RDFS Reasoning and Query Answering on Top of DHTs

Zoi Kaoudi*, Iris Miliaraki**, and Manolis Koubarakis

Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens, Greece

Abstract. We study the problem of distributed RDFS reasoning and query answering on top of distributed hash tables. Scalable, distributed RDFS reasoning is an essential functionality for providing the scalability and performance that large-scale Semantic Web applications require. Our goal in this paper is to compare and evaluate two well-known approaches to RDFS reasoning, namely backward and forward chaining, on top of distributed hash tables. We show how to implement both algorithms on top of the distributed hash table Bamboo and prove their correctness. We also study the time-space trade-off exhibited by the algorithms analytically, and experimentally by evaluating our algorithms on PlanetLab.

1 Introduction

As the Semantic Web has become a reality, there is an emerging need not only for dealing with a huge amount of distributed metadata, but also for being able to reason with it. Previous work on *centralized* RDF stores has considered forward chaining, backward chaining and hybrid approaches to implement RDFS reasoning and query processing [6, 27, 2, 12]. In the forward chaining approach, new statements are exhaustively generated from the asserted ones. In contrast, a backward chaining approach only evaluates RDFS entailments on demand, i.e., at query processing time. Intuitively, we expect that a forward chaining approach has minimal requirements during query answering, but needs a significant amount of storage. In contrast, the backward chaining approach has minimal storage requirements, at the cost of an increase in query response time. There is a time-space trade-off between these two approaches [24], and only by knowing the query and update workload of an application, we can determine which approach would suit it better. This trade-off has never been studied in detail in a *distributed* Internet-scale scenario, and this is one of the challenges that we undertake in this paper.

P2P networks and especially distributed hash tables (DHTs) [3] have gained much attention recently, given the scalability, fault-tolerance and robustness features they can provide to Internet-scale applications. Since current centralized

* Supported by project "Peer-to-Peer Techniques for Semantic Web Services" (funded from the Greek General Secretariat for Research and Technology).

** Supported by Microsoft Research through its European PhD Scholarship Programme.

RDF repositories lack the required scalability and fault tolerance for such applications [10], DHTs have been proposed for the storage and querying of RDF data at Internet scale [7, 19, 14, 1]. However, these works are solely concerned with query processing for RDF data, and pay no attention to RDF Schema (RDFS) reasoning and query processing. The only DHT-based RDF store that has dealt with RDFS reasoning in the past is BabelPeers [5, 14]. It is implemented on top of Pastry [23] and supports a subclass of the SPARQL query language [21]. BabelPeers uses a forward chaining approach in order to provide the RDFS inference capability required to answer the supported class of SPARQL queries.

In this paper we design and implement both forward and backward chaining algorithms for RDFS reasoning and query answering on top of the Bamboo DHT [22]. Our algorithms have been integrated in the RDF query processing system Atlas (`http://atlas.di.uoa.gr`) [15] and have been used to enable the processing of RQL [17] schema queries. To the best of our knowledge, our backward chaining algorithm is the first distributed backward chaining algorithm proposed on top of DHT-based RDF stores. Another contribution of this work is proving the correctness of our algorithms and providing a comparative study of forward and backward chaining algorithms both analytically and experimentally. In addition, we propose an optimization technique for backward chaining which decreases query response time and allow us to minimize the response time difference between the two approaches. For the experimental part of our study, we deploy our system on PlanetLab to obtain measurements in a realistic large-scale distributed environment. The results obtained in our experiments agree with the predictions of our analytical model. An important result of our experiments is that forward chaining is a very expensive algorithm in terms of storage load, bandwidth and time and does not scale for a large number of triples.

2 Data and Query Model

In the rest of the paper, we will constantly use the notion of an RDF triple. RDF data as well as RDFS descriptions (we will further use the term RDF(S) to refer to both) can be written as triples and constitute an *RDF(S) database*.

To support RDFS reasoning, the RDFS entailment rules of RDF Semantics [13] constitute a vital element of our data model. Following a datalog-like notation with extensional database relation (edb) *triple* and intensional database relations (idb) *subClass*, *subProperty* and *type*, the RDFS entailment rules can be written as shown in Table 1. Each rule is indexed by a number that we will use to refer to it. Rules are also indexed with their symbolic name from [13] for co-reference reasons. Certain rules are deliberately omitted (such as the ones with the axiomatic triples) since we are more interested in rules needed for the computation of the transitive closure. However, our algorithms work with all the RDFS entailment rules except the ones with blank nodes. We leave it as future work to consider various implications that these rules might have.

In our notation, arguments beginning with a capital letter (such as X and Y) denote variables, and arguments starting with a lowercase letter denote constants. Predicate names always start with a lowercase letter. Namespaces *rdf*

Table 1. RDFS Entailment Rules

Rule	Head	Body
1	$type(X, Y)$	$triple(X, rdf{:}type, Y)$
2 (rdfs2)	$type(X, Y)$	$triple(X, P, Z)$, $triple(P, rdfs{:}domain, Y)$
3 (rdfs3)	$type(X, Y)$	$triple(Z, P, X)$, $triple(P, rdfs{:}range, Y)$
4 (rdfs9)	$type(X, Y)$	$type(X, Z)$, $subClass(Z, Y)$
5	$subProperty(X, Y)$	$triple(X, rdfs{:}subPropertyOf, Y)$
6 (rdfs5)	$subProperty(X, Y)$	$triple(X, rdfs{:}subPropertyOf, Z)$, $subProperty(Z, Y)$
7	$subClass(X, Y)$	$triple(X, rdfs{:}subClassOf, Y)$
8 (rdfs11)	$subClass(X, Y)$	$triple(X, rdfs{:}subClassOf, Z)$, $subClass(Z, Y)$

and $rdfs$ are the namespaces of the core RDF and RDFS vocabulary and will be used throughout the paper. To avoid confusion with the double meaning of the word *predicate*, from now on we will refer to the predicate of a triple with the word *property* and to a term of a datalog rule with the word *predicate*.

