J Data Semant (2013) 2:189–227
DOI 10.1007/s13740-013-0018-0

ORIGINAL ARTICLE

# Distributed RDFS Reasoning Over Structured Overlay Networks

**Zoi Kaoudi · Manolis Koubarakis**

Received: 23 February 2012 / Revised: 10 December 2012 / Accepted: 6 February 2013 / Published online: 21 March 2013

**Abstract** In this paper, we study the problem of distributed RDFS reasoning over structured overlay networks. Distributed RDFS reasoning is essential for providing the functionality that Semantic Web and Linked Data applications require. Our goal is to present various inference techniques for RDFS reasoning in a distributed environment, and analyze them both theoretically and experimentally. The reasoning methods we present are based on bottom-up and top-down techniques and have been implemented on top of the distributed hash table Bamboo. Our algorithms range from forward and backward chaining ones to rewriting algorithms based on magic sets. We formally prove the correctness of the algorithms and study the time-space trade-off they exhibit analytically and experimentally in a local cluster.

**Keywords** RDFS reasoning · DHT · Forward chaining · Backward chaining · Magic sets · Datalog

## 1 Introduction

With the interest in Semantic Web applications rising rapidly, the Resource Description Framework (RDF) [50] and its accompanying vocabulary description language, RDF Schema (RDFS) [9], have become one of the most widely used data models for representing and integrating structured information in the Web. The Linked Data initiative[1], which aims at connecting data sources on the Web, has already become very popular and has exposed many datasets using RDF and RDFS. DBpedia[2], BBC music information [41], government datasets[3] are only a few examples of the constantly growing Linked Data cloud. Therefore, there is an emerging need not only for dealing with a huge amount of distributed data expressed in RDF, but also for being able to infer new information from it.

Reasoning algorithms have been widely studied in the past in the areas of logic and artificial intelligence. Two important reasoning techniques are *forward chaining* and *backward chaining*. Forward chaining works in a bottom-up fashion. It starts from a given dataset and using the inference rules produces new data that is entailed by the given dataset. Backward chaining works in a top-down manner. It starts from a goal and finds a proof of this goal using the given dataset and the inference rules. In this paper, we study similar reasoning algorithms for RDFS.

Previous work on *centralized* RDF stores has considered forward chaining, backward chaining and hybrid approaches to implement RDFS reasoning and query processing [4,11,24,76]. In the forward chaining approach, new RDF statements are exhaustively generated from the asserted ones until the full RDFS closure is computed. 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 which materializes all inferences has minimal requirements during query answering, but

Z. Kaoudi (✉) · M. Koubarakis
Department of Informatics and Telecommunications,
National and Kapodistrian University of Athens, Athens, Greece
e-mail: zoi.kaoudi@inria.fr

M. Koubarakis
e-mail: koubarak@di.uoa.gr

*Present address*
Z. Kaoudi
INRIA Saclay, Orsay, France

[1] http://linkeddata.org/
[2] http://dbpedia.org
[3] http://www.data.gov/, http://data.gov.uk/

needs a significant amount of storage for all the inferred data. On the other hand, a backward chaining approach has minimal storage requirements, at the cost of an increase in query response time. However, in the case of frequent updates in the RDFS database, computing the RDFS closure may become very expensive and time-consuming and can be outperformed by the backward chaining approach. There is a time-space trade-off between these two approaches [68], and only by knowing the query and update workload of an application, one can determine which approach would suit it better. One of the challenges that we undertake in this paper is studying this trade-off in a distributed environment, and more specific using distributed hash tables.

P2P networks and especially distributed hash tables (DHTs) [5] have gained much attention in the past years, given the fault-tolerance and robustness features they can provide to Web-scale applications. DHTs have been proposed for the storage and querying of RDF data at Web scale by [1,12,27,45]. However, these works are solely concerned with query processing for RDF data, and pay no attention to RDFS reasoning. The first DHT-based RDF store that has dealt with RDFS reasoning in the past is BabelPeers [7,27]. It is implemented on top of Pastry [60] and supports a subset of the SPARQL query language [58]. BabelPeers uses a forward chaining approach to provide the RDFS inference capability required to answer the supported class of SPARQL queries.

Apart from DHTs, other distributed and parallel computing platforms have been proposed lately for the RDFS reasoning. Marvin [55,56] supports a forward chaining approach for RDFS reasoning and runs on DAS-3 (Distributed ASCI Supercomputer). In [72], a different forward chaining approach is proposed based on MapReduce [15]. The work in [75] considers the problem of producing the full RDFS closure of a given dataset using parallel computing techniques such as workload partitioning.

The above recent approaches demand locally deployed high-end infrastructures whose cost can be very high in many cases. Our work focuses on DHT-based algorithms which can also run on commodity machines deployed all over the world, as it is the case with many other P2P applications.

*Contributions.* 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 [59]. In addition, we present an algorithm which works in a bottom-up fashion using the magic sets transformation technique [8]. Our algorithms have been integrated in the system Atlas[4], a full-blown open source P2P system for the distributed processing of SPARQL queries on top of DHTs. In this paper, we only concentrate on the functionality offered by Atlas for RDFS reasoning. Discussions of the architecture, query optimization techniques and various applications of Atlas can be found in [33,34,36,37].

To the best of our knowledge, our backward chaining algorithm, originally presented in [35], is the first distributed backward chaining algorithm proposed for RDFS reasoning in a decentralized environment. Moreover, a magic sets transformation technique for distributed RDFS reasoning that we discuss in this paper has not been studied in the literature before. We prove the correctness of our algorithms and provide a comparative performance study both analytically and experimentally.

Preliminary results of this research have appeared in [35]. The current paper revises [35] and presents the following extensions and additional contributions.

- We base our data model on the minimal deductive system *mrdf* of [51] using a Datalog-like notation of the rules. In [51], the authors present a small fragment of the RDFS language which preserves the core functionalities and avoids certain complexities. The set of inference rules of this fragment is truly useful for modeling an application domain, and leaves out vocabulary and inferences that capture the internals of RDF and RDFS.
- We give formal proofs for the termination, soundness and completeness of both forward and backward chaining algorithms based on the semantics and inference rules of [51].
- We propose a slightly different algorithm for the forward chaining approach that deals with an important case of redundant triple generation in the forward chaining algorithm of [35]. Redundant triple generation is a problem in all the forward chaining approaches of distributed RDFS reasoning that have been published in the literature [56,72,75]. In this paper we show how we can avoid redundant triple generation for an important special case in a distributed setting.
- We design and describe the magic sets transformation algorithm [8] for bottom-up RDFS inference and show how it can be implemented in a distributed fashion.
- In the experimental part of our work, we demonstrate the effect of redundant triple generation in our system. In addition, we compare the backward chaining algorithm with the algorithm based on the magic sets transformation and show that the backward chaining algorithm performs better in our system. The behaviour of our system is explored in a local cluster with bigger datasets than the ones we have used in [35].
- We provide an extensive survey of related work in the area of RDFS reasoning.

Since proving correctness of distributed algorithms is an important topic of theoretical distributed systems research

[4] http://atlas.di.uoa.gr

[49], we consider our theoretical analysis to be one of the most important contributions of our work and an advancement in the area of distributed RDFS reasoning. Previous related research in this area has concentrated mostly on practical issues [7,18] and ignored theoretical ones. Proof techniques similar to the ones we develop here can be used to prove the soundness and completeness of the algorithms proposed by related papers, e.g., [62,72]. Thus, we hope that the techniques of this paper will actually motivate other authors to produce similar theoretical results.

The organization of the paper is as follows. Section 2 presents background knowledge required for the comprehension of the rest of the paper. Section 3 presents the architecture of the system as well as some basic protocols that are used by the algorithms presented in this paper. In Sect. 4, we present how a forward chaining algorithm can be implemented in a DHT and give formal proofs for the correctness of the algorithm. In addition, in Sect. 4.5, we discuss the issue of redundant triple generation in forward chaining algorithms for the RDFS reasoning. Section 5 presents a backward chaining algorithm with its correctness proofs, while Sect. 6 describes the algorithm based on the magic sets transformation. In Sect. 7, we give an analytical cost model of our algorithms, while in Sect. 8 we present the results of our experimental evaluation. Finally, Sect. 9 presents a survey of work in the related areas of RDFS reasoning and distributed systems. Section 10 concludes the paper and Sect. 11 discusses open issues future directions.

## 2 Preliminaries

In this section, we present concepts, terminology and results that will be used throughout the paper. We start with the basics of RDF(S) and the SPARQL query language. Then, since we base our data model on the minimal deductive system of [51], we present useful results from [51].

### 2.1 RDF(S) and SPARQL

RDF [50] is a W3C standard and the most widely used data model for representing and integrating structured information in the Semantic Web. In RDF, a Web resource is identified by a Uniform Resource Identifier (URI), and information about resources is encoded using subject-property-object triples. The subject of a triple identifies the resource that the statement is about, the predicate identifies a property or a characteristic of the subject, while the object gives the value of the property. These values can be either URIs or constants from primitive types called literals (such as strings or integers). In addition, RDF allows for blank nodes which are identifiers for unknown values. More formally, we define an RDF triple as follows.

**Definition 1** Let $U$, $L$ and $B$ denote the sets of URIs, literals, and blank nodes, respectively. These sets are pairwise disjoint. An *RDF triple* is a tuple $(s, p, o)$ from $(U \cup B) \times U \times (U \cup L \cup B)$, where $s$ is the subject, $p$ is the property and $o$ is the object of the triple.

We will call a triple *ground* if it contains no blank nodes.

The vocabulary description language developed for RDF is another W3C standard called RDF Schema (RDFS) [10]. RDFS extends RDF to allow grouping and connecting resources by defining classes and properties. RDF data as well as RDFS descriptions (we will further use the term RDF(S) to refer to both) can be written as RDF triples of the above form. We will call a set of RDF(S) triples an *RDF(S) database* or an *RDF(S) graph*. An RDF(S) graph is *ground* if the set of triples of the graph contains only ground triples.

SPARQL [58] is the W3C standard query language used for querying RDF data. The core construct of SPARQL is a basic graph pattern, i.e., a conjunction of triple patterns. A triple pattern is a subject-predicate-object tuple where the components can be either constants or variables. More formally, we define a triple pattern as follows.

**Definition 2** Let $U$, $L$ and $V$ denote the pairwise disjoint sets of URIs, literals and variables, respectively. A *triple pattern* is a tuple $(s, p, o)$ from $(U \cup B \cup V) \times (U \cup B \cup V) \times (U \cup B \cup L \cup V)$.

*Notation* The following abbreviations of some predefined RDF(S) URIs will be used in the rest of the paper: `sc` for `rdfs:subClassOf`, `sp` for `rdfs:subPropertyOf`, `type` for `rdf:type`, `dom` for `rdfs:domain` and `range` for `rdfs:range`.[5]

### 2.2 A Minimal Sound and Complete Deductive System for RDF(S)

To support RDFS reasoning, one could use the sound RDFS inference rules presented in RDF Semantics [26] or their extension given in [29] which is also complete. We choose to base our work on the minimal set of inference rules of [51] which we briefly present here. This set of inference rules covers only the subset of the RDFS vocabulary which is truly useful for modeling an application domain, and leaves out vocabulary and inferences that capture the (possibly complicated) internals of RDF and RDFS.

In [51,52], the authors start with a subset of RDF and RDFS vocabulary which they call $\rho df$. The only RDFS

[5] Namespaces `rdf` and `rdfs` are the namespaces of the core RDF and RDFS vocabulary defined by the URIs http://www.w3.org/1999/02/22-rdf-syntax-ns and http://www.w3.org/2000/01/rdf-schema, respectively.

