this section, we define the used logic called $DL_{LH}$ and introduce the query rewriting methods informally. For a more formal description, see [5].

3.1 Syntax

Our DL $DL_{LH}$ is defined as usual starting with the syntax, followed by model-theoretic semantics. The syntax for concept terms is as follows:

$$B := A \mid \exists R.A$$
$$C := B \mid B_1 \sqcap B_2$$

In this notation, $A$ denotes an atomic concept, $B$ denotes a basic concept, and $C$ denotes a complex concept. A knowledge base $\mathcal{K}$ consists of a TBox and an ABox, both containing a set of assertions. Using the above listed concept terms and $R$ to denote atomic roles, the assertions listed in table 1 are allowed in the TBox.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1 \equiv C_2$</td>
<td>equality of complex concepts</td>
</tr>
<tr>
<td>$C_1 \sqsubseteq C_2$</td>
<td>subsumption of complex concepts</td>
</tr>
<tr>
<td>$R_1 \sqsubseteq R_2$</td>
<td>subsumption of atomic roles</td>
</tr>
<tr>
<td>(transitive $R$)</td>
<td>transitive roles</td>
</tr>
<tr>
<td>(domain $R \ A$)</td>
<td>domain restriction</td>
</tr>
<tr>
<td>(range $R \ A$)</td>
<td>range restriction</td>
</tr>
<tr>
<td>(inverse $R_1 \ R_2$)</td>
<td>inverse of a role</td>
</tr>
</tbody>
</table>

Table 1. TBox assertions in $DL_{LH}$. 
In the ABox, concept and role membership assertions are allowed only for atomic concepts or roles, respectively. This is shown in table 2.

<table>
<thead>
<tr>
<th>A(a)</th>
<th>individual a belongs to concept A</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(a,b)</td>
<td>a is related to b via R</td>
</tr>
</tbody>
</table>

Table 2. ABox assertions in $DL_{LH}$.

A query is an expression of the form

$$q(x) \leftarrow \exists y : \text{conj}(x, y)$$

$x$ are the distinguished variables, $y$ are the non-distinguished variables, and $\text{conj}(x, y)$ is a conjunction of disjunctions of the form $(A_1 \lor \cdots \lor A_n)(z)$ or $(R_1 \lor \cdots \lor R_m)(z_2, z_3)$. $A_1$ to $A_n$ are atomic concepts, $R_1$ to $R_m$ are atomic roles, and $z, z_2, z_3$ are either individuals in $K$ or variables from $x$ or $y$.

Please note that although both queries and ABox assertions only allow for atomic concepts, the expressivity is not diminished due to the existence of concept equality assertions which enable the user to define named concepts that are then used in the queries or ABox assertions.

### 3.2 Semantics

We now define the semantics using the model-theoretic approach. An interpretation $I = (\Delta^I, \cdot^I)$ consists of a domain $\Delta^I$ and an interpretation function $\cdot^I$.

Each atomic concept and role is mapped to a subset of the domain or a set of pairs of domain elements, respectively. Each individual name is mapped to a distinct domain element (unique name assumption):

$$A^I \subseteq \Delta^I$$
$$R^I \subseteq \Delta^I \times \Delta^I$$
$$a^I \in \Delta^I$$
$$a \neq b \Rightarrow a^I \neq b^I$$

The semantics of the further constructors are defined as follows:

$$(\exists R.A)^I = \{ x \in \Delta^I \mid \exists y \in A^I : (x, y) \in R^I \}$$
$$(B_1 \cap B_2)^I = B_1^I \cap B_2^I$$

An interpretation is a model of some assertion in a knowledge base if the conditions given in table 3 hold.
As usual, a model of a knowledge base $K$ is an interpretation that is a model of each assertion in the knowledge base. A knowledge base is satisfiable if it has at least one model. A knowledge base $K$ logically implies an assertion $\alpha$ if all models of $K$ are also models of $\alpha$.

The interpretation $q^I$ of a query $q$ under an interpretation $I$ is the maximal set of tuples $c \in \Delta^I \times \cdots \times \Delta^I$ such that when we substitute the variables in $x$ with the constants in $c$, the formula $\exists y : \text{conj}(x, y)$ evaluates to true.