Rules 1-4 compute all possible instances of a class, rules 5 and 6 compute the transitive closure of an RDFS property hierarchy, and rules 7 and 8 compute the transitive closure of an RDFS class hierarchy. We note that all recursive rules above are *linear* (with at most one recursive predicate in their body) and *safe* (all variables appear as an argument in the rule bodies).

In the rest of the paper, we consider queries consisting of a *single* edb or idb predicate with arguments that are constants or variables. A formula of the form (s, p, o) where s, p and o can be URIs, literals or variables is called a *triple pattern*. We will use the equivalence of triple patterns (s, p, o) with p equal to $rdf{:}type$, $rdfs{:}subClassOf$, or $rdfs{:}subPropertyOf$, and idb predicates $type$, $subClass$ and $subProperty$, to navigate freely among these two representations.

Let DB be an RDF(S) database. The *answer* to a query is defined as the answer to the same query posed over the logic program formed by the union of the triples in DB and the RDFS entailment rules of Table 1. We omit detailed formal definitions since they are very well understood [20, 9].

3 Distributed RDFS Reasoning

Firstly, let us briefly describe the functionality of DHTs. DHTs are *structured* P2P systems which try to solve the *lookup problem*: given a data item x, find the node which holds x. Each node and each data item is assigned a unique m-bit identifier by using a hashing function such as SHA-1. The identifier of a node can be computed by hashing its IP address. For data items, we first have to compute a *key* and then hash this key to obtain an identifier id. The lookup problem is then solved by providing a simple interface of two requests; PUT(id, x) and GET(id). In Bamboo [22], when a node receives a PUT request, it efficiently routes the request to a node with an identifier that is numerically closest to id using a technique called prefix routing. This node is responsible for storing the data item x. In the same way, when a node receives a GET request, it routes it to the responsible node to fetch data item x. Such requests can be done in $O(logn)$ hops, where n is the number of nodes in the network.

Although for the implementation of our algorithms we used Bamboo [22], our algorithms are DHT-agnostic; they can be implemented on top of any DHT.

3.1 Storing protocol

In both approaches, the same protocol is followed for storing RDF(S) triples. We have adopted the triple indexing algorithm originally presented in [7] where each triple is indexed in the DHT *three times*, once for its subject, once for its property and once for its object. Whenever a node receives a request to store a triple, it sends three DHT PUT requests using as key the subject, property and object respectively, and the triple itself as the item. The key is hashed to create the identifier that leads to the responsible node where the triple is stored. We call that node the *responsible node* for this key or identifier. Notice that a node responsible for a key which is a class name C (*responsible node for class* C), will have in its local database all triples that contain class C either as a subject or as an object (class C cannot be a property).

Since an RDF(S) database is actually a graph, we can exploit the fact that many of the triples share a common key (i.e., they have the same subject, property or object) and end up to be stored in the same node. So, instead of sending different PUT messages for each triple, we group them in a list *triples* based on the distinguished keys that exist, hash these keys to obtain identifiers and send a MULTIPUT$(id, triples)$ message for each identifier. The node responsible for the identifier *id*, which receives this message, stores in its local database all triples included in the list *triples*.

3.2 Forward chaining

The general idea of forward chaining (FC) is that all inferred triples are precomputed and stored in the network a priori. Each time a node receives a triple to be stored in its local database, it computes all inferred triples and sends them to the network to be stored.

Let us now introduce the notation that will be used in the algorithms description. Keyword **event** precedes every event handler for handling messages, while keyword **procedure** declares a procedure. In both cases, the name is prefixed by the node identifier in which the handler or the procedure is executed. Keywords **sendto** and **receive** declare the message that we want to send to a node with known either its identifier (thus DHT routing will be used) or the IP address, and the message we receive from a node respectively.

Figure 1 shows in pseudocode how the FC algorithm works. Suppose a PUT$(id, triple)$ request arrives at node n and a new triple should be stored in the local database of n. First, node n retrieves from the local database all triples that have been stored under the identifier *id* and puts them in list *triples*. Then, it computes the inferred triples from this list according to the RDFS entailment rules using local function INFER(*triples*). For all newly inferred triples, three identifiers are created based on the subject, property and object of each triple, and three PUT requests are sent to the network. Each node holds a list $infTriples$ with all inferred triples that it has computed, so that it can check when it reaches a fixpoint where no new triples are generated. The algorithm terminates when all nodes have reached a fixpoint.

The following proposition states that the FC algorithm is sound and complete. By *sound* we mean that if t is a new triple produced by FC and stored in the network, then t is entailed by the set of logical formulas formed by the RDF(S) database before the algorithm is executed, the input triple that fires the algorithm and the RDFS entailment rules of Table 1. By *complete* we mean that if a triple t is entailed by the set of logical formulas formed by the triples stored in the network before FC execution, the input triple that fires the algorithm and the RDFS entailment rules of Table 1, then t will eventually be generated by FC and will be stored in the network.

Algorithm 1: FwdRDFS

```
1 event n.PUT(id, triple)
2     triples=GETTRIPLESFROMDB (id);
3     newtriples=INFER (triples);
4     forall t of newtriples not in infTriples do
5         id1 =Hash (t.subject);
6         id2 =Hash (t.property);
7         id3 =Hash (t.object);
8         sendto id1.PUT(id1, t);
9         sendto id2.PUT(id2, t);
10        sendto id3.PUT(id3, t);
11        add t to infTriples;
12    end
13 end event
```