**Table 1** $\rho df$ inference rules

| | | |
|---|---|---|
| 1 (simple) | (a) $\frac{G}{G'}$ for a map $\mu : G' \to G$ | (b) $\frac{G}{G'}$ for $G' \subseteq G$ |
| 2 (subproperty) | (a) $\frac{(A,sp,B)(B,sp,C)}{(A,sp,C)}$ | (b) $\frac{(A,sp,B)(X,A,Y)}{(A,B,Y)}$ |
| 3 (subclass) | (a) $\frac{(A,sc,B)(B,sc,C)}{(A,sc,C)}$ | (b) $\frac{(A,sc,B)(X,type,A)}{(X,type,B)}$ |
| 4 (typing) | (a) $\frac{(A,dom,B)(X,A,Y)}{(X,type,B)}$ | (b) $\frac{(A,range,B)(X,A,Y)}{(Y,type,B)}$ |
| 5 (implicit typing) | (a) $\frac{(A,dom,B)(C,sp,A)(X,C,Y)}{(X,type,B)}$ | (b) $\frac{(A,range,B)(C,sp,A)(X,C,Y)}{(Y,type,B)}$ |
| 6 (subproprerty reflexivity) | (a) $\frac{(X,A,Y)}{(A,sp,A)}$ | (b) $\frac{}{(p,sp,p)}$ for $p \in \rho df$ |
| | (c) $\frac{(A,sp,B)}{(A,sp,A)(B,sp,B)}$ | (d) $\frac{(A,p,X)}{(A,sp,A)}$ for $p \in \{dom, range\}$ |
| 7 (subclass reflexivity) | (a) $\frac{(A,sc,B)}{(A,sc,A)(B,sc,B)}$ | (b) $\frac{(X,p,A)}{(A,sc,A)}$ for $p \in \{dom, range\}$ |

predefined terms allowed in the $\rho df$ vocabulary are `sp`, `sc`, `dom`, `range` and `type`. Muñoz et al. [51] and Munoz et al. [52] give a deductive system over a minimal set of inference rules over $\rho df$ and prove that it is sound and complete. For ease of the reader, we present below some of the definitions and results of [51] that we will use throughout the paper. Table 1 shows the inference rules used for the $\rho df$ fragment.

We now give the definition of a $\rho df$ proof. First, we define the concept of a map. A *map* is a function $\mu : U \cup B \cup L \to U \cup B \cup L$ preserving URIs and literals, i.e., $\mu(u) = u$ for all $u \in U \cup L$. Given an RDF(S) graph $G$, $\mu(G)$ is defined as the set of all triples $(\mu(s), \mu(p), \mu(o))$ such that $(s, p, o) \in G$. In [51], the authors overload the meaning of a map and speak of a map $\mu$ from $G_1$ to $G_2$ ($\mu : G_1 \to G_2$), if the map $\mu$ is such that $\mu(G_1)$ is a subgraph of $G_2$.

**Definition 3** Let $G$ and $H$ be RDFS graphs. We will say that there is a *$\rho df$ proof* of $H$ from $G$ (denoted by $G \vdash_{\rho df} H$) iff there exists a sequence of graphs $P_0, P_1, \ldots, P_k$, with $P_0 = G$ and $P_k = H$, and for each $j$ ($1 \leq j \leq k$) one of the following cases holds:

- there exists a map $\mu : P_j \to P_{j-1}$ (rule (1a)),
- $P_j \subseteq P_{j-1}$ (rule (1b)),
- there is an instantiation $\frac{R}{R'}$ of one of the rules (2)–(7), such that $R \subseteq P_{j-1}$ and $Pj = P_{j-1} \cup R'$.

Muñoz et al. [51] constrain the $\rho df$ subset further by disallowing $\rho df$ vocabulary as subject or object of a triple. The new subset of RDFS is called minimal RDFS and denoted by $mrdf$. This is the subset of RDFS we will use in this paper. A *minimal RDFS triple (mrdf-triple)* is a ground $\rho df$ triple having no $\rho df$ vocabulary as subject or object [51]. A *mrdf-graph* is a set of mrdf-triples. Based on this notion and the Definition 3 of a $\rho df$ proof, the authors of [51] define a $mrdf$ proof as follows.

**Definition 4** Let $G, H$ be mrdf-graphs. We call a $mrdf$ proof of $H$ from $G$, $G \vdash_{mrdf} H$, if and only if there is a $\rho df$ proof of $H$ from $G$ involving solely the rules (1b), (2), (3) and (4).

Each pair $(P_{j-1}, P_j)$, $1 \leq j \leq k$ is called a *$mrdf$ step* of the proof or *$mrdf$ proof step*, which is labeled by the respective instantiation of the rule applied.

Let $\models$ be the RDFS entailment relation defined in [26]. The following result presented in [51] shows that the normative semantics of RDFS are preserved by the deductive system $\vdash_{mrdf}$ for those mrdf-graphs that do not contain triples of the form $(x, sp, x)$ or $(x, sc, x)$ for $x \in U \cup L$.

**Theorem 1** *Let $G$ and $H$ be mrdf-graphs. Assume that $H$ does not contain triples of the form $(x, sp, x)$ nor $(x, sc, x)$ for $x \in U \cup L$. Then $G \models H \Leftrightarrow G \vdash_{mrdf} H$.*

In the rest of the paper, we base our work on the minimal RDFS fragment $mrdf$ of [51]. Following the above theorem and by assuming ground graphs, our algorithms focus on the $\rho df$ inference rules 1(b) and (2)–(4) of Table 1. In addition, we do not consider reflexive triples such as $(x, sc, x)$ or $(x, sp, x)$. We will use the deductive system for $mrdf$ to prove that our algorithms are sound and complete.

## 3 System Description and Data Model

The algorithms presented in this paper are based on structured overlay networks where nodes are organized according to a DHT protocol. 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 are assigned a unique $m$-bit identifier 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 Pastry [60], when a node receives a PUT request, it efficiently routes the request to a node with 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 $O(log n)$ hops, where $n$ is the number of nodes in the network. More details about the routing protocol of Pastry can be found in [60]. Bamboo [59] improves on Pastry by using more incremental algorithms for node joins and neighbour management. This allows Bamboo to withstand very dynamic changes in network membership i.e., it is resilient to churn.

The algorithms described in this paper have been integrated in the system Atlas[6], a full-blown open source P2P system for the distributed processing of SPARQL queries on top of DHTs. All nodes in the network are equal and can accept either a request for storing RDF(S) data in the system or a request for evaluating a SPARQL query. The user can insert both RDF data and RDFS ontologies in the form of RDF documents which can be expressed in RDF/XML or NTRIPLE format.

## 3.1 Data Model and Query Language

Our system handles both RDF and RDFS data containing $mrdf$ graphs as defined in Sect. 2.2.

The query language supported by our system is a subset of SPARQL queries and, in particular, basic graph pattern queries. A *basic graph pattern* (BGP) SPARQL query is a conjunction of triple patterns and can be expressed as follows:

$$?x_1, \ldots, ?x_k : (s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge \cdots \wedge (s_n, p_n, o_n),$$

where $?x_1, \ldots, ?x_k \in V$ are variables and $(s_i, p_i, o_i)$ is a triple pattern where at least one of the $s_i, p_i, o_i$ is a constant. Variables $?x_1, \ldots, ?x_k$ are called *answer variables* and each variable $?x_i$ appears in at least one triple pattern. In the description of the reasoning algorithms we present below, we focus on atomic queries to demonstrate the differences among the various algorithms. It is easy to incorporate these reasoning techniques to query processing algorithms for evaluating BGP SPARQL queries, such as the ones proposed in [37,45,46], where the decomposition of the queries always leads to the evaluation of atomic queries.

## 3.2 Architecture

A higher level view of each node's architecture as implemented in Atlas is shown in Fig. 1. On the top layer of the architecture, the API enables a user to store RDF(S) data in the network or pose a SPARQL query. The basic components of Atlas are the store and query processors. Upon receiving a store request, the storage manager of the node is responsible for decomposing the document into RDF(S) triples and initiating the protocol that distributes the triples in the network. The triples are indexed in various nodes of the network according to a specific indexing scheme explained later. If forward chaining is chosen as the reasoning scheme, the RDFS engine is responsible for making the appropriate inferences according to the $\rho df$ inference rules. Then, the store processor distributes the inferred triples in the network as well. Every triple that arrives at the responsible node is stored in the local RDF(S) database of this node.

When a node receives a query request, the SPARQL translator transforms the query to an equivalent conjunctive query form based on the system's internal query representation. Then, the query evaluation engine is responsible for the distributed evaluation of the query. If backward chaining is the chosen reasoning scheme, the RDFS engine is also in charge of carrying out the distributed reasoning taking into account the $\rho df$ inference rules. Both the store processor and the query processor are able to communicate with the local database of each node. We have used SQLite[7] as the local database of each node[8]. At the lower levels of the architecture lies the Bamboo network layer which is responsible for the communication among the nodes.

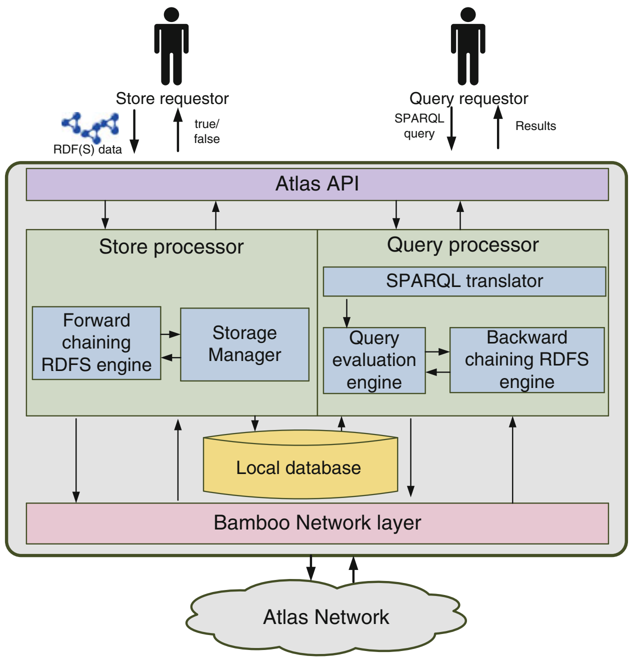


**Fig. 1** System architecture

6 http://atlas.di.uoa.gr

7 http://www.sqlite.org/

8 In previous versions of Atlas, the Berkeley DB database was used which is included in the Bamboo implementation and used by Bamboo for various database tasks. However, we found that this implementation is inefficient and have moved to SQLite, a lightweight relational database instead.

In this paper, we concentrate on the functionality offered by Atlas for RDFS reasoning and discuss only the relevant components. A complete system-level description of Atlas can be found in [36]. As we have already mentioned, Atlas uses the Bamboo DHT [59]. However, our algorithms for RDFS reasoning to be presented in Sect. 4, 5, 6 are DHT-agnostic; they can be implemented on top of any DHT network.

### 3.3 Basic Storing Protocol

In the following, we describe the basic protocol used for indexing RDF(S) triples in the network. When the backward chaining approach is used, this protocol is used exactly as described below, while, when forward chaining is used, it is augmented with the computation of additional inferred triples as described in Sect. 4.