### 3.3 Normalization

Before starting the query evaluation, the knowledge base is normalized. This process is fairly standard. First, all concept equalities are translated into two subsumption assertions. Second, we translate $C \sqsubseteq C_1 \sqcap C_2$ into $C \sqsubseteq C_1$ plus $C \sqsubseteq C_2$. Applying these substitutions recursively yields a TBox having only concept assertions of the following form:

\[
C \sqsubseteq A \\
C \sqsubseteq \exists R.A
\]

Thus we have complex concepts on the left hand side, however we only have atomic concepts or $\exists R.A$ on the right hand side.

Furthermore, we replace the domain and range restrictions by the following concept subsumptions: (domain $R A$) is replaced by $R[1] \sqsubseteq A$, and (range $R A$) is replaced by $R[2] \sqsubseteq A$. $R[1]$ and $R[2]$ denote new atomic concepts that encompass every individual that is either within the domain or within the range of $R$ in the ABox.

### 3.4 Query Rewriting

The basic idea of rewriting is to generate a set of rewritten queries from a single query so that the union of the results of all queries will give the answer to
the original query w.r.t. $DL_{LH}$ semantics. The set of rewritten queries can be
answered using only the ABox, because all assertions from the TBox are already
included in the rewritten query set.

This is achieved by iteratively applying a set of rewriting rules, until no more
rules are applicable. These rules reflect the different types of assertions in the
tBox. We now look at the various types of TBox assertions.

**Concept Subsumption** First, we look at the concept subsumption. Assume
there is a subsumption assertion $C \sqsubseteq A$ in the TBox, and the query contains a
disjunction including $A$. Then we also have to look for individuals that belong to
$C$ additionally to those that belong to $A$. Thus we extend the query by replacing
$A$ with $C \lor A$ in the disjunction. However, $C$ might be a complex concept. In this
case, we have to generate a set of disjunction that assert the membership in $C$.
A new query has then to be added to the query set that contains the generated
set of disjunctions while the occurrence of $A$ is removed. The process is detailed
in the following subsection.

Furthermore, we have to look at subsumption assertions that have $\exists R.A$ as
right-hand side. In this case, we have to search for disjunctions like $R(x_1, x_2)$ in
the query in combination with $A(x_2)$. $x_2$ must not appear in any other part of
the query, and it must be an undistinguished variable. This is because the $\exists R.A$
claims only the existence of a role filler for $R$ of type $A$. It does not include that
the individual is explicitly known or otherwise specified.

Note that this construct might be applied multiple times to resolve chains.
Consider e.g. the query $q(x_1) \leftarrow \exists(x_2, x_3) \cdot R(x_1, x_2), Q(x_2, x_3), A(x_3)$. The
knowledge base contains the assertions $B \sqsubseteq \exists Q.A$ and $C \sqsubseteq \exists R.B$. In the first
step, a rewritten query $q_2(x_1) \leftarrow \exists x_2 \cdot R(x_1, x_2), B(x_2)$ is added. Analyzing the
added query, we see that also $q_3(x_1) \leftarrow C(x_1)$ must be added to the query set.

**Complex Concepts** Now consider a complex subsumee concept, e.g. $C \sqsubseteq \exists R.D \sqsubseteq A$. When we have a query that contains $A$, also instances of $C \sqsubseteq \exists R.D$
must be included in the query answer. Thus we have to rewrite the query that
it encompasses these results. This is achieved in multiple steps that are applied
exhaustively. First, the query is duplicated, and the disjunction containing $A$ is
removed from the second version. It is then replaced by the complex concept.
This is done by analyzing the structure of the complex concept. If the complex
concept is of the form $C_1 \sqcap C_2$, then $C_1$ and $C_2$ are added as two different
disjunctions. If it is of the form $\exists R.D$, then it is replaced by $R(x, y)$ and $D(y)$,
where $y$ is a new non-distinguished variable, and $x$ is the variable originally used
in the disjunction where $A$ was part of. Thus, in our example, $q(x) \leftarrow A(x)$ is
expanded by adding $q_2(x) \leftarrow \exists y : C(x), R(x, y), D(y)$.

**Roles and Transitivity** Role subsumption is straightforward, as we only allow
atomic roles. Consider a query containing an atom $R(x, y)$ and a knowledge base
containing $S \sqsubseteq R$. Then we replace $R$ with $R \lor S$. 
Now consider a transitivity assertion \( \text{transitive } R \). To evaluate queries containing transitive roles, we introduce a third type of disjunction \( (R_1, \ldots, R_n)^+(x, y) \) that evaluates to true if \((x, y)\) is bound to a pair of individuals that are contained in the transitive hull of the union \( R_1 \cup \cdots \cup R_n \).