Fig. 1. FC algorithm

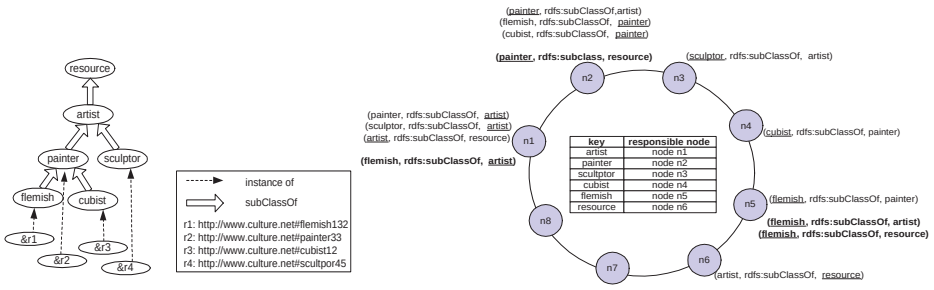
Proposition 1. *The FC algorithm is sound and complete.*

Proof (Sketch). The algorithm is sound since it is based on the RDFS entailment rules of Table 1. To prove that the algorithm is complete, we need to show that triples which are used to satisfy the body of a rule and generate a new triple will meet at the same node. If we check the rules of Table 1, we will see that predicates of rule bodies always have a common argument. Triples are indexed three times based on three identifiers, namely the hash values of their subject, property and object. Therefore, triples with a common part will meet at the node responsible for the identifier of this common part. □

Note that the proof of the proposition depends on the assumption that no messages are lost due to network churn, i.e., nodes joining, leaving voluntarily or failing. Although the Bamboo DHT has many recovery mechanisms and can handle churn using various methods [22], we leave it as a subject of future work how our algorithms can be extended to deal with dynamic networks.

In order to evaluate a triple pattern after FC has taken place, we choose a *key* from the triple pattern and then hash it to create the identifier that will lead to the appropriate node. The key is the constant part of the triple pattern. When there is more than one constant parts, we select keys in the order "subject, object, property" based on the fact that we prefer keys with lower selectivity and the reasonable assumption that subjects or objects have more distinct values than properties. At the destination node, the triple pattern is checked against the local database and matching triples are found.

Since the RDFS entailment rules are highly-redundant [18], even a centralized forward chaining approach can be very expensive. As we will also show in the experimental evaluation of FC, the distributed version of FC results in generating more redundancies than expected in a centralized environment and leads to a significant increase in network traffic and load. Let us demonstrate that with an example. Figure 2(a) depicts a small RDFS class hierarchy of the cultural domain [17] with some sample instances populated underneath. Figure 2(b) shows the indexed triples that concern the subclass relation. Triples that are not in bold are initially inserted in the network. The key of each triple that led to a specific node is underlined. After two iterations of FC, both nodes n_1 and n_2 will result



(a) Example RDF(S) (b) Indexed triples before and after (*in bold*) FC

Fig. 2. Example for forward chaining

in generating the triple $(flemish, rdfs{:}subClassOf, resource)$. This triple will be sent twice in the network to be stored. This could have been avoided in a centralized environment where all triples are stored in one local database.

3.3 Backward chaining

In contrast to the data driven nature of FC, backward chaining (BC) starts from the given query and tries to find rules that can be used to derive answers. Thus, each time a node receives a request for evaluating a query, it should also use the RDFS entailment rules to compute the complete answer.

The challenge here is to construct an algorithm that can process recursive rules in a distributed environment such as DHTs. To achieve that, considering the RDFS entailment rules, it is helpful to transform the rules presented in Section 2 to a set of *adorned rules* that indicate which variables are bound and which are free. This is useful for finding the optimal order in which predicates should be evaluated.

We will extend the concept of rule adornment from recursive query processing [26] in order to exploit the distributed philosophy of DHTs. As already mentioned, in order to evaluate a triple pattern, a *key* has to be computed and then hashed to create the identifier that will lead to the responsible node. Therefore, the corresponding predicate of the triple pattern has an argument that not only is bound, but it is also the *key* that led to the responsible node.

Definition 1. *An* adornment *of a predicate p with n arguments is an ordered string a of k's, b's and f's of length n, where k indicates an argument which is the key, b indicates a bound argument which is not the key, and f a free argument.*

Following this definition, a predicate p^a indicates which argument of p is the key, which ones are bound and which are free. Table 2 shows all possible adornments of the rules presented in Table 1.

Let us now describe the BC algorithm which is shown in Fig. 3. Suppose that a GetReq request with unique identifier *rid* arrives at node n and a query should be evaluated. Node n firstly checks if the predicate corresponding to the triple pattern *tp* matches the head of any of the adorned rules. If no rule can be

Table 2. Adorned RDFS Entailment Rules

	Head	**Body**
1a	$type^{kf}(X, Y)$	$triple^{kbf}(X, rdf{:}type, Y)$
1b	$type^{fk}(X, Y)$	$triple^{fbk}(X, rdf{:}type, Y)$
2a	$type^{kf}(X, Y)$	$triple^{kff}(X, P, Z)$, $triple^{fbf}(P, rdfs{:}domain, Y)$
2b	$type^{fk}(X, Y)$	$triple^{fff}(X, P, Z)$, $triple^{fbk}(P, rdfs{:}domain, Y)$
3a	$type^{kf}(X, Y)$	$triple^{ffk}(Z, P, X)$, $triple^{fbf}(P, rdfs{:}range, Y)$
3b	$type^{fk}(X, Y)$	$triple^{fff}(Z, P, X)$, $triple^{fbk}(P, rdfs{:}range, Y)$
4a	$type^{kf}(X, Y)$	$triple^{kbf}(X, rdf{:}type, Z)$, $subClass^{ff}(Z, Y)$
4b	$type^{fk}(X, Y)$	$type^{ff}(X, Z)$, $triple^{fbk}(Z, rdfs{:}subClassOf, Y)$
5a	$subProperty^{kf}(X, Y)$	$triple^{kbf}(X, rdfs{:}subPropertyOf, Y)$
5b	$subProperty^{fk}(X, Y)$	$triple^{fbk}(X, rdfs{:}subPropertyOf, Y)$
6a	$subProperty^{kf}(X, Y)$	$triple^{kbf}(X, rdfs{:}subPropertyOf, Z)$, $subProperty^{ff}(Z, Y)$
6b	$subProperty^{fk}(X, Y)$	$subProperty^{ff}(X, Z)$, $triple^{fbk}(Z, rdfs{:}subPropertyOf, Y)$
7a	$subClass^{kf}(X, Y)$	$triple^{kbf}(X, rdfs{:}subClassOf, Y)$
7b	$subClass^{fk}(X, Y)$	$triple^{fbk}(X, rdfs{:}subClassOf, Y)$
8a	$subClass^{kf}(X, Y)$	$triple^{kbf}(X, rdfs{:}subClassOf, Z)$, $subClass^{ff}(Z, Y)$
8b	$subClass^{fk}(X, Y)$	$subClass^{ff}(X, Z)$, $triple^{fbk}(Z, rdfs{:}subClassOf, Y)$