We have adopted the triple indexing algorithm originally presented in [12] where each triple is indexed in the DHT *three times*. This algorithm is by now standard in DHT-based systems for RDF(S) stores. The hash values of the subject, property and object of each triple are used to compute the identifiers that will indicate the nodes responsible for storing the triple. Whenever a node receives a request to store a set of triples, it sends three DHT PUT requests for each triple, using as key the subject, property and object, respectively, and the triple itself as the item. The key is hashed using hash function SHA-1 [63] 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. When a node is responsible for a key which is a class name `C` (*responsible node for class* `C`), it 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). Each node keeps its triples in its local database consisting of a single relation with four columns (*triple relation*). The first three columns correspond to the three components of the triples stored, while the fourth column indicates which of the three components is the key that led the triple to this node.

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 $triples$.

We should point out that we handle data and schema triples in a uniform way. While other approaches, such as [18,72, 75], require that every node in the system keeps all the RDFS triples, and in some cases in main memory, in our system we adopt a more generic approach where no global knowledge about the schema is required. RDFS triples are distributed in the same way that RDF triples are and all of them are stored on the local SQLite database of the relevant node.

### 3.4 Basic Querying Protocol

In this section, we describe the basic protocol used in our system to answer an atomic SPARQL query consisting of a *single* triple pattern `(s,p,o)` where at least one of `s`, `p`, `o` is a constant.

Whenever a node receives a query request, it should decide to which node it should route the request to evaluate the query. The query requestor node chooses a *key* from the triple pattern and hashes 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, the query requestor node selects the keys in the order "subject, object, property" based on the fact that we prefer keys with lower selectivity and the assumption that subjects are more selective than objects which are more selective than properties[9]. At the destination node, all triples that contain this key will be found in the local database due to our indexing scheme. The triple pattern will be matched with these triples and the bindings of the triple pattern's variables will be returned to the requestor node.

This protocol is used as is for the forward chaining approach and the magic sets rewriting algorithm. For the backward chaining approach, it is augmented with the appropriate actions to include the $\rho df$ inference rules as described in Sect. 5.

Note that we do not deal with answering queries that are triple patterns with no constant parts. The answer to such queries is the whole database of triples stored in all the nodes of the network. These queries can be computed by a broadcasting algorithm but this is out of the scope of this paper.

## 4 Distributed Forward Chaining

In this section, we describe our forward chaining algorithm and its implementation in a DHT. In all the algorithms of this paper, we choose to encode the $\rho df$ inference rules as Datalog rules and use ideas from Datalog query evaluation. Thus, we start by introducing the relevant Datalog notation.

### 4.1 Datalog Rules

Following the notation of Datalog with extensional database relation (edb) `triple` and the intensional database relation

[9] This simple scheme was previously used in [45,66] and suffices for the purposes of this paper. In (Kaoudi et al. 2010b) we study relevant issues in the context of conjunctive triple pattern queries using selectivities.

**Table 2** $\rho df$ inference rules in Datalog

| Rule | Head | Body |
|---|---|---|
| 1 | `newTriple(X, P, Y)` | `triple(X, P, Y)` |
| 2 | `newTriple(X, sp, Y)` | `triple(X, sp, Z), newTriple(Z, sp, Y)` |
| 3 | `newTriple(X, P, Y)` | `triple(X, P1, Y), newTriple(P1, sp, P)` |
| 4 | `newTriple(X, sc, Y)` | `triple(X, sc, Z), newTriple(Z, sc, Y)` |
| 5 | `newTriple(X, type, Y)` | `newTriple(X, type, Z), triple(Z, sc, Y)` |
| 6 | `newTriple(X, type, Y)` | `newTriple(X, P, Z), triple(P, dom, Y)` |
| 7 | `newTriple(X, type, Y)` | `newTriple(Z, P, X), triple(P, range, Y)` |

**Algorithm 1**: FC*: Forward chaining algorithm

```
1  event $n$.StoreMsg($id, triples, k, inf$)
       /*$infTriples$: list holding all inferred triples */
2      $localTriples$= $triples \cup$ GetTriplesFromDB ($id$);
3      $newTriples$=Infer ($localTriples$);
4      $pairs = \{\}$;
5      forall $t \in \{newTriples \setminus infTriples \setminus localTriples\}$ do
6          $pairs.put(t.subject, t)$;
7          $pairs.put(t.property, t)$;
8          $pairs.put(t.object, t)$;
9      forall $k' \in pairs.keys()$ do
10         $id'$ =Hash ($k'$);
11         $triples' = pairs.get(k')$;
12         sendto $id'$.StoreMsg($id', triples', k', true$);
13         $infTriples$.add($triples'$);
14     InsertToDB($triples, inf$);
15 end event
```

(idb) `newTriple`, the $\rho df$ inference rules can be written as shown in Table 2. The edb relation `triple` denotes triples that are explicitly given in an RDF(S) dataset, while the idb relation `newTriple` denotes triples that have been inferred by the rules. An important aspect of the above set of rules is that all rules are *linear* (with at most one recursive predicate in their body) and *safe* (all variables appear as an argument in the predicates of the rule bodies).

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. Rule predicate names always start with a lowercase letter. To avoid confusion, we refer to the second element `p` of an RDF triple `(s, p, o)` with the word *property* and to a rule predicate name with the word *predicate* or *relation*. In comparison with the rules of the deductive system $\rho df$ of [51], rule 1 is actually rule (1b) (simple), rules 2 and 3 represent rules (2) (subproperty), rules 4 and 5 represent rules (3) (subclass), and rules 6, 7 represent rules (4) (typing).

## 4.2 Algorithm Description

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 of the handler or the procedure is prefixed by the node identifier in which the handler or the procedure is executed. Local procedure calls are not prefixed with the node identifier. 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 (thus the node is immediately contacted), and the message we receive from a node, respectively.

Our forward chaining algorithm, called FC*, is activated every time a set of RDF(S) triples is inserted in the network. As explained in Sect. 3.3, whenever a node receives a request to store a set of triples $G$, it sends three DHT Put requests for each triple, using as key the subject, property and object respectively, and the triple itself as the item. In the case of FC*, instead of using a Put request, we use a variation of the MultiPut request, i.e., a StoreMsg message. Algorithm 1 shows in pseudocode how FC* works. Suppose a StoreMsg($id, triples, k, inf$) request arrives at node $n$ which is responsible for the identifier $id$ and a set of triples $triples$ should be stored in the local database of $n$. $k$ is the key that led $triples$ to this node and $inf$ is a boolean value that indicates whether $triples$ are inferred triples or not. First, node $n$ retrieves from the local database all triples that contain the key $k$ either as a subject, property or object and puts them in list $localTriples$ together with $triples$. Then, it computes the inferred triples from this list according to the Datalog rules of Table 2 using local function Infer($localTriples$). Function Infer($localTriples$) assigns the triples which have originated from the initial RDF(S) graph to the edb relation `triple` and the triples which have been inferred from an inference rule to the idb relation `newTriple`[10]. It outputs the idb relation `newTriple` by matching the triples with the antecedent of the rules of Table 2 in a data-driven manner. The new facts of relation `newTriple` generated by function Infer form a list $infTriples$ with all inferred triples. In this way, the node can check when it

[10] In the local database of each node there is information of whether a triple is inferred or not.

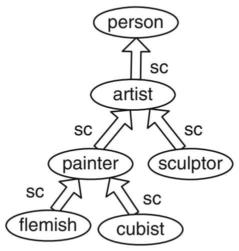


**Fig. 2** Example RDF(S) class hierarchy

reaches a fixpoint where no new triples can be generated. Node $n$ groups the newly inferred triples based on their distinguished keys and puts them in a map $pairs$ (lines 5–8). For each unique key $k'$ and newly triples $triples'$ in the map $pairs$, a new StoreMsg request is sent to the network. The initial set of triples $triples$ is stored in node's $n$ local database using local function InsertToDB. The algorithm terminates when all nodes have reached a fixpoint.

Invoking FC* every time new triples are stored in the network allows us to compute the closure of the stored triples under the $\rho df$ inference rules and the $mrdf$ semantics (we prove this formally below). After FC* has terminated, query evaluation can be performed exactly as described in Sect. 3.4. The query request is routed to the node that is responsible for the key of the triple pattern and all triples (initial and inferred) matching this triple pattern are found locally at its database.

Figure 2 depicts a small RDFS class hierarchy of the cultural domain [40]. In Fig. 3, we demonstrate an example of how FC* works based on this RDFS hierarchy. Figure 3a shows the initial triples that are indexed in the nodes of the network. The key of each triple that led to a specific node is underlined. Nodes $n_1$ and $n_2$ infer two triples each using rules 1 and 4 of Table 2 (Fig. 3b). Inferred triples are shown in bold. These triples are sent to be stored to the corresponding nodes of the network. Then, node $n_2$ will infer two more triples by considering the already inferred triple `(painter, sc, person)` and the two initial triples stored in its local database using rule 4 of Table 2 (Fig. 3c). These two triples are finally sent to be stored at nodes $n_4$, $n_5$ and $n_6$. The final state of the nodes' databases is depicted in Fig. 3d.

Suppose now that a user submits the query "Find all the subclasses of class artist" to a node of the network. This query can be expressed as the triple pattern `(X, sc, artist)` and will be routed to node $n_1$, which is responsible for the key `artist`. The answer will be formed at this node since all triples will be found at its local database.

Note that in Fig. 3, we have omitted showing the triples indexed by their property for simplicity. In this case, all triples of the RDFS hierarchy are also indexed to the node responsible for the key `sc` and thus, all inferences are also generated by this node. This leads to sending redundant messages to the network, an issue we further discuss in Sect. 4.5 and demonstrate in our experimental evaluation.

## 4.3 Termination, Soundness and Completeness

In this section, we give formal proofs for the termination, soundness and completeness of algorithm FC*.

**Theorem 2** (Termination) *Algorithm FC* terminates.*

*Proof* Nodes in the network execute the for loop of lines 4–12 of FC* a finite number of times. The loop is executed a finite number of times since there is a finite upper bound on the number of triples entailed by a given finite $mrdf$ graph $G$ ($O(|G^2|)$ according to [51]). The loop propagates inferred triples in other nodes of the network hence FC* terminates. □

In the following, we prove that algorithm FC* is sound and complete under the assumptions that network is stable and messages are always delivered after a finite amount of time to have clear semantics. We discuss issues that arise at the presence of node failures at Sect. 4.4.