The rewriting algorithm now has to replace every occurrence of \( R(x) \) with \( R^+(x) \), whenever \( \text{transitive } R \) is contained in the TBox. If \( R \) is contained in a disjunction like \( (R \lor S)(x, y) \), then the query must be split into two, one containing \( R^+(x, y) \), and another one containing \( S(x, y) \). Note that this is even the case if \( S \) is also transitive, because \( (R \cup S)^+ \neq R^+ \cup S^+ \) in general.

However, after \( R^+(x, y) \) has been generated, the sub-roles of \( R \) can be inserted into the same disjunction no matter whether they are transitive or not. Thus any transformation from \( R(x, y) \) to \( R^+(x, y) \) must be applied before executing the above rule concerning \( S \sqsubseteq R \).

As an example, consider the role hierarchy \( T \sqsubseteq S, S \sqsubseteq R, P \sqsubseteq Q, Q \sqsubseteq R \), where only \( S \) is transitive. Given the query \( q(x, y) \leftarrow R(x, y) \), the first transformation includes \( S \) and \( Q \) as sub-roles of \( R \) in the atom: \( q'(x, y) \leftarrow (R \lor S \lor Q)(x, y) \). Now the query has to be split up into two queries because \( S \) is transitive: \( q_1(x, y) \leftarrow (R \lor Q)(x, y) \) and \( q_2(x, y) \leftarrow S^+(x, y) \). Now we can apply the sub-role assertions again, leading to \( q_1(x, y) \leftarrow (R \lor Q \lor P)(x, y) \) and \( q_2(x, y) \leftarrow (S \lor T)^+(x, y) \).

**Domain and Range Assertions** A domain assertion like \( \text{domain } R A \) states essentially, that every individual that appears at the left side in a role filler for \( R \) must be part of \( A \). This means that queries asking for instances of \( A \) also have to consider left-side role fillers for \( R \). To support this concept, we have already introduced the artificial atomic concepts \( R[1] \) and \( R[2] \). As the normalization has furthermore generated the subsumption \( R[1] \sqsubseteq A \), no special treatment of domain and range assertions is necessary.

**Inverse Roles** Inverse roles are processed straightforward. Whenever \( \text{inverse } R S \) is part of the knowledge base, we introduce for every role disjunction that contains \( R \) an additional role \( S^- \) in the same disjunction, and vice versa. Consider e.g. the query \( q(x, y) \leftarrow (R \lor P)(x, y) \) and the assertion \( \text{inverse } R S \). Then the rewritten query is: \( q'(x, y) \leftarrow (R \lor P \lor S^-)(x, y) \).

Whenever a artificial concept \( R[1] \) is used in some concept disjunction, and \( \text{inverse } R S \) is included in the tbox, also \( S[2] \) will be added to the same concept disjunction, and vice versa.

After the introduction of inverse roles \( S^- \), every transformation concerning roles listed in the previous sections also has to be applied to these inverse roles. For example, consider \( P \sqsubseteq S \) is in the tbox, and a role disjunction \( (S^-)(x, y) \) is in the query. This disjunction is then expanded to \( (P^- \lor S^-)(x, y) \). The same holds for transitivity of roles and concept subsumption assertions where the right-hand side is like \( \exists A.R \).
Multiple Occurrences of a Role or Concept  It is important to underline that multiple occurrences of the same role or concept in a query are handled separately. For example, consider the query $q_0(x, y, z) \leftarrow A(x), R(x, y), B(y), R(x, z), C(z)$. If the knowledge base contains $S_1 \sqsubseteq R$, where $S_1$ is transitive, the expanded query set will include the following queries:

$q_0(x, y, z) \leftarrow A(x), R(x, y), B(y), R(x, z), C(z)$
$q_1(x, y, z) \leftarrow A(x), S^*(x, y), B(y), R(x, z), C(z)$
$q_2(x, y, z) \leftarrow A(x), R(x, y), B(y), S^*(x, z), C(z)$
$q_3(x, y, z) \leftarrow A(x), S^*(x, y), B(y), S^*(x, z), C(z)$

Reduction of the Query Set  As described in the previous subsection, various transformation steps generate new versions of the query that are added to the overall query set. During this process, it might happen that very similar queries are generated. Thus we regularly run a reduction algorithm that identifies similar queries that can be merged into a single query. This is important to avoid fast growing of the query set. Additionally, we check here whether a query is subsumed by another query in the set generated so far. If this is the case, the query is removed.