found, the triple pattern is simply checked against the node's local database and the bindings of the triple pattern's variables are returned to the node that made the request. In this case no backward chaining takes place. If there are rules that can be applied, local procedure `BwdRDFS` is called, which takes as an input the adorned predicate p^a and the request identifier rid and outputs a relation R which contains the bindings of the free arguments (i.e., the variables) of the predicate. These bindings form the answer to the query.

When `BwdRDFS` is called, the input predicate p^a is checked against the rules. Rules that can be applied to the predicate are added to the list *adornedRules*. Each rule can have one or two predicates in its body. Rules that have one predicate in their body (e.g., rule 1a) can always be evaluated locally since this predicate is an edb relation. In this case, node n calls local procedure MatchPredicate(p^a) and assigns to relation R the bindings of the predicate's variables that match the triples locally stored in its database.

For rules with two predicates in their body, we have to decide which predicate should be evaluated first. We select to evaluate first the predicate that can be processed locally. There always will be one such predicate since one of the arguments of the head predicate will be the key that led to the specific node. Therefore, there will be a body predicate (p_k) which has an adornment containing the letter k (always possible as seen in Table 2) and can be processed locally. Then, p_k is checked against the local database to find matching triples and the variable bindings are returned in a relation R'. By evaluating the first predicate locally, we have values that can be passed to the second predicate which is sent to be evaluated remotely at a different node. Notice in Table 2 that all rule bodies with two predicates have a single common variable (Z). Therefore, each tuple in relation R' will include a binding for this common variable Z. For each of these bindings (Z/v_i), node n rewrites the second predicate p_f to a new predicate p'_f where it has substituted the variable Z with its value v_i and made the corresponding letter of the adornment equal to k. Then, it sends a `BwdRDFSReq` message to the node responsible for the hashed value of key v_i. This

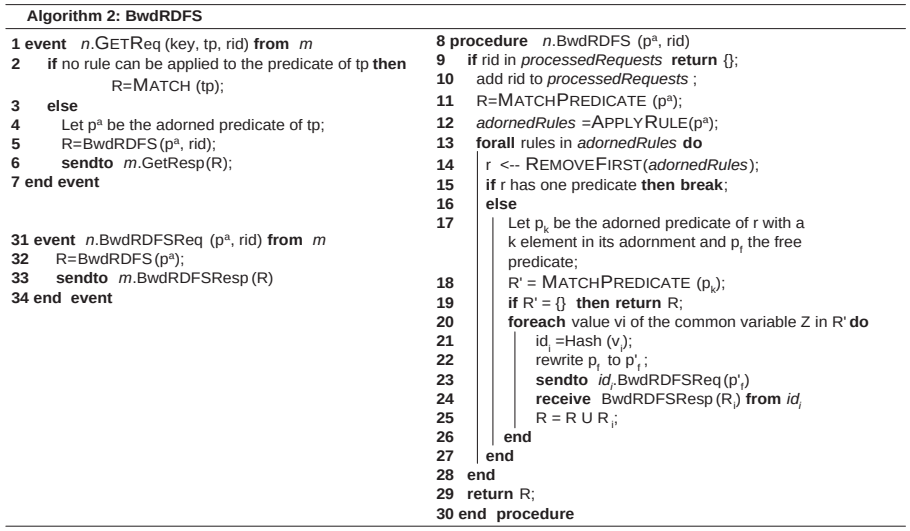


Fig. 3. BC algorithm

part of the procedure is executed *in parallel* for each value v_i since the messages are sent to different nodes. Node n sends $|R'|$ number of messages (equal to the number of bindings found) and receives the responses asynchronously. When node n has collected all responses `BwdRDFSResp`(R_i), it adds the tuples of each R_i to relation R and returns R.

This procedure is recursive and terminates when the node that received the initial query has collected all responses. A recursion path ends when the predicate which is evaluated locally returns no bindings and therefore there are no values to pass to the second predicate. Cyclic hierarchies are handled by keeping a list of all processed requests (lines 9-10) so that an infinite loop is avoided.

The following proposition states that the BC algorithm is sound and complete. In this case, by *sound* we mean that if R is the relation with the bindings of a query q produced by BC and triple t is obtained from replacing the variables of the query q by their value in R, then t is entailed by the set of logical formulas formed by the RDF(S) database and the RDFS entailment rules. By *complete* we mean that if t is entailed by the set of logical formulas formed by the triples stored in the network and the RDFS entailment rules, then a relation R will be eventually produced, such that by replacing the variables of q by their values in R, triple t will be obtained. Similarly with FC, we make the same assumption regarding the stability of the network.