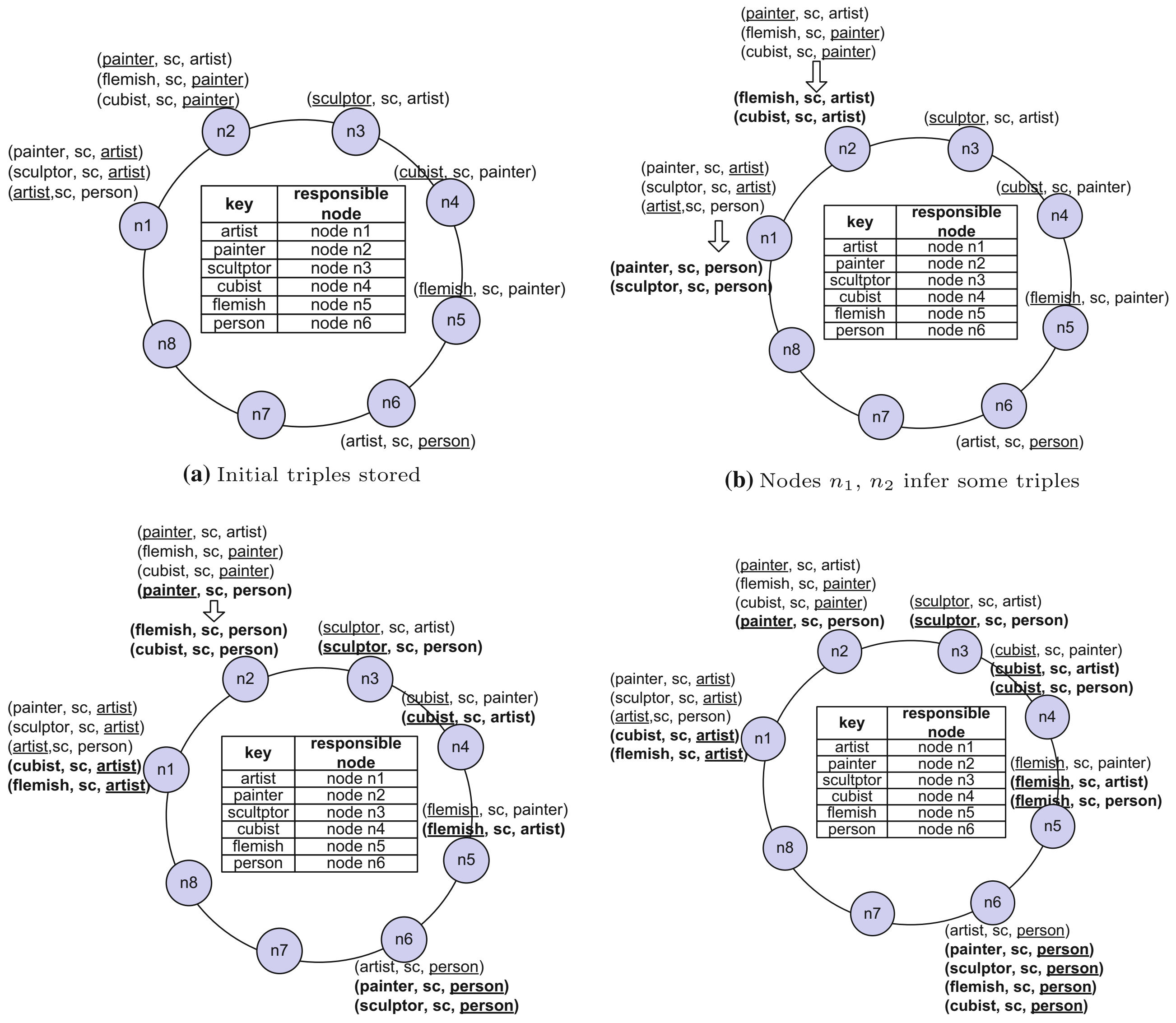
By *sound*, we mean that if $H$ is the RDF(S) graph produced by the FC* algorithm and stored in the network, then $G \models H$ where $G$ is the initial graph. By *complete* we mean that if $G$ is the RDF(S) graph initially stored in the network, $H$ any graph and $G \models H$, then all triples of $H$ will be stored in the network after the completion of FC*.

In the proofs of soundness for the forward and backward chaining algorithms, we will use the notion depth of an appropriate kind of finite tree that captures the relevant computation. We define this notion as usual. The *depth of a node* in a tree is the length of the path from the root to this node. The *depth of a tree* is the maximum depth of a node in the tree.

An execution of FC* on top of a DHT can be modeled using the following notion of computation tree.

**Definition 5** A *computation tree* is a finite tree with the following properties:

(1) Every node is of the form $a_i : (H^i_{\text{old}}, H^i_{\text{new}})$ where $a_i$ is a DHT node and $H^i_{\text{old}}, H^i_{\text{new}}$ are non-empty sets of triples.
(2) For every node $a_i : (H^i_{\text{old}}, H^i_{\text{new}})$ we have the following:
   - $H^i_{\text{old}}$ is a set of triples stored at the local database of node $a_i$.
   - $H^i_{\text{new}}$ is the set of triples computed at node $a_i$ during some step of FC* by applying some $mrdf$ rules in parallel.
(3) For every child $a_{i+1} : (H^{i+1}_{\text{old}}, H^{i+1}_{\text{new}})$ of node $a_i$, we have that $H^i_{\text{new}} \cap H^{i+1}_{\text{old}}$ is non-empty and is the set of new triples produced at node $a_i$ during some step of FC* and sent to node $a_{i+1}$ where FC* continues.



**(a)** Initial triples stored

**(b)** Nodes $n_1$, $n_2$ infer some triples

**(c)** Node $n_2$ infers some more triples

**(d)** Final triples stored

**Fig. 3** FC* running example

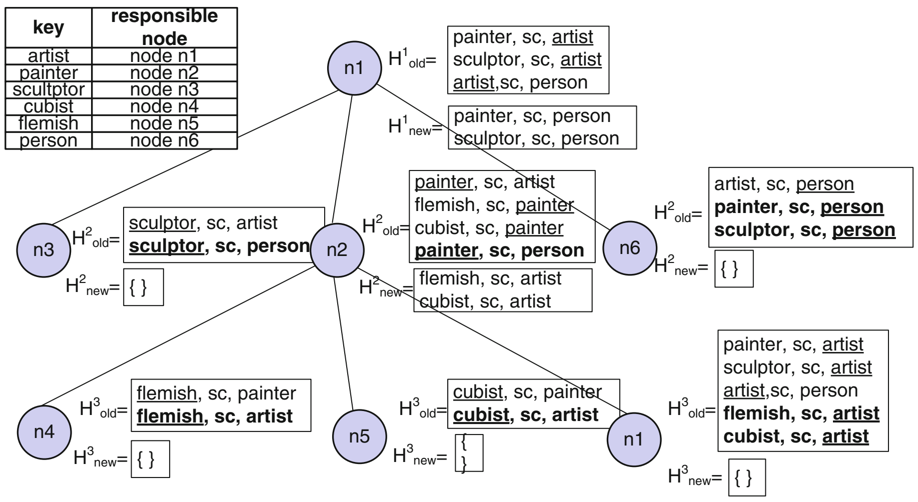
A computation tree offers a nice pictorial representation of the computation of FC*. Figure 4 shows the computation tree for the example shown in Fig. 3. The root of the tree corresponds to the activation of FC* when a message StoreMsg arrives at a DHT node. The nodes of the tree at increasing depths allow us to understand FC* proceeding in rounds although no such strong assumption is used by the algorithm.

**Theorem 3** (Soundness) *Let $G$ be the initial graph stored in the network and $H$ the set of triples in the local databases of nodes after FC* has terminated. Then it holds that $G \models H$.*

*Proof* The proof is by induction on $d$, the depth of the computation tree representing the execution of FC*.

Base case: $d = 1$ (the root is $d = 0$). In this case, FC* runs at the network node represented by the root of the tree, produces $k$ new sets of triples and sends them to $k$ nodes. FC* runs again at these $k$ nodes, but no new triples are computed. A sequence of graphs ($mrdf$ proof according to Definition 4) that shows $G \vdash_{mrdf} H$ can easily be constructed. The first element of the sequence is $G$ followed by $k$ subsequences representing the $k$ sets of triples mentioned above (the order of the subsequences does not matter). Each of the $k$ sets of triples can be represented by as many proof steps as the number of triples it contains. Each step is the result of the application of a $\rho df$ rule to the set of triples produced in the immediately previous proof step.

Inductive step: We assume that the theorem holds for executions of FC* with computation trees of depth $d$ and we will show that the theorem holds for executions of FC* with computation trees of depth $d + 1$.



key | responsible node
artist | node n1
painter | node n2
scultptor | node n3
cubist | node n4
flemish | node n5
person | node n6
n1
H1old= painter, sc, artist; sculptor, sc, artist; artist,sc, person
H1new= painter, sc, person; sculptor, sc, person
n2
H2old= painter, sc, artist; flemish, sc, painter; cubist, sc, painter; painter, sc, person
H2new= flemish, sc, artist; cubist, sc, artist
n3
H2old= sculptor, sc, artist; sculptor, sc, person
H2new= { }
n6
H2old= artist, sc, person; painter, sc, person; sculptor, sc, person
H2new= { }
n4
H3old= flemish, sc, painter; flemish, sc, artist
H3new= { }
n5
H3old= cubist, sc, painter; cubist, sc, artist
H3new= { }
n1
H3old= painter, sc, artist; sculptor, sc, artist; artist,sc, person; flemish, sc, artist; cubist, sc, artist
H3new= { }

**Fig. 4** Computation tree of the DHT node n1

We take the computation tree $T$ of depth $d+1$ corresponding to the execution of FC* on the stored graph G and prune all the nodes at depth $d+1$. The resulting tree $T'$ has depth $d$ and the theorem holds for the corresponding execution of FC*. Hence, if $H'$ is the set of triples in the local databases of the DHT nodes as represented by the tree $T'$ then $G \vdash_{mrdf} H'$. Now we can continue the proof of $H'$ from $G$ by adding the steps corresponding to the nodes we pruned from $T$ to arrive at a proof $H$. This is done as in the base case.

Since we have proved $G \vdash_{mrdf} H$, Theorem 1 of Sect. 2 gives us that $G \models H$. □

**Theorem 4** (Completeness) *Let G and H be $mrdf$ graphs with $H$ not containing triples of the form $(x, sp, x)$ nor $(x, sc, x)$ for $x \in U \cup L$. Assume that graph G is stored in the network. If $G \models H$, then the triples of graph H will be also stored in the network when algorithm FC* terminates.*

*Proof* Using Theorem 1 and the fact that $G \models H$, we have that $G \vdash_{mrdf} H$. We will prove the result using induction on $k$, the number of $mrdf$ proof steps that show $G \vdash_{mrdf} H$. Using the notation of Definition 3, we assume that $P_0 = G$ and $P_k = H$.

Base case ($k = 1$): $H$ is derived from $G$ by the application of a single rule $r$ of the deductive system of Table 1.

– If $r$ is rule (1b) then $H \subseteq G$ and thus, all triples of $H$ will be stored in the network.
– If $r$ is one of the rules (2)–(4), then there is an instantiation $\frac{R}{R'}$ of rule $r$ such that $R \subseteq G$ and $H = G \cup R'$. $G$ is stored in the network and hence we need to show that triple $R'$ will also be stored. The instantiation of $r$ will happen at the node where triples which match the antecedent of rules (2)–(4) (i.e., $R$) will meet. If we check Table 2, we observe that the antecedent triples of rules 2–7 always have a common element. Triples are indexed three times based on three identifiers, namely the hash values of their subject, property and object. Therefore, triples with a common element will meet at the node responsible for the identifier of this common element where the antecedent $R$ will be matched. Then, the triple $R'$ will be generated by FC* using local function Infer and will be stored in the network. Thus, all triples included in $H$ will be stored in the network.

*Inductive step* We assume that the result holds for $k-1$. We will show that it holds for $k$.

Let us consider $G$ and $H$ such that $G \vdash_{mrdf} H$ in $k$ proof steps. Using the notation of Definition 3, $H$ can be proved from $P_{k-1}$ by the application of a single rule $r$ of the deductive system of Table 1. The rest of the proof is similar to the one for the base case.

– If $r$ is rule (1b), then $H \subseteq P_{k-1}$ and following the hypothesis of the induction all triples of $P_{k-1}$ have been stored in the network. Consequently, all triples of $H$ are stored in the network.
– If $r$ is one of the rules (2)–(4), there will be an instantiation $\frac{R}{R'}$ such that $R \subseteq P_{k-1}$ and $H = P_{k-1} \cup R'$. According to the hypothesis of the induction all triples of $P_{k-1}$ and, therefore, all triples of $R$ were stored in the network according to our indexing scheme. Hence, if the rule is one of the rules (2)–(4), the instantiation of the rule will happen at the node where the triples of $R$ will meet. Again since triples in the antecedent of rules (2)–(4) always have a common element, these triples will meet at the node responsible for this common element. Then, triple $R'$ will be generated using local procedure Infer and will be stored in the network. Consequently, all triples of $H$ will be stored in the network.