Here is an example. If the query set contains the following queries:

$q_i(x, y, z) \leftarrow A(x), B(y), R(x, y)$
$q_j(x, y, z) \leftarrow A(x), C(y), R(x, y)$

then these queries can be replaced by a single query

$q'_i(x, y, z) \leftarrow A(x), (B \lor C)(y), R(x, y)$

4 Query Execution

In this section, we discuss how the knowledge can be distributed over BabelPeers, and how the rewritten query set can be efficiently evaluated.

4.1 Overview

For the distribution, the general idea is as follows. The p2p network is only used to store the ABox, i.e. the concept assertions and role assertions. A role assertion $R(a, b)$ is stored as a triple $\langle a, R, b \rangle$. A concept assertion $C(a)$ is also stored as a triple: $\langle a, \texttt{rdf:type}, C \rangle$. The TBox is broadcasted to every node in the network, as we assume it is orders of magnitude smaller than the ABox. The triples are still distributed using the three indices subject, predicate, and object. This is to support also the other BabelPeers query algorithms.
For query evaluation, the following steps are performed. First, the client sends the original query to an arbitrary BabelPeers node. This node takes responsibility for the execution of the query. The query rewriting is done completely local, as every node has the complete TBox stored. However, for the evaluation of the rewritten query set, the ABox is needed which is distributed over the nodes of the p2p network. Thus the responsible node contacts various nodes during the evaluation to collect relevant parts of the ABox. In the following, we describe this process in detail. An overview is given in figure 1.

Due to the way BabelPeers distributes the triples, and due to the way we use the triples to store role and concept assertions, we end up with a situation where each role and each concept is stored entirely on an individual peer. This approach greatly helps during query execution, because we only have to contact a single peer for each involved concept or role.

4.2 Query Planning

The output of the rewriting algorithm is a set of queries that all have to be executed. The final result will be the union of all these queries. However, the queries always have large parts in common. This is due to the fact that they are generated from the same original query by replacing certain disjunctions. The main challenge in this phase is to build a query plan for the whole set of queries that exploits the overlaps between the queries in an efficient way.

The evaluation starts with a preprocessing phase. In this phase, every role and concept is checked whether it is empty. Although this implies a DHT lookup per concept, the effort for this phase can be kept small by caching the results. In
this context, “empty” means that there are no ABox assertions regarding this role or concept. As any sub-concepts are already included in the queries, we can simply drop empty concepts/roles from the queries. In case a whole disjunction runs empty we can immediately delete the corresponding query, as it cannot have any results. After the simplification, we can reduce the query set again as described in section 3.4.

With the remaining query set, a plan is build to execute these queries using a heuristic algorithm. In the first phase, each pair of queries is checked for common sub-queries. All common sub-queries are recorded. In the second phase, each query is tested against all common sub-queries, and it is counted how often each of the common sub-queries appears in the queries. If there is more than one common sub-query, the set of common sub-queries is optimized recursively using the same algorithm. This yields a DAG of common sub-queries for the original query-set.

4.3 Query Execution

After these preliminaries, the actual execution is done. Here, each query is evaluated sequentially by executing relational expressions. These fetch the existing values for concepts and roles from the p2p network for each disjunction. The values are aggregated using a union operator to reflect the disjunction. The conjunctions are executed as joins between the existing sub-results. Here, the order of the disjunctions within a query is important for performance.

In each step, the query processor first tries to use already evaluated sub-queries from a cache that stores the previous sub-results. Here, a sub-query is selected that has the largest number of matching query disjunctions with the not-yet evaluated part of the current query. If no sub-query with cached values can be found, then the sub-query that has the largest overlap with the remaining query is selected and executed. Only in case that no useful sub-query is found, we select a “fresh” disjunction and evaluate it directly to proceed with the query. This disjunction is chosen based on the size of the base relations. The relation size is fetched from the network and cached.

As the same algorithm is used to evaluate and cache the sub-queries, also sub-sub-queries are cached and reused. This results in a dramatic reduction of query evaluation delay.

The evaluation of a disjunction takes place in two different ways. When no transitivity is involved, it is possible to fetch each participating role or concept directly from the peer that stores the data. In order to make the bandwidth consumption as small as possible, filtering is performed directly on the target peer. Here we have to consider different kinds of filtering. When one of the two columns of a role is specified as a constant, the peer can directly filter all rows that don’t match this constant in the respective column. However, more efficient filtering can be done based on already known values for variables.