Proposition 2. *The BC algorithm is sound and complete.*

Proof (Sketch). The algorithm is sound since the answers are computed using the edb relations and the RDFS entailment rules. To prove that the algorithm is complete, it is important to stress the following. The local database of each node is part of the edb relation *triple*. The adorned predicate of a rule body that has the letter k is always the edb predicate *triple*. Therefore, these predicates will be evaluated locally according to the algorithm. Now, it is sufficient to show that each time an adorned predicate *triple* is checked against a node's local database

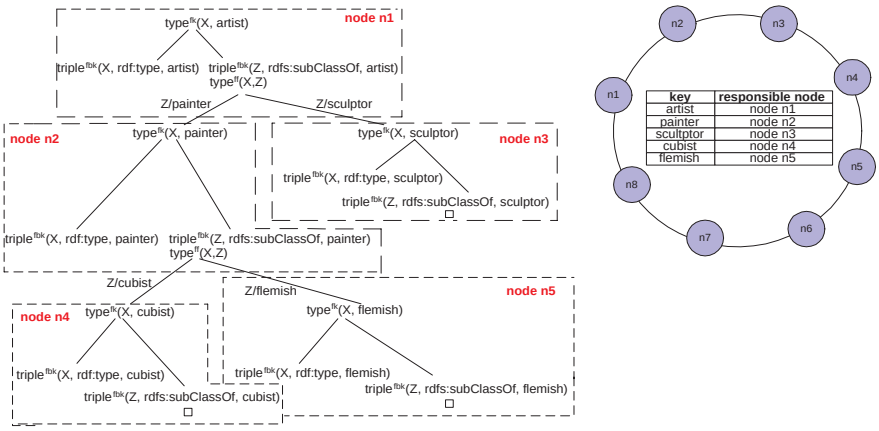


Fig. 4. Distribution of proof tree for backward chaining

all triples needed are found at this node. The adornment letter k indicates an argument of the predicate *triple* which is the key that was hashed and led to the specific node. This node is the responsible node for this key. Based on our indexing scheme all triples that contain this key either as a subject, property or object were sent to be stored to this node. As a result the edb relation of this node will include all triples that contain the specific key. □

Figure 4 shows how the proof tree of backward chaining is distributed in the various nodes of the network using the example RDF(S) hierarchy of Fig. 2(a). We want to pose the query "Find all the instances of class artist", which is expressed as $(X, rdf{:}type, artist)$. Node n_1 responsible for key *artist* receives a request for evaluating this triple pattern. Now BC should take place by starting from the adorned predicate $type^{fk}(X, artist)$.

4 An Analytical Cost Model

In this section, we present an analytical cost model for both FC and BC. We will show in the experimental evaluation that our implementation follow the predictions of this cost model. We focus on a frequently used type of queries which asks for all the instances of a certain class in an RDFS hierarchy. Intuitively, this type of queries is the most expensive and most frequently used in RDFS reasoning, so we regard it as the most representative one for comparing our algorithms. However, the algorithms are able to answer any type of queries considered in the paper. Our results for class hierarchies trivially hold for property hierarchies too. We consider as future work the evaluation of more complex queries such as the ones of the LUBM benchmark [11].

We assume a complete tree-shaped RDFS class hierarchy of depth d and branching factor b as our RDF Schema (i.e., $(b^{d+1} - 1)/(b - 1)$ classes) with instances distributed under classes following either a uniform or a Zipfian distribution. RDF Schema triples are of the form $(C_k, rdfs{:}subClassOf, C_l)$ while RDF data triples are of the form $(r_j, rdf{:}type, C_k)$. In the analytical calculations presented below we start with an RDF(S) database. Then, we apply the FC and

BC algorithms to answer a query of the type mentioned above, and estimate its cost. We will denote the number of RDF data triples (i.e., also total number of instances) in our initial database by T_d. Similarly, we use T_s to denote the number of RDF Schema triples in the initial database. For a uniform distribution, given the total number of instances (i.e., T_d), the number of instances under *each* class is $I_u = (T_d * (b-1))/(b^{d+1}-1)$. Considering a Zipfian distribution of instances with a skew parameter of 1, a class with rank r has $I_r = T_d/(r * h)$ instances where $h = \sum_{j=1}^{N} 1/j$ for N classes ($N = (b^{d+1}-1)/(b-1)$). Leaf classes are given a lower rank.

In the following, we constantly use the fact that the total number of subclasses of class C including itself is $(b^{d-\ell+1}-1)/(b-1)$ where ℓ is the level of the class. The proof is omitted due to space limitations. Furthermore, we utilize the fact that the reasoning and query answering algorithms for the type of queries we consider are essentially transitive closure computations.

4.1 Storage cost model

We first estimate analytically the costs associated with the storage of triples by both algorithms.

Storage load. We define as *storage load* the total number of triples that are stored in the network. In BC, the storage load (SL_b) is three times the number of RDF(S) triples that were inserted in the network based on our indexing scheme. In FC, it is sufficient to compute the total number of triples that result from the transitive closure computations of the hierarchy (triples initially in the database plus inferred ones). Then, the storage load incurred in FC (SL_f) is three times this total number of triples.

Lemma 1. *The total number of triples generated after the computation of the transitive closure of the RDFS class hierarchy is* $T_s + \sum_{i=1}^{d} b^i(i-1) + [(b^{d+1})/(b-1)] - 2$.

Proof. For each level i of the tree, we have b^i classes, and for each class we infer $i-1$ triples of the form $(C_k, rdfs{:}subClassOf, C_l)$ for the upper levels of the tree. Furthermore, we have the inferred triples (which are $[(b^{d+1}-1)/(b-1)]-1$) that state that all classes are subclasses of the $rdfs{:}Resource$ class. □

Lemma 2. *The total number of triples generated after the computation of the transitive instances of the RDFS class hierarchy is* $T_d + I_u * \sum_{i=0}^{d} b^i(i+1)$.

Proof. Each class has I_u direct instances. For each level i of the tree, we have b^i classes, and for each class we infer i triples of the form $(r_j, rdf{:}type, C_k)$ for its superclasses plus the triple $(r_j, rdf{:}type, rdfs{:}Resource)$. □

Based on these two lemmas, the sum of the formulas computed above is the total number of triples that result from the transitive closure computations of FC. Depending on the distribution of instances, the storage load of FC changes based on the number of instances per class (i.e., I_u and I_r). Table 3 summarizes the storage load of FC and BC for both kinds of instance distributions. For the Zipfian distribution we also made use of the following proposition. The proof is omitted due to space limitations.