□

## 4.4 Handling Node Failures

Until now, we have assumed that the network is stable for the duration of the reasoning process to have clear semantics for the results. However, node failures may pose extra difficulties for the forward chaining algorithm. While a comprehensive study of these issues is beyond the scope of this paper, we propose some simple steps that can increase the robustness of our method. These are based on standard techniques for monitoring the liveness of nodes such as timeouts and replication mechanisms.

In case nodes fail before or after the reasoning process takes place, we can exploit the storage redundancy provided by the underlying DHT network. FC* is the most vulnerable algorithm to node failures. The reason behind this is that during RDFS reasoning almost all nodes in the network contribute to the generation of inferred triples. Therefore, the failure of even a single node may cause a complex situation. Such a situation can be handled with assigning timeouts to the messages that are sent during FC*.

A source node $n_s$ creates a StoreMsg message and assigns a timeout $t_s$ and a unique identifier $mid$ with it. When $n_s$ sends a StoreMsg message to a destination node $n_d$ for a specific key $k$, it waits an acknowledgement message ACK from $n_d$. If timeout $t_s$ has passed and $n_s$ has not received and ACK, it resends the message. To ensure that the reasoning process at the destination node has been completed successfully, $d$ sends an ACK only after it has finished its local reasoning process for key $k$. To prevent nodes processing duplicate messages, each message contains the identifier of the source node and the message unique identifier $mid$. Certainly, choosing the correct value for the timeout will affect the performance of the system; setting a small timeout value may lead to a big number of duplicate messages while a node is still alive, while setting a big timeout value leads to delays in detecting node failures.

## 4.5 Redundant Triple Generation in Distributed Forward Chaining algorithms

Several authors have pointed out recently the generation of many redundant triples in forward chaining algorithms for RDFS reasoning, and especially in distributed ones [56,72,75]. The drawback of duplicate triple generation can be greater and more harmful in a distributed system. Duplicate triples are generated at different nodes and sent through the network causing an enormous amount of traffic as well as unnecessary load to the nodes. [56] treats the reasoning process at each node as a black box and does not elaborate on the rule set used. However, the authors do observe a big rate of duplicate triples and offer an elimination process to remove the duplicate triples that have been generated. In [75], the authors present results showing that as the number of processes grows the number of duplicate triples also increases and they especially refer to rule $rdfs9$ (i.e., rule (3b) of Table 1) for generating duplicate triples. [72] also provides solutions in the MapReduce framework to avoid duplicate triples and eliminate them if necessary. However, none of these works elaborate on the cause of redundant triples.

An important source of redundant triples which, to the best of our knowledge, has not been discussed in the literature before, is the recursive rules of Table 1. When the rules of Table 1 are used as they are or, in our case, translated into Datalog in the obvious way, they give rise to bilinear recursion. *Bilinear rules* are rules which have two occurrences of a recursive idb relation in their body and hence contain double recursion. Rules with double recursion can produce a large number of duplicate triples even in a centralized environment [2]. Let us demonstrate this with two simple examples.

*Example 1* Assume that our initial graph $G$ contains three triples: $t_1 =$`(a, sc, b)`, $t_2 =$`(b, sc, c)`, $t_3 =$`(c, sc, d)`. These triples form a small RDFS class hierarchy. Using rule (3a) of Table 1, we infer the following triples:

$t_1, t_2 \Rightarrow t_4 =$`(a, sc, c)`
$t_2, t_3 \Rightarrow t_5 =$`(b, sc, d)`
$t_3, t_4 \Rightarrow t_6 =$`(a, sc, d)`
$t_1, t_5 \Rightarrow t_6 =$`(a, sc, d)`

Therefore, triple $t_6$ is inferred twice. In the following example, we show how the generation of duplicate triples can be prevented if we use the linear rules we presented in Table 2.

*Example 2* Assume that we have the same initial graph $G$ of Example 1, but now we use rules 1 and 4 of Table 2. From applying rule 1, we add 3 assertions with idb relation `newTriple`:

$t'_1 =$`newTriple(a, sc, b)`
$t'_2 =$`newTriple(b, sc, c)`
$t'_3 =$`newTriple(c, sc, d)`

Using rule 4 of Table 2, we infer the following triples:

$t_1, t'_2 \Rightarrow t'_4 =$`newTriple(a, sc, c)`
$t_2, t'_3 \Rightarrow t'_5 =$`newTriple(b, sc, d)`
$t_1, t'_5 \Rightarrow t'_6 =$`newTriple(a, sc, d)`

Notice that the pair of triples $t_3$ and $t'_4$ cannot satisfy any of the bodies of rules of Table 2 and hence triple $t'_6$ is inferred only once.

In our previous implementation of forward chaining, called FC in [35], we used bilinear rules and made no distinction between explicit and inferred triples in the algorithm. Every time a new triple $t$ was generated, it was sent to be stored at the responsible nodes without specifying that $t$ is an inferred triple. The nodes that received $t$ were producing

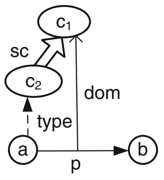


**Fig. 5** Redundancy in RDF(S) graphs

new triples from all triples stored locally without taking into consideration whether they were previously inferred or not. For instance, in the example of Fig. 3, if we had used the rules of our previous algorithm FC [35], node $n_1$ in Fig. 3(c) would have also inferred the same triples that node $n_2$ produced causing unnecessary traffic and more processing load to nodes $n_4$, $n_5$ and $n_6$.

Another source of triple generation redundancy is when RDF data is derived using different entailment rules. In this case, techniques such as the above cannot prevent the generation of redundant triples. For example, `dom` and `range` statements can interact with other statements in many ways to produce redundant triples. The following example shows one such case where a triple generation redundancy may occur.

*Example 3* In graph $G$ of Fig. 5, triple (`a, type,` $c_1$) can be inferred from triples (`a, p, b`), (`p, dom,` $c_1$) using rules 1 and 6 of Table 2. The same triple can also be derived from triples (`a, type,` $c_2$), ($c_2$, `sc,` $c_1$) using rules 1 and 5.

In our setting, such a triple generation occurs at different nodes which leads to flooding the network with redundant information and decreasing the system's performance. Identifying all such cases and dealing with them in our system is part of our future work.

## 5 Distributed Backward Chaining

In this section, we describe how a backward chaining algorithm can be implemented in the distributed environment of a DHT. In contrast to the data driven nature of forward chaining, backward chaining (BC) starts from the given query and to finds rules that are used to derive answers to the query.

### 5.1 Adorned Datalog Rules

Given that we are using a Datalog version of the $\rho df$ inference rules, the challenge here is to construct an algorithm that can process recursive Datalog rules in a distributed environment such as DHTs. To achieve this, t is helpful to transform the Datalog rules into a set of *adorned rules* that indicate which variables are bound and which are free. This is useful for finding a good order in which the predicates of the rule bodies should be evaluated. First, we rewrite the Datalog rules of Table 2 as shown in Table 3. This will allow us to compare BC with the entailment algorithm presented in [51] (see Sect. 5.2 below). In addition, we use this set of rules for the magic sets transformation technique described later since it satisfies the unique binding property [71] which is an essential prerequisite for the magic sets technique (see Sect. 6 for details). Using the rules of Table 2 would also be possible in BC without changing the algorithm.

In Table 3, we use the edb relation `triple` and the idb relations `subClass`, `subProperty`, `type` and `newTriple`. Edb relation `triple` denotes all triples that are locally stored, while the idb relations denote the various kinds of triples that can be inferred. The difference with Table 2 is that now `newTriple` is not the only idb relation used to store inferred triples. When the property of a triple or triple pattern is equal to `sp` then relation `subProperty` is used, when the property is equal to `sc` then relation `subClass` is used, and when the property is equal to `type` then relation `type` is used. In any other case, the relation `newTriple` is used. Rules 1, 3, 5 and 7 assign the triples found locally to the idb relations `subProperty`, `newTriple`, `subClass` and `type`, respectively, depending on the property of the triple. These rules together are equivalent to rule 1 of Table 2. Rules 2, 4 ,6, 8, 9, and 10 of Table 3 are equivalent with rules 2, 3, 4, 5, 6 and 7 of Table 2, respectively.

We extend the concept of rule adornment from Datalog query processing [71] to exploit the distributed philosophy of DHTs. As already mentioned, 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. We compute this key by choosing a constant part of the triple pattern in the order subject, object, property. Therefore, the corresponding predicate of the triple pattern has an argument that is not only bound, but also the *key* that led to the responsible node.

**Definition 6** 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 a bound argument which is also a *key*, $b$ indicates a bound argument which is not the *key*, and $f$ a free argument.

Following this definition, an adorned predicate $p^a$ indicates which argument of $p$ is the key, which ones are bound and which are free. Table 4 shows all possible adornments of the rules presented in Table 3 based on the indexing scheme of a triple pattern explained before.

Each adorned rule of Table 4 covers one possibility of using the corresponding rule of Table 3 in our backward chaining algorithm by additionally encoding run-time information about key, bound and free arguments. All possibilities that can actually take place are covered in Table 4. Since we do not allow queries where the triple pattern does not contain any constant values, note that there is no adorned predicate `newTriple`$^{fff}$. In addition, based on the order of the triple pattern's terms which we choose to use as an

**Algorithm 2**: BC: Backward chaining algorithm

```
event $n$.QUERYREQ ($key, tp, rid$) from $m$
    Let $p^a$ be the adorned predicate of $tp$;
    $R$=BCRDFS($p^a, rid$);
    sendto $m$.GetResp($R$);
end event

event $n$.BCRDFSReq ($p^a$) from $m$
    $R$=BCRDFS($p^a$);
    sendto $m$.BCRDFSResp($R$)
end event

procedure $n$.BCRDFS ($p^a, rid$)
    if $rid \| p^a \in processedRequests$ then
        return {};
    $processedRequests$.add($rid \| p^a$);
    $adornedRules$ =APPLYRULE($p^a$);
    $R = \{\}$;
    forall rules in $adornedRules$ do
        $r \leftarrow$ REMOVEFIRST($adornedRules$);
        if $r.body$ has one predicate then $R$=MATCHPREDICATE ($p^a$);
        else
            Let $q_1$ be the adorned predicate of $r.body$ with a $k$ element in its
            adornment and $q_2$ the other predicate of $r.body$;
            if $q_1$ is the edb triple then $R'$ =MATCHPREDICATE ($q_1$);
            else
                $R'$=BCRDFS($q_1, rid$);
            end
            if $R' = \{\}$ then return $R$;
            if $R' = \emptyset$ then return $R$;
            foreach value $v_i$ of the common variable $Z$ in $R'$ do
                $id_i$ =HASH ($v_i$);
                rewrite $p_2$ to $p'_2$;
                sendto $id_i$.BCRDFSReq($p'_2$)
                receive BCRDFSResp($R_i$) from $id_i$
                $R = R \cup R_i$;
            end
        end
    end
    return $R$;
end procedure
```

identifier (i.e., subject, object, property), not all combinations of the adornments are required. For example, the adorned predicate `subProperty`$^{bk}$ `(X, Y)` will never appear in our algorithm, since such a request would require to use as a key the object of the corresponding triple pattern although the subject is also bound.