For this, consider a query memberOf(?x, departement0), telephone(?x, ?y). The peer that stores the memberOf role locally filters those rows that do not pertain to departement0. Thus it sends a list of every member of departement0
to the source peer that processes the query. The peer that holds the telephone role also sends the complete list of everybody’s telephone numbers to the source peer. Here, the join is processed.

This behavior can be improved using bloom filters [7]. Firstly, after transferring the data from one relation to the source peer, this peer can build a bloom filter that captures the existing values for the common variable ?x. This filter can then be sent to the other peer that uses it to reduce the sent data set and thus to lower bandwidth consumption. Here, the question is how to decide which side of the join is fetched first. Our heuristic chooses the smaller relation. This leads to the situation shown in figure 2.

An interesting improvement (which is, however, currently not implemented in BabelPeers) would be to build bloom filters directly on both sides and to exchange the filters before sending the data to the source peer. This could lower bandwidth consumption further. The approach is visualized in figure 3.

When transitivity is included, the situation is different. Here, no filtering is allowed before all participating roles are brought together. Consider e.g. the disjunction (subDepOf, subOrgOf)*(?x, ?y). The goal is to find the transitive closure of the sub-organization relationship. As the role subDepOf that captures sub departments is a child role for subOrgOf, also these tuples have to be processes. However, when there are known restrictions to either variable, it is not possible to apply these filters before computing the closure.

As in our setting the data for subDepOf and subOrgOf is distributed over two different peers, we have to first collect all data in one place. To minimize
traffic, we do not collect the data on the source peer. We rather select that peer that already holds the largest part of the data. In our example, this might be the peer that holds the subOrgOf role. This peer becomes responsible to evaluate the whole disjunction. It fetches the data from the other involved relations, then computes the transitive closure, and finally applies any filters like available bloom filters. The result of the filtering process is then sent to the source peer for further processing. Figure 4 gives an overview.

During the evaluation of atomic roles and concepts, also the special concepts $R[1]$ and $R[2]$ are evaluated by projecting $R$’s triples to their subject or object, respectively. Also inverse roles $R^\neg$ are respected by swapping the subject and the object of role’s triples. This is done directly on the peer that holds the role’s or concepts’s data, leading also to optimized bandwidth consumption.

5 Application Scenario

The enhanced BabelPeers version is used for our Peermarks scenario. For details about Peermarks, see [4]. Here we only focus on how we use BabelPeers with $DL_{LH}$ for this application. The idea of Peermarks is to do a distributed classification of tags and bookmarks for a distributed collaborative bookmarking application. However, as the classification is the user-supplied data, we store it in the Abox rather than in the Tbox. The Tbox describes the used concepts and roles and contains general axioms.

First, there is a concept Category that contains every user-defined category as an instance. For the category hierarchy, we use the hasChildCategory and
hasParentCategory roles. These roles are defined as inverse roles, thus it is sufficient to either state Cat1 hasChildCategory Cat2 or Cat2 hasParentCategory Cat1. The other axiom will be inferred. Furthermore, we use two roles named hasTransitiveChildCategory and hasTransitiveParentCategory, which are super-roles for hasChildCategory and hasParentCategory, respectively. As these roles are declared to be transitive in the Ontology, they can be used to query the transitive closure of the category hierarchy.

Furthermore, we allow arbitrary relationships between the categories, to declare a category to be related to another category. For this, a role hierarchy is used that is rooted at the role relatedCategory. This hierarchy is meant to be extended to more specifically describe the type of relationship between the categories.

To classify tags or other resources, a role classified is used with range Category. This approach is very flexible, as anything can be regarded as a resource. It has just to be described through an URI. So both bookmarks, tags, or anything else can be classified using the category hierarchy. RDF is very flexible in this respect, additional roles or concepts can be used to further describe the classified resources, making Peermarks open to further szenarios.
6 Evaluation

In this section, we present preliminary evaluations of our methods based on simulations. We are mainly interested in bandwidth consumption of the query processing as this is typically the bottleneck in p2p scenarios. We focus here on two questions:

1. What is the effect of our query execution methods compared to a simpler approach without bloom filters?
2. What is the scalability, i.e. how fast grows the bandwidth consumption when the size of the ABox grows?

In future work, we plan to focus on additional evaluations like load-balancing effects, and scalability in terms of the network size.