Proposition 3. *Given a class with rank* r *in a Zipfian distribution of instances, the level of the class in the hierarchy is* $\ell_r = \lfloor \log_b((b-1) * [(b^{d+1}-1)/(b-1) - r + 1]) \rfloor$.

Store messages. We define as *store messages* the number of DHT messages sent for storing triples. In BC, the number of store messages sent (SM_b) is three times the number of triples stored and therefore it is equal to the storage load incurred. It is also independent from the instance distribution.

However, the total number of messages sent by FC is much larger than the storage load incurred as it was already illustrated in Section 3.2. Each node responsible for a certain class C that is in the i level of the class hierarchy generates for each instance i triples of the form $(r_j, rdf{:}type, C_k)$, where C_k is a superclass of class C, and one triple of the form $(r_j, rdf{:}type, Resource)$, and $i-1$ triples of the form $(C, rdfs{:}subClassOf, C_l)$, where C_l is an ancestor class of class C. Note that messages are sent not only for each direct instance of each class but also for the instances of its subclasses. The number of messages sent in FC is depicted in Table 3 for both kinds of instance distribution.

Table 3. Storage cost summary table

Storage cost	Uniform	Zipfian
SL_b	$3*(T_s+T_d)$	$3*(T_s+T_d)$
SL_f	$3*[T_s+T_d+\sum_{i=1}^{d} b^i(i-1)+[(b^{d+1})/(b-1)]-2+I_u*\sum_{i=0}^{d} b^i(i+1)]$	$3*[T_s+T_d+\sum_{i=1}^{d} b^i(i-1)+\sum_{r=1}^{N} I_r*\ell_r]$
SM_b	$3*(T_s+T_d)$	$3*(T_s+T_d)$
SM_f	$3*[T_s+T_d+\sum_{i=1}^{d} b^i(i-1)*((b^{d-i+1}-2)/(b-1))+I_u*\sum_{i=1}^{d} b^i*(i+1)*((b^{d-i+1}-1)/(b-1))]$	$3*[T_s+T_d+\sum_{r=1}^{N} I_r*\ell_r*(\ell_r+1)/2]$

4.2 Querying cost model

In this section, we calculate the cost of answering the query $(X, rdf{:}type, C)$ where class C is at level ℓ of the hierarchy.

Query processing load. We define as *query processing load* of a node the number of triples that this node retrieves from its local database in order to answer a query.

The FC algorithm generates query processing load (QL_f) only at the node that is responsible for class C. This load is equal to the total number of instances of class C: $QL_f = ((b^{d-\ell+1}-1)*I_u)/(b-1)$. On the other hand, BC will generate load at $(b^{d-\ell+1}-1)/(b-1)$ nodes, namely the nodes that are responsible for the subclasses of class C including C. The load of each of these nodes (QL_b) is simply I_u. Note that the total query load is the same for both approaches.

For a Zipfian distribution, the number of instances of the class would be $I_r = T_d/(h*r)$, where the rank of the class is in the interval $[(b^d-1)/(b-1)+1, (b^{d+1}-1)/(b-1)]$.

Query messages. We define as *query messages* the messages sent while answering a query. The case of FC is straightforward since just one message is sent to the node responsible for class C. In BC, the number of messages sent (QM_b) is as many as the number of the subclasses of class C. Therefore, we have: $QM_b = [(b^{d-\ell+1}-1)/(b-1)]-1$. The distribution of the instances does not affect the number of messages sent for the query answering.

5 Experimental Evaluation

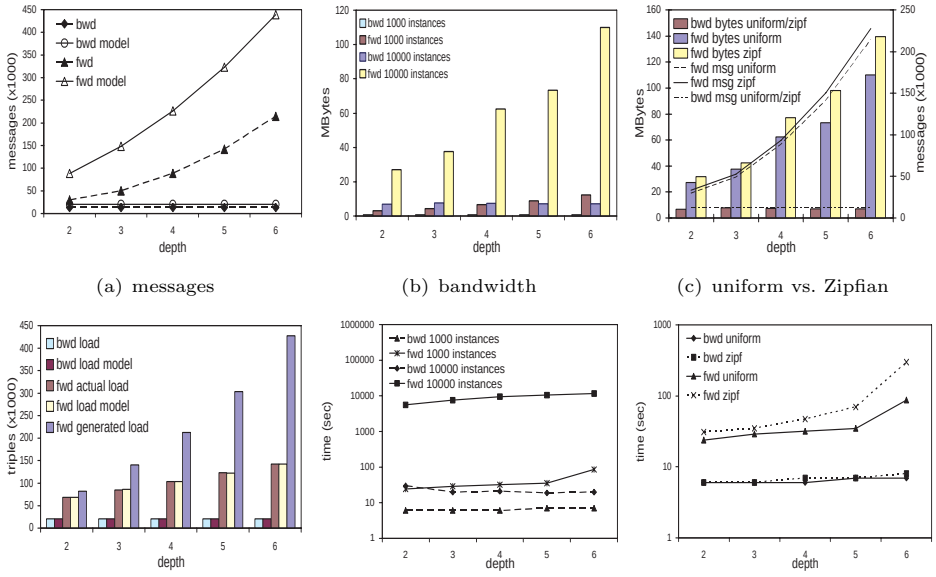
In this section, we present an experimental evaluation of both FC and BC as we have implemented them on top of the Bamboo DHT [22]. Our goal is to evaluate the performance of FC and BC in a real distributed system and compare it with the analytical cost model of the previous section. We used as a testbed the PlanetLab network (`http://www.planet-lab.org/`) with 123 nodes that were available and lightly loaded at the time of the experiments.

The RDF(S) data we used was produced synthetically from the RBench generator[25]. The generator produces binary-tree-shaped RDFS class hierarchies parameterized on three different aspects: the depth of the tree, the total number of instances under the tree, and the distribution of the instances under the nodes of the tree. The queries we measure are queries that ask for all the transitive instances of the root class of the RDFS hierarchy. We used both uniform and Zipfian distribution of instances under the RDFS class hierarchy. In the Zipfian distribution, we used a skew parameter of value 1. Leaf classes were given a lower rank and therefore more instances of the lower level classes were generated.