## 5.2 Algorithm Description

Let us now describe our backward chaining algorithm BC which is shown in pseudocode in Algorithm 2. The node that wants to pose a query composes a QUERYREQ message and sends it to the network as described in Sect. 3.4. Suppose that a QUERYREQ request with unique identifier $rid$ arrives at node $n$ which is responsible for a constant $key$ included in the triple pattern $tp$. Node $n$ firstly transforms the triple pattern $tp$ to an adorned predicate $p^a$. property is a variable, then four

Then, $n$ calls local procedure BCRDFS which takes as an input the adorned predicate $p^a$ and the request identifier $rid$ and outputs a relation $R$ which contains the tuples of the bindings of the free arguments (i.e., the variables) of the predicate. These tuples of bindings form the answer to the query.

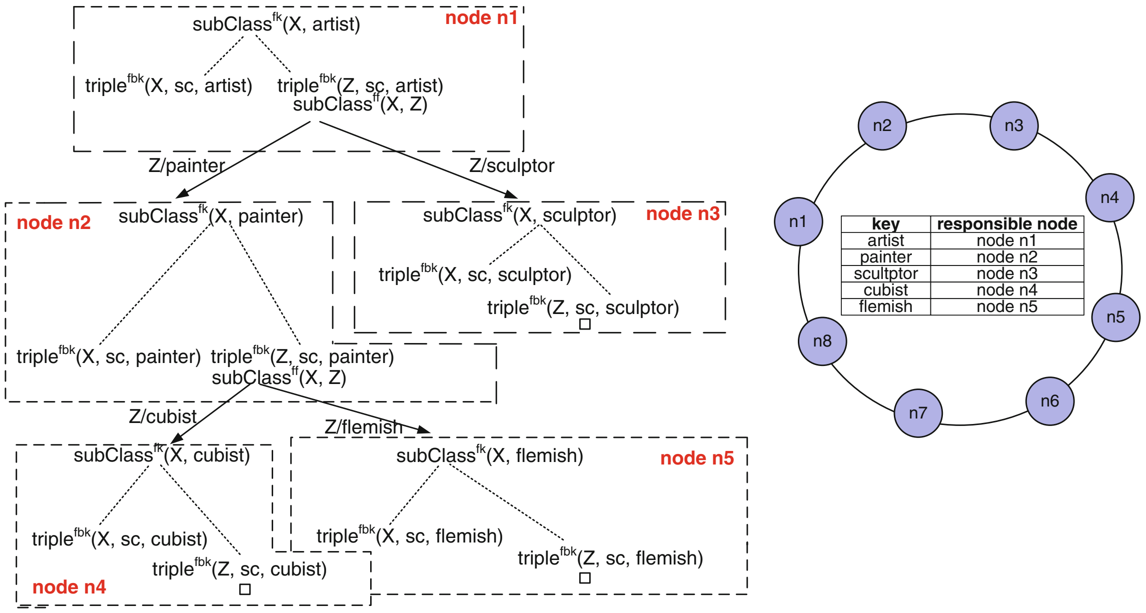
When BCRDFS is called, first the node ensures that the algorithm will not process the same request and hence avoid going into a infinite loop. Symbol $\|$ denotes string concatenation. Then, the input predicate $p^a$ is checked against the head of the rules of Table 4 using local function APPLYRULE. Rules that can be applied to the predicate are added to the list

**Table 3** $\rho df$ inference rules in Datalog (2nd version)

| Rule | Head | Body |
|---|---|---|
| 1 | `subProperty(X, Y)` | `triple(X, sp, Y)` |
| 2 | `subProperty(X, Y)` | `triple(Z, sp, Y), subProperty(X, Z)` |
| 3 | `newTriple(X, P, Y)` | `triple(X, P, Y)` |
| 4 | `newTriple(X, P, Y)` | `triple(X, P1, Y), subProperty(P1, P)` |
| 5 | `subClass(X, Y)` | `triple(X, sc, Y)` |
| 6 | `subClass(X, Y)` | `triple(Z, sc, Y), subClass(X, Z)` |
| 7 | `type(X, Y)` | `triple(X, type, Y)` |
| 8 | `type(X, Y)` | `type(X, Z), triple(Z, sc, Y)` |
| 9 | `type(X, Y)` | `newTriple(X, P, Z), triple(P, dom, Y)` |
| 10 | `type(X, Y)` | `newTriple(Z, P, X), triple(P, range, Y)` |

**Table 4** Adorned $\rho df$ inference rules

| Rule | Head | Body |
|---|---|---|
| 1a | subProperty$^{kf}$ (X, Y) | triple$^{kbf}$(X, sp, Y) |
| 1b | subProperty$^{fk}$(X, Y) | triple$^{fbk}$(X, sp, Y) |
| 1c | subProperty$^{kb}$(X, Y) | triple$^{kbb}$(X, sp, Y) |
| 1d | subProperty$^{ff}$(X, Y) | triple$^{fbf}$(X, sp, Y) |
| 2a | subProperty$^{kf}$(X, Y) | triple$^{kbf}$(X, sp, Z),subProperty$^{ff}$(Z, Y) |
| 2b | subProperty$^{fk}$(X, Y) | subProperty$^{ff}$(X, Z),triple$^{fbk}$(Z, sp, Y) |
| 2c | subProperty$^{kb}$(X, Y) | triple$^{kbf}$(X, sp, Z),subProperty$^{bf}$(Z, Y) |
| 2d | subProperty$^{ff}$(X, Y) | subProperty$^{ff}$(X, Z),triple$^{fbf}$(Z, sp, Y) |
| 3a | newTriple$^{kff}$(X, P, Y) | triple$^{kff}$(X, P, Y) |
| 3b | newTriple$^{kbf}$(X, P, Y) | triple$^{kbf}$(X, P, Y) |
| 3c | newTriple$^{kbb}$(X, P, Y) | triple$^{kbb}$(X, P, Y) |
| 3d | newTriple$^{ffk}$(X, P, Y) | triple$^{ffk}$(X, P, Y) |
| 3e | newTriple$^{fbk}$(X, P, Y) | triple$^{fbk}$(X, P, Y) |
| 3f | newTriple$^{fkf}$(X, P, Y) | triple$^{fkf}$(X, P, Y) |
| 4a | newTriple$^{kff}$(X, P, Y) | triple$^{kff}$(X, P1, Y),subProperty$^{ff}$(P1, P) |
| 4b | newTriple$^{kbf}$(X, P, Y) | triple$^{kff}$(X, P1, Y),subProperty$^{fb}$(P1, P) |
| 4c | newTriple$^{kbb}$(X, P, Y) | triple$^{kfb}$(X, P1, Y),subProperty$^{fb}$(P1, P) |
| 4d | newTriple$^{ffk}$(X, P, Y) | triple$^{ffk}$(X, P1, Y),subProperty$^{ff}$(P1, P) |
| 4e | newTriple$^{fbk}$(X, P, Y) | triple$^{ffk}$(X, P1, Y),subProperty$^{fb}$(P1, P) |
| 4f | newTriple$^{fkf}$(X, P, Y) | newTriple$^{fff}$(X, P1, Y),triple$^{fbk}$(P1, sp, P) |
| 5a | subClass$^{kf}$(X, Y) | triple$^{kbf}$(X, sc, Y) |
| 5b | subClass$^{fk}$(X, Y) | triple$^{fbk}$(X, sc, Y) |
| 5c | subClass$^{kb}$(X, Y) | triple$^{kbb}$(X, sc, Y) |
| 5d | subClass$^{ff}$(X, Y) | triple$^{fbf}$(X, sc, Y) |
| 6a | subClass$^{kf}$(X, Y) | triple$^{kbf}$(X, sc, Z),subClass$^{ff}$(Z, Y) |
| 6b | subClass$^{fk}$(X, Y) | subClass$^{ff}$(X, Z),triple$^{fbk}$(Z, sc, Y) |
| 6c | subClass$^{kb}$(X, Y) | triple$^{kbf}$(X, sc, Z),subClass$^{bf}$(Z, Y) |
| 6d | subClass$^{ff}$(X, Y) | subClass$^{ff}$(X, Z),triple$^{fbf}$(Z, sc, Y) |
| 7a | type$^{kf}$(X, Y) | triple$^{kbf}$(X, type, Y) |
| 7b | type$^{fk}$(X, Y) | triple$^{fbk}$(X, type, Y) |
| 7c | type$^{kb}$(X, Y) | triple$^{kbb}$(X, type, Y) |
| 7d | type$^{ff}$(X, Y) | triple$^{fbf}$(X, type, Y) |
| 8a | type$^{kf}$(X, Y) | triple$^{kbf}$(X, rdf:type, Z),subClass$^{ff}$(Z, Y) |
| 8b | type$^{fk}$(X, Y) | type$^{ff}$(X, Z),triple$^{fbk}$(Z, subClassOf, Y) |
| 8c | type$^{kb}$(X, Y) | triple$^{kbf}$(X, rdf:type, Z),subClass$^{fb}$(Z, Y) |
| 8d | type$^{ff}$(X, Y) | type$^{ff}$(X, Z),triple$^{fbf}$(Z, sc, Y) |
| 9a | type$^{kf}$(X, Y) | newTriple$^{kff}$(X, P, Z),triple$^{fbf}$(P, dom, Y) |
| 9b | type$^{fk}$(X, Y) | newTriple$^{fff}$(X, P, Z),triple$^{fbk}$(P, dom, Y) |
| 9c | type$^{kb}$(X, Y) | newTriple$^{kff}$(X, P, Z),triple$^{fbb}$(P, dom, Y) |
| 9d | type$^{ff}$(X, Y) | newTriple$^{fff}$(X, P, Z),triple$^{fbf}$(P, dom, Y) |
| 10a | type$^{kf}$(X, Y) | newTriple$^{ffk}$(Z, P, X),triple$^{fbf}$(P, range, Y) |
| 10b | type$^{fk}$(X, Y) | newTriple$^{fff}$(Z, P, X),triple$^{fbk}$(P, range, Y) |
| 10c | type$^{kb}$(X, Y) | newtriple$^{ffk}$(Z, P, X),triple$^{fbb}$(P, range, Y) |
| 10d | type$^{ff}$(X, Y) | newTriple$^{fff}$(Z, P, X),triple$^{fbf}$(P, range, Y) |


node n1
subClass^fk(X, artist)
triple^fbk(X, sc, artist)
triple^fbk(Z, sc, artist)
subClass^ff(X, Z)
Z/painter
Z/sculptor
node n2
subClass^fk(X, painter)
triple^fbk(X, sc, painter)
triple^fbk(Z, sc, painter)
subClass^ff(X, Z)
node n3
subClass^fk(X, sculptor)
triple^fbk(X, sc, sculptor)
triple^fbk(Z, sc, sculptor)
Z/cubist
Z/flemish
subClass^fk(X, cubist)
triple^fbk(X, sc, cubist)
triple^fbk(Z, sc, cubist)
node n4
subClass^fk(X, flemish)
node n5
triple^fbk(X, sc, flemish)
triple^fbk(Z, sc, flemish)
n1
n2
n3
n4
n5
n6
n7
n8

| key | responsible node |
|---|---|
| artist | node n1 |
| painter | node n2 |
| scultptor | node n3 |
| cubist | node n4 |
| flemish | node n5 |

**Fig. 6** Distributed evaluation of rules for backward chaining

*adornedRules*. Each rule can have one or two predicates in its body. Rules that have one predicate in their body (i.e., rules 1, 3, 5, 7) can always be evaluated locally since this predicate is always the edb relation `triple` and one of its bound arguments is the key that led to this node. 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 is one such predicate always since one of the arguments of the head predicate is the key that led to the specific node. Therefore, there will be a body predicate (let us call it $q_1$) which has an adornment containing the letter $k$ (this is always possible as seen in Table 4) and can be processed locally. If predicate $q_1$ is the edb relation `triple`, it is checked against the local database to find matching triples using local function MatchPredicate. The variable bindings are returned in relation $R'$. In case predicate $q_1$ is an idb relation, then procedure `BCRDFS` is called recursively with input the new adorned predicate. By evaluating one predicate locally, we have values that can be passed to the other predicate which is sent to be evaluated remotely at different nodes. Notice in Table 4 that all rule bodies with two predicates have a single common variable, let it be $Z$. 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 $q_2$ to a new predicate $q_2'$ 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 `BCRDFSReq` message to the node responsible for the hashed value of key $v_i$. This 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 `BCRDFSResp`($R_i$), it adds the tuples of each $R_i$ to relation $R$ and returns $R$. In case of a Boolean query, i.e., a query without any variables, relation $R$ actually holds a true or false answer. In this case, the union operator of line 2 is actually an OR operator meaning that procedure `BCRDFS` requires at least one answer to be true in order to return true. Otherwise, it returns false.