For the evaluation, we use the well-known Lehigh University Benchmark [11]. Our logic supports all axioms used by this benchmark, so we are able to answer each query of the benchmark correctly. The ABox for the benchmark is available in various sizes, denoted LH1, LH5, LH10 etc. The number is proportional to the size, meaning that LH10 has roughly 10 times as much ABox assertions than LH1. The benchmark consists of 14 queries of varying difficulty.

In the first scenario, we have simulated a 32 node network. We have distributed the data of LUBM1 over the network and evaluated the 14 queries. Thereby, we measured the overall bandwidth consumption, both in the upstream and downstream. We did this with two variants, one using a rather plain version of the query processing, the other one using bloom filters. The results are presented in figure 5. For every query, we show the volume of data transmitted during the answering of the queries. The values are separated in query messages (sent) and answer messages (rec.).

Overall, we can see that the additional filtering leads to a reduction in bandwidth consumption by roughly 50% in the downstream. In the meantime, we see that the upstream demand rises as more filter information has to be transmitted. However, it is still very small compared to the gain the downstream. Looking at individual queries, we can see that especially the transfer query 9, the most demanding query of the LUBM benchmark, is reduced significantly.

The second scenario measures the growth of the bandwidth consumption when the size of the ABox increases. We have again used a 32 node network, and compared the bandwidth demand during the execution of the queries over LUBM1, LUBM5, and LUBM10. The results shown in figure 6.

The main result of this measurement is to see that the consumption grows roughly linearly in the size of the ABox. Looking again at individual queries, we see that especially for query 9, the step from LH5 to LH10 does not lead to a doubled transfer volume.

7 Related Work

Storing and querying machine-processable data is at the heart of the Semantic Web. However, the full potential is only unleashed with sophisticated reasoning
algorithms, that allow to derive implicit knowledge from the explicitly encoded data. However, the problem is the anticipated amount of data that renders computationally demanding reasoning algorithms unusable.

Thus, different streams of research can be seen in the Semantic Web community. On the one hand, the traditional description logics research focuses on tableau based reasoning that yields powerful reasoning features. However, tableau reasoners normally suffer from performance problems. Along many others, pellet [17] is an example for a sophisticated tableau-based DL reasoner including state-of-the-art optimizations. Although pellet is quite fast, its query answering speed cannot compete with relational databases.

On the other side of the spectrum, Sesame [8] is a fast and efficient RDF triple store that allows for RDF Schema semantics. It does reasoning by forward-chaining, effectively generating every possible conclusion. However this approach is limited in terms of the reasoning semantics.

The DL community has realized the need for semantically rich but yet fast reasoning mechanisms. Two prominent examples of this quest are EL [2] and DL-Lite [10]. EL is an simple DL dialect that has a polynomial reasoning complexity. DL-Lite is also a simplified DL that is executed using query rewriring. DL-Lite was the basic inspiration for our work, however we focus on a different set of basic constructs in the DL.
KAON2 [13] is a reasoner and Semantic Web framework that supports an expressive DL dialect. The DL reasoning is also done without a tableau algorithm. Instead, a translation to disjunctive datalog programs is employed.

The Jena framework [18] from HP is quite general, including various different reasoners. These reasoners encompass different trade-offs between semantics and performance.

In order to improve scalability several other projects use distributed hash table P2P networks (DHTs). Among these are RDFPeers [9], Atlas [14], RDF-FCube [15] and GridVine [1]. While the projects share common objectives they differ in their query processing and load-balancing strategies and capabilities. Edutella [16] follows a different route because it uses super-peer P2P networks as an underlying architecture.

8 Conclusion

In this paper, we have focussed on the question how semantically rich reasoning based on query rewriting can be used to realize a scalable semantic data store based on distributed hash table technology. Thus solution is both scalable and supports many useful axiom types in the underlying Ontology.
Specifically, the structure of the query set generated from the query rewriting method is exploited to build efficient plans for executing these queries in the p2p network. We have introduced several techniques to reduce the bandwidth demands of query processing, which is the main performance factor in our scenario.

We also included hints how to further optimize the query processing. Our methods already reduce the bandwidth demand by 50 percent. We believe that further improvements are possible, leading to even better performance and scalability.

Overall, we believe that the presented approach is an important step in realizing the vision of the Semantic Web, that will only come to its full potential when reasoning algorithms are able to cope with large and distributed amounts of data in an efficient way without restricting semantics too much.

References

   URL citeseer.ist.psu.edu/aberer04gridvine.html