Storing RDF(S) data. In this section, we measure the performance of both algorithms while storing RDF(S) data. Unless otherwise specified, in the experiments below we have generated and inserted in the network 10^4 instances uniformly distributed under an RDFS class hierarchy of varying depth.

Initially, we present results concerning the network traffic that is generated by the system measured in terms of both total number of messages sent in the network and bandwidth usage. As shown in Fig. 5(a), the number of messages sent by FC increases exponentially with the tree-depth while it remains constant in BC. In this experiment, we used the MULTIPUT functionality described in Section 3.1 for both approaches and thus the number of messages measured in the experiment is less than the number of messages computed by the analytical cost model. By using this functionality, we decrease significantly the number of messages sent by FC. Figure 5(b) shows how bandwidth is increasing exponentially with the tree-depth in FC while remaining constant in BC. When we increase the number of initially inserted triples from 10^3 to 10^4, we observe a huge increase in FC's bandwidth consumption which is also affected by the tree depth. Finally, in Fig. 5(c) the difference between a uniform and a Zipfian distribution of instances is depicted for both bandwidth (left y-axis) and messages (right y-axis). BC is not affected by the distribution and as a consequence the number of messages sent as well as the bandwidth consumption remain unchanged. However, FC's bandwidth usage deteriorates when a Zipfian distribution is used. The more skewed the instances are to the lower level classes, the more triples are needed to be sent to the upper level classes and thus the more bandwidth is spent.

Figure 5(d) shows the total storage load incurred in the network. BC's storage load is significantly lower than FC and is independent of the tree-depth. However, FC's storage load is increasing linearly with the tree-depth. In the same graph, we have included the total number of triples that are sent to be stored at various nodes during FC including the redundant ones generated as we have discussed in



(a) messages (b) bandwidth (c) uniform vs. Zipfian

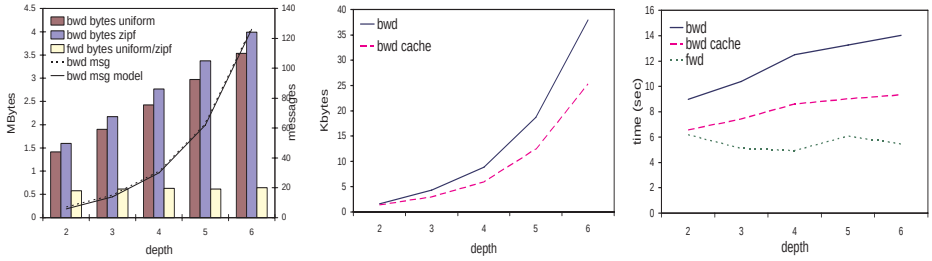
(d) storage load (e) storage time (f) uniform vs. Zipfian

Fig. 5. Storing RDF(S) data

Section 3.2. The difference between these two measurements, named *fwd actual load* and *fwd generated load*, shows that the redundant information generated by FC is remarkable and increases significantly with the tree-depth. Figure 5(d) also shows that our cost model (*bwd load model* and *fwd load model* in the graph) precisely predicts the results obtained in the experiments.

In Fig. 5(e) and 5(f), we show the time needed by each approach to complete the insertion of 10^3 and 10^4 triples that either follow a uniform or a Zipfian distribution. In BC, this time represents the time needed until all inserted triples are stored at the respective nodes. In FC, we also take into account the time spent for the inferred triples to be stored in the network. We observe here that when we go from 10^3 to 10^4 initial triples, there is a blow up in FC's storage time (y-axis is in logarithmic scale). Generally, FC needs an enormous amount of time to reach a fixpoint and makes the measurement of inserting 10^5 initial triples infeasible. Our experiment for inserting this number of triples using FC was active for *16 hours* and still a fixpoint had *not* been reached.

Finally, we conducted some measurements concerning the storage load distribution in both algorithms. The results showed that as the depth of the hierarchy increases, the number of classes increases exponentially and more nodes share the load resulting in a more balanced distribution. Furthermore, BC distributes the load slightly better than FC for larger tree depths. This is a result of a characteristic property of FC, namely that classes of higher levels of the hierarchy have more instances than classes from lower levels (since each class keeps all the instances of its subclasses). Therefore nodes that are responsible for classes of higher levels are more loaded with triples than nodes responsible for classes of



(a) bandwidth and messages (b) bandwidth (c) query response time

Fig. 6. Querying 10^4 instances

lower levels. Due to space limitations we are not presenting these graphs and we consider as future work further improvements [16, 4] in both approaches that will distribute the storage load more evenly among nodes.

Querying RDFS class hierarchies. In this experiment, we present results while evaluating queries of the form "give me all instances of the root class". We generated and stored 10^4 instances for both uniform and Zipfian distributions under an RDFS class hierarchy of varying depth. We run 100 queries of the above form and averaged the measurements taken.

Figure 6(a) shows two metrics concerning the network traffic; bandwidth (left y-axis) and messages sent (right y-axis). Since FC sends a single message regardless of the tree-depth, we do not depict it in the graph. In BC, the number of messages sent in the network is analogous to the number of classes in the RDFS hierarchy since it sends one message for each class. Therefore, as the hierarchy becomes deeper, the number of classes increases and the total number of messages sent also increases. This was also shown by the analytical cost model of Section 4.2 and is depicted in the graph as well. Figure 6(a) also shows the bandwidth used in the network for both approaches and for both types of instance distribution. FC bandwidth consumption is limited to the number of bytes sent for the delivery of the results of the query and is independent of both the tree-depth and the instance distribution. In comparison, the bandwidth of BC increases with the tree-depth as a result of the increasing number of messages that are sent in the network. Furthermore, a skewed instance distribution slightly affects the bandwidth used since more instances belong to lower classes and need to be sent towards the root class.