This procedure is recursive in two ways. Recursion appears locally at a node when predicate $q_1$ is an idb predicate and among the nodes participating in the query evaluation when a new `BCRDFSReq` message is sent. The procedure terminates when the node that received the initial query has collected a response message for each request it has sent. A recursion path ends when the predicate which is evaluated first 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 6–7) so that an infinite loop is avoided.

Let us now show how BC works using the example RDF(S) hierarchy of Fig. 2. Suppose that all triples of the RDF(S) hierarchy are stored in the network and a user poses the query "Find all the subclasses of class artist" to a node in the network. Figure 6 shows how the rules will be distributed in the various nodes of the network. The query is expressed as the triple pattern `(X, sc, artist)` and is routed to node $n_1$, which is responsible for key `artist`.

Node $n_1$ transforms the triple pattern to the adorned predicate $\texttt{subClass}^{fk}$`(X, artist)` which matches the head of rules 5b and 6b of Table 4. Using rule 5b, node $n_1$ finds the local matches of the query, i.e., it retrieves all the immediate subclasses of class `artist` (`painter` and `sculptor`). Using rule 6b, it finds matches locally for the predicate `triple` and rewrites the second predicate `subClass` to other adorned predicates that are sent to other nodes to be evaluated. More specifically, node $n_1$ sends messages to nodes $n_2$ and $n_3$ to retrieve the subclasses of classes `painter` and `sculptor`, respectively. Node $n_3$ finds no matches for class `sculptor` and returns an empty result to node $n_1$. Node $n_2$ finds locally classes `cubist` and `flemish` using rule 5b and sends two other requests to nodes $n_4$ and $n_5$ using rule 6b. Nodes $n_4$, $n_5$ find no matches and return an empty result to node $n_2$. Node $n_2$ that collects the answers of nodes $n_4$ and $n_5$ adds to them the answers found locally and returns the answer to node $n_1$. Finally, as soon as node $n_1$ has the answers of both nodes $n_2, n_3$, it composes the final answer by adding its local answers with the answers of these nodes and returns the result set to the node that posed the query.

*Comparison with the entailment algorithm of* [51]. It is interesting to examine how our backward chaining algorithm is related with the entailment algorithm proposed in [51]. The entailment algorithm of [51] checks whether a triple $t$=`(a,p,b)` is entailed from a graph $G$. Comparing our algorithm with the entailment algorithm of [51], we observe certain similarities. The algorithm of [51] takes certain actions depending on the property `p` of the triple. Similarly, we trigger a rule depending on the property of the triple. The first point of the algorithm deals with the case where `p` is equal to `dom` or `range`. In this case, it just checks if $t$ exists in $G$. Similarly, BC would trigger rules 3c and 4c from Table 4 and since no triple will be found to satisfy the first predicate of rule 4c, rule 3c will actually retrieve triples locally stored and, therefore, triples that are in $G$. The second step of the algorithm deals with triples that have as property the `sp` value. In this case, the algorithm checks if there exists a path from `a` to `b` in the graph through `sp` links. Similarly, BC triggers rules 1c and 2c which traverse the transitive closure of the graph consisting of `subProperty` relations. The same holds if `p` equals to `sc`. The 4th step of the algorithm, where property `p` does not belong to the $\rho df$ vocabulary, is actually covered in our case with rules 3c and 4c. The difference is that the algorithm of [51] builds a graph to check if there is a path from a vertex marked with `(a,b)` that reaches `p` in a bottom-up fashion. On the contrary, BC works in a top-down fashion which searches if there is another property `p'` in a triple `(a,p',b)` and checks if there is a path from `p'` to `p` in the transitive closure of the subproperty relation. Finally, the algorithm deals with the case where the property of $t$ is equal to `type`. The difference here is that the algorithm of [51] needs to preprocess the whole graph $G$ to mark certain nodes that are relevant to the triple in question. This case is covered in BC by rules 7c, 8c, 9c, 10c, 11c, 12c of Table 4 and traverses only the part of graph that is relevant to the query in a top-down manner.

## 5.3 Termination, Soundness and Completeness

In this section, we formally prove that BC terminates and is sound and complete.

**Theorem 5** (Termination) *Algorithm BC terminates.*

*Proof* BC is a recursive algorithm in two ways. First, recursion occurs for a query $q$ when one of the predicates of the body of a rule is an idb relation. If the predicate evaluated first is the edb relation `triple`, the second predicate to be evaluated is an idb relation and recursion occurs by sending a BCRDFSREQ message to another node (line 24 of BC). A recursion path starts from the query $q$ and traverses the RDF(S) graph in a top-down fashion until there is no other node to follow. In this case, the recursion path terminates since no tuples of bindings are found at a node for the first predicate and, therefore, there are no values to pass to the second predicate. In case, the RDF(S) graph contains cycles an infinite loop can occur. To avoid infinite loops caused by graphs with cycles, the unique query request identifier together with the adorned predicate is inserted in a list $processedRequests$. Each time procedure BCRDFS is called, the list of all processed requests is checked; if a request has been processed already, BC terminates by returning an empty set of tuples (lines 7–8 of BC).

Second, if the predicate that should be evaluated first (predicate with $k$ in its adornment) is an idb relation, then we have local recursion at one node (line 18 of BC). This recursion can happen only once since the only rules which contain an adorned predicate with $k$ in its adornment which is not the edb relation `triple` are rules 9a, 9c, 10a, and 10c of Table 4. In this case, local procedure BCRDFS is called again, but this time the first predicate to be evaluated is the edb relation `triple` which results in a recursion of the previous case. □

In the following, we prove that algorithm BC is sound and complete. By *sound* we mean the following. Let $G$ be a graph stored in the network and $q$ be a query answered by BC. Let $R$ be the relation-answer to $q$ which has as attributes the variables of $q$ and as tuples the tuples of bindings for these variables. If $H$ is a set of triples obtained from $q$ by replacing the variables of $q$ by all the corresponding values in $R$, then $G \models H$. By *complete* we mean that if $G$ is the graph stored in the network, $H$ any graph and $G \models H$, then for each triple $t$ in $H$, BC will return true for the query $q = t$. We make the same assumptions as in FC* regarding the stability of the network.

We define the concept of a proof tree for BC, which models the execution of BC on top of a DHT, as follows.

**Definition 7** A *proof tree* is a finite tree with the following properties:

(1) Every node is of the form $a_i : (q_i, R_i, LR_i, RR_i)$, where $a_i$ is a DHT node, $q_i$ is the query that the node should evaluate, $R_i$, $LR_i$ are relations with tuples of bindings for the variables of query $q_i$ and $RR_i = \{RR_i^1, RR_i^2, \ldots, RR_i^{k_i}\}$ is a set of relations with tuples of bindings for the variables of query $q_i$. Relations $R_i$ and $LR_i$ as well as the set $RR_i$ can be empty.

(2) For every node $a_i : (q_i, R_i, LR_i, RR_i)$, we have the following:
- $q_i$ is the query (triple pattern) with a key for which the node is responsible.
- $LR_i$ is a relation that contains the tuples of bindings for $q_i$ found from the local database of the node by applying one of the rules 1, 3, 5 or 7 depending on the property of $q_i$.
- $RR_i = \{RR_i^1, RR_i^2, \ldots, RR_i^{k_i}\}$ is a set of relations that node $a_i$ received from its $k_i$ children. If node $a_i$ does not have any children, then the set $RR_i$ is empty.
- If $RR_i$ is non-empty then $R_i = LR_i \cup \bigcup_{j=1}^{k} RR_i^j$. Otherwise, $R_i = LR_i$.

(3) If node $a_i$ has $k_i$ children, then for each child $a_j : (q_j, R_j, LR_j, RR_j = \{RR_j^1, RR_j^2, \ldots, RR_j^{k_j}\})$, we have that $R_j = RR_i^j$ $(1 \leq j \leq k_i)$.

Such a proof tree offers us a nice representation of the computation of BC. Figure 7 shows the proof tree as defined above for the example shown in Fig. 6. The root of the tree corresponds to the activation of BC when a message QueryReq arrives at a DHT node. The nodes of the tree at increasing depths also show the communication between the nodes as BC proceeds. Each edge in the proof tree shows that a message BCRDFSReq was sent from the parent node to the child and one BCRDFSResp message was sent from the child to its parent node.

**Theorem 6** (Soundness) *Let q be a query and R the relation produced by BC which has as attributes the variables of q. If G is the set of triples stored in the network and H is the set of triples obtained from q by replacing its variables by all their values in the relation R, then $G \vdash_{mrdf} H$ and $G \models H$.*

*Proof* The proof is by induction on the depth $d$ of the proof tree of BC.

Base case: $d = 1$ (the root is $d = 0$). In this case, BC starts at the DHT node $n_1$ represented by the root of the

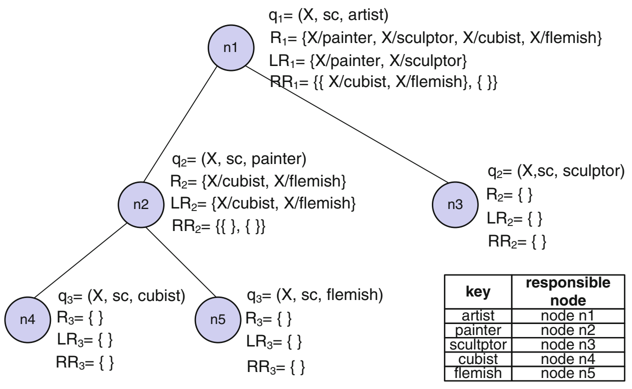


**Fig. 7** Proof tree for backward chaining

tree and transforms the query $q$ to an idb relation. If the predicate of $q$ is equal to `sp`, then the corresponding predicate is `subProperty`, and rules 1 and 2 of Table 4 are applied, depending on which arguments of the query are bound or not. Node $n_1$ finds locally $l$ tuples of bindings for the variables of $q$ by applying rule 1, which has a single predicate in its body i.e., the edb predicate `triple`, and assigns them to relation $LR$. Then, the node applies rule 2, which has two predicates in its body. The predicate that is evaluated first is the one that has $k$ in its adornment (predicate `triple` in this case). Since the first predicate is the edb predicate `triple`, $k$ values are found locally and are passed to the second predicate (idb predicate `subProperty`) forming $k$ new queries that will be evaluated at $k$ different nodes. Therefore, node $n_1$ sends $k$ BCRDFSReq messages to its $k$ nodes (children nodes in the proof tree) for evaluating a query with predicate `sp`. BC runs again at these $k$ nodes and finds local results from rule 1. Since the depth of the tree is 1, rule 2 is not satisfied and no other messages are sent. Node $n_1$ receives $k$ BCRDFSResp response messages from the $k$ nodes, each one containing a relation $R^j$ (for $1 \leq j \leq k$) with the data the $k$ nodes found locally. Then, $n_1$ composes the answer to query $q$ by assigning to relation $R$ the union of the tuples of all relations (i.e., $R = LR \cup \bigcup_{j=1}^{k} R^j$).