We also experimented with the query load. As already shown in the analytical evaluation the total query load occurred is similar for both approaches and only the distribution differs. While FC involves a single node in the query processing, in BC the load is shared among the nodes that are responsible for the subclasses of the root class. Due to space limitations we have omitted these graphs. As ongoing work, we are exploring various load balancing techniques [16, 4] for both approaches.

Optimizations. In this section, we measure the effect of a caching optimization technique for the BC algorithm. Since there is a need to traverse the subclass hierarchy quite often, we could take advantage of the first time it is traversed and cache useful routing information. Assuming that different nodes are responsible

for different classes in a hierarchy, we need to make $d*O(logn)$ hops to reach from a node responsible for the root class of the hierarchy to the nodes responsible for the leaf classes, where d is the depth of the hierarchy tree and n the number of the nodes in the network. We can minimize this by adding extra routing information to each node x which is responsible for a certain class C of the hierarchy. The first time a node x contacts a node y which is responsible for a subclass of C, it keeps the IP address of y and uses it for further communication. In this way, a direct subclass is found in just 1 hop and the whole hierarchy is traversed in d hops. This technique minimizes network traffic and decreases query response time, while the overhead of maintaining such a table is not significant and it is only kept in memory.

In Fig. 6(b), we show how the bandwidth of BC is decreased when using this caching technique. Finally, Figure 6(c) shows the query response time. A query needs an almost constant time to be evaluated when in FC. This is reasonable, considering that only one node participates in the query processing, although the query processing it needs to handle is quite heavy. On the contrary, BC response time increases with the tree-depth. However, using the caching technique we manage to improve query response time at a satisfying degree.

6 Related Work

A representative centralized RDF store that supports the forward chaining approach is Sesame [6]. Each time an RDF Schema is uploaded in Sesame, an inference module computes the schema closure of the RDFS and asserts the inferred RDF statements. Jena [27] can support both approaches depending on the underlying rule engine. RSSDB [2] follows a totally different approach in which the taxonomies are stored using the underlying DBMS inheritance capabilities so that retrieval is more efficient. Nevertheless, this approach is still an *on demand* approach and resembles the backward chaining evaluation algorithm. 3store [12] follows a hybrid approach in order to gain from the advantages of both approaches. Finally, in Oracle RDBMS [8], RDFS inference is done at query execution time using appropriate SQL queries.

From the distributed point of view, RDFPeers [7], GridVine [1], and [19] consider RDF query processing on top of structured overlay networks, such as DHTs. [7] and [19] consider no RDFS reasoning while [1] provides semantic interoperability through schema mappings. The only DHT-based RDF store that is closely related with our work is BabelPeers [5] which enables RDFS reasoning. It is implemented on top of Pastry [23] and only a forward chaining approach is supported. The reasoning process runs in regular intervals on each node and checks for new triples that have arrived to the node. Then, it exhaustively generates new inferred triples based on the RDFS entailment rules and sends them to be stored in the network. [5] presents no experimental evaluation of the forward chaining algorithm. However, the results of our experiments show how expensive FC is in terms of storage load, time and bandwidth.

7 Conclusions and Future Work

We presented and evaluated both analytically and experimentally two algorithms, namely forward and backward chaining, that enable RDFS reasoning

on top of DHTs. We proposed the first distributed backward chaining algorithm on top of DHT-based RDF stores together with an optimization which improves significantly query response time. A main result of our experiments is that a simple forward chaining implementation as it is also supported in BabelPeers [5] cannot scale and thus related optimizations should be considered.

In future work, we plan to evaluate exhaustively the query processing algorithms of Atlas [15, 19] taking into account RDFS reasoning as presented in this paper and examine how various load balancing techniques [4, 16] can be applied to our algorithms. We also plan to investigate how forward chaining can be made more efficient as well as hybrid approaches that combine forward and backward chaining.

References

1. K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. V. Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *WWW 2004*.
2. S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis. On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs. In *WebDB 2001*.
3. H. Balakrishnan, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM (2003)*.
4. D. Battre, F. Heine, A. Hoing, and O. Kao. Load-balancing in P2P based RDF stores. In *SSWS 2006*.
5. D. Battre, A. Hoing, F. Heine, and O. Kao. On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores. In *DBISP2P 2006*.
6. J. Broekstra and A. Kampman. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC 2002*.
7. M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW 2004*.
8. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB 2005*.
9. R. Cyganiak. A relational algebra for SPARQL, 2005. Technical Report.
10. Y. Guo, Z. Pan, and J. Heflin. An Evaluation of Knowledge Base Systems for Large OWL Datasets. In *ISWC 2004*.
11. Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
12. S. Harris and N. Gibbins. 3Store: Efficient Bulk RDF Storage. In *PSSS 2003*.
13. P. Hayes. RDF Semantics. W3C Recommendation, February 2004.
14. F. Heine, M. Hovestadt, and O. Kao. Processing Complex RDF Queries over P2P Networks. In *P2PIR 2005*, November 2005.
15. Z. Kaoudi, I. Miliaraki, M. Magiridou, E. Liarou, S. Idreos, and M. Koubarakis. Semantic Grid Resource Discovery in Atlas. *Knowledge and Data Management in Grids, 2006*.
16. D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA 2004*.
17. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *WWW 2002*.
18. O. Lassila. Taking the RDF Model Theory Out For a Spin. In *ISWC 2002*.
19. E. Liarou, S. Idreos, and M. Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In *ISWC 2006*.
20. A. Polleres. From SPARQL to Rules (and back). In *WWW 2007*.
21. E. Prud'hommeaux and A. Seaborn. SPARQL Query Language for RDF.
22. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference*, 2004.
23. A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale- Peer-to-Peer Storage Utility. In *Middleware 2001*.
24. H. Stuckenschmidt and J. Broekstra. Time-Space Trade-offs in Scaling up RDF Schema Reasoning. In *WISE Workshop 2005*.
25. Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *ISWC 2005*. http://athena.ics.forth.gr:9090/RDF/RBench/.
26. J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
27. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Raynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB 2003*.