The sequence of graphs ($mrdf$ proof according to Definition 4) that shows $G \vdash_{mrdf} H$ can easily be constructed. The first element of the sequence is $G$ followed by $k$ subsequences representing the sets of triples obtained from replacing the variables of $q$ by their values in relations $R^j$ for $1 \leq j \leq k$ (the order of the subsequences does not matter). Each of the $k$ sets of triples can be represented by as many proof steps as the number of tuples of bindings contained in $R^j$. Each step is the result of the application of rule (2a) of the $\rho df$ rules of Table 1 to the set of triples produced in the immediately previous proof step. Then, the sequence of graphs continues with $l$ subsequences representing the triples obtained from replacing the variables of $q$ by their values in relation $LR$. This can be done by applying rule (1b) of Table 1.

Similarly, if the property of the query is equal to `sc`, then rules 5 and 6 of Table 4 will be fired. To compose a sequence of graphs, $\rho df$ rules (3a) and (1b) of Table 1 will be used. If the property of the query is anything but `sp`, `sc` and `type`, rules 2, 3 and 4 of Table 4 will be fired and $\rho df$ rules (2a), (2b) and (1b) would be used to construct a sequence of graphs. Finally, if the property of the query is equal to `type`, then rules 7, 8, 9 and 10 of Table 4 will be fired at the DHT nodes. In case of rules 9 and 10, the idb predicate `newTriple` is the first predicate that will be evaluated. In this case, rules 2, 3 and 4 will also be fired at node $n_1$. Messages are now sent to other nodes for the evaluation of the first predicate, while the values returned are passed to the second predicate which is the edb relation `triple` and will be evaluated locally. A $mrdf$ proof for a query with property equal to `type` would then be constructed by rules (3b), (4), (2b) and (1b) of Table 1 together with rules (3a) and (2a) for the latter case.

*Inductive step* We assume the theorem holds for the execution of BC with a proof tree of depth $d$. We will show the result for a proof tree $T$ of depth $d + 1$.

We take the proof tree $T$ of depth $d + 1$ and prune the root node. The resulting $k$ subtrees $T_j$ $(1 \leq j \leq k)$ have depth $d$ and the theorem holds for the queries $q_j$ of their root nodes. Hence, if $H'$ is the union of the sets of triples obtained by replacing the variables of queries $q_j$ by their value in the relations $R_j$, then $G \vdash_{mrdf} H'$. Now we can continue the proof of $H'$ from $G$ by adding the steps corresponding to the root node of tree $T$ to arrive at graph $H$. This is done as in the base case.

Since we have proved $G \vdash_{mrdf} H$, Theorem 1 of Sect. 2 gives us that $G \models H$ as well. □

**Theorem 7** (Completeness) *Let $G$, $H$ be $mrdf$ graphs with $H$ not containing triples of the form $(x, sp, x)$ nor $(x, sc, x)$ for $x \in U \cup L$. Let $G$ be the graph stored in the network. If $G \models H$, then for each triple $t \in H$, BC will return true to the query $q = t$.*

*Proof* Using Theorem 1 and the fact that $G \models H$, we have that $G \vdash_{mrdf} H$. If $G \vdash_{mrdf} H$, we will show that for each triple $t \in H$, BC will return true to query $q = t$. We will prove this using induction on $k$, the number of $mrdf$ proof steps required for $H$ to be derived from $G$.

For a Boolean query $q$, let it be $q = t =$`(a,p,c)`, the subject `a` is used as the identifier of the triple pattern and algorithm BC is instantiated at node $n_1$ which is responsible for key `a`. As soon as node $n_1$ receives a QUERYREQ message, it transforms the triple pattern ($t$ in this case) to an adorned predicate $p^a$ depending on the property of the triple pattern $t$ (i.e., $t$ is transformed to one of the adorned predicates `subProperty`$^{kb}$, `new Triple`$^{kbb}$, `subClass`$^{kb}$, `type`$^{kb}$).

Base case ($k = 1$): All triples of $H$ are derived from $G$ in 1 proof step by the application of a single rule $r$ of the minimal deductive system of Table 1.

– If $r$ is rule (1b) then $H \subseteq G$. In this case, all triples of H are stored in the network according to our indexing scheme. At node $n_1$, BC will fire one of the rules 1c, 3c, 5c, 7c of Table 4 depending on the property of $q$. These rules have one predicate in their body which is the edb predicate `triple`. Local function MATCHPREDICATE will retrieve matching triples from the node's local database. According to the indexing scheme, the triple `(a,p,c)` will be located at node's $n_1$ local database and the answer for query $q$ will be true.
– If $r$ is rule (2a), then there is an instantiation $\frac{R}{R'}$ of $r$ such that $R \subseteq G$ and $H = G \cup R'$ and $t = R' =$`(a, sp, c)`. Let $t \in H$. $R$ consists of two triples, $t_1 =$`(a, sp, b)` and $t_2 =$`(b, sp, c)`. Since this is the first $mrdf$ proof step, $t_1, t_2 \in G$. The property of the triple $t$ is equal to `sp` and hence the adorned predicate $p^a$ at node $n_1$ is equal to `subProperty`$^{kb}$`(a, c)` and will be matched with the head of rules 1c and 2c of Table 4. If $t \in G$, then rule 1c will match with $t$ and BC will return true. Otherwise, rule 2c will match locally predicate `triple(a, sp, Z)`. Since all triples of $G$ have been stored using the triple indexing scheme and triple $t_1$ shares the same subject with $t$, triple $t_1$ (which has subject `a`) will be located at node $n_1$. Variable $Z$ will be bound to the object of triple $t_1$, i.e., $Z$ will be bound to value `b`. According to the algorithm, for each value $v_i$ of the bindings, the second predicate of rule 2c is rewritten into a new one with a `subProperty` predicate without any variable. For the value `b` found from triple $t_1$, a new rewritten predicate will be `subProperty`$^{kb}$`(b, c)` and a BCRDFSREQ message is sent to node $n_2$ which is responsible for key `b`. Node $n_2$ that receives this message matches the predicate with rule 1c. Triple $t_2$ has as subject the same value with triple's $t_1$ object (i.e., `b`). Therefore, node $n_2$ retrieves locally triple $t_2 =$`(b, sp, c)` and returns true to node $n_1$. Then, node $n_1$ returns true for query $q$.
– If $r$ is one of the rules (2b),(3) or (4) then there is an instantiation $\frac{R}{R'}$ of $r$ such that $R \subseteq G$ and $H = G \cup R'$. The proof is similar with the one for rule (2a).

*Inductive step* We assume that the result holds for $H$ that can be proved from $G$ with number of proof steps less than $k$. We will prove that it holds for $H$ that can be proved from $G$ with number of proof steps equal to $k$.

So let us assume that $G \vdash_{mrdf} H$ in $k$ proof steps. Using the notation of Definition 3, $P_k = H$. Let $r$ be the rule of Table 1 that was used in the proof to go from $P_{k-1}$ to $P_k$.

– If $r$ is rule (1b), then $H \subseteq P_{k-1}$ and for each triple $t \in H$, $G \vdash_{mrdf} t$ in $k-1$ or less steps. Then, from the induction hypothesis, BC returns true to query $q = t$.
– If $r$ is rule (2a), there will be an instantiation $\frac{R}{R'}$ of $r$ such that $R \subseteq P_{k-1}$ and $H = P_{k-1} \cup R'$. Let $t \in H$. If $t \in P_{k-1}$, then $G \vdash_{mrdf} t$ in less than $k$ proof steps and, therefore, BC returns true for the query $q = t$ by the induction step. If $t \in R'$, then it is equal to `(a, sp, c)`. In BC, node $n_1$ matches the adorned predicate `subProperty`$^{kb}$`(a, c)` with the head of rules 1c and 2c of Table 4.
– If $r$ is one of the rules (2b),(3) or (4) then there is an instantiation $\frac{R}{R'}$ of $r$ such that $R \subseteq P_{k-1}$ and $H = P_{k-1} \cup R'$. The proof is similar with the one for rule (2a). □

### 5.4 Handling Node Failures

BC is more robust to node failures compared to FC*. The intuition behind this is that given a query only part of the network nodes are involved in the query proof tree. Thus, the probability that a node involved in the RDFS reasoning process will fail if lower than in FC*.

If the query requestor node fails, the query can be aborted anyway and the user can pose the query to another node. If a node fails that was involved in message routing (e.g., an intermediate node in a lookup message routing) but is not involved in the query execution proof tree, we employ whatever routing redundancy the underlying network provides. For example, Bamboo DHT employs a periodic recovery method and proposes a method for effective lookup timeout calculations which reduces lookup latency [59].

The case where one of the nodes involved in the query proof tree fails requires more attention. We distinguish the following cases based on the distributed proof tree:

– A child node fails before returning the results to its parent node. In this case, the parent node resends the `BCRDFSReq` message after a certain timeout has passed and no results were returned. Again, the timeout has to be carefully selected and should be dependent on the depth of the proof tree and the level the node belongs to.
– A parent node fails before its child node has sent its results to it. In this case, the child node does not receive an ACK and hence returns the results directly to the query requestor node or to a known ancestor node in the proof tree. Note that the IP addresses of the query requestor node and/or the ancestor nodes are piggybacked in the messages that are exchanged during the algorithm.

## 6 Forward Chaining for Magic Rules

Comparing a backward chaining algorithm with a forward chaining one is not always a fair comparison since forward chaining always computes all possible inferences while backward chaining focuses on a specific goal. In this section, we present a bottom-up method that benefits from the top-down technique of backward chaining. This method is well known from the database literature as *magic sets* transformation technique [8]. Such a technique is suitable for application scenarios where the query workload is known a priori and, therefore, only necessary triples related to the query workload are precomputed and stored to the network. In this way, storage is not overloaded by useless information, while query processing is done without any overhead.

The basic idea is that, given a specific type of query, rules are rewritten using information from the query so that a bottom-up evaluation is able to generate only the appropriate inferences. The benefit of using the new ruleset in the bottom-up evaluation is that it focuses only on data which is associated with the query and hence no unnecessary information is generated. In our case, we use the same ideas and rewrite the Datalog version of the $\rho df$ inference rules using the magic sets transformation. When the rewritten rules are executed in a forward chaining fashion, only triples that are related to the query are involved.

### 6.1 Magic Rules

Imagine that we need to answer the query "Find all instances of class `a`". This query is very often used and requires the most complicated RDFS inference as it involves *all rules*. In this section, we show how the magic sets algorithm can be implemented on top of a DHT for queries of the form `(X, type, a)` and we compare the bottom-up approach of this algorithm with the top-down approach of backward chaining. Note that because the query of the form `(X, type, a)` requires all rules to be answered, the set of rules we present in Table 5 is a superset of the rules required for queries of the form `(*, p, *)` where the subject and object of the triple pattern can be either a variable or a constant, `(X, sc, a)` and query `(X, sp, a)`. Although we rewrite the $\rho df$ inference rules using the magic sets transformation technique based on this type of query, the algorithm we present in the following can be applied for any type of query with the rules rewritten accordingly.

To transform a set of rules using the magic sets technique, we require the *unique binding property* [71]. This requirement means that each idb relation should appear with a unique adornment when a backward chaining algorithm is used. The rules of Table 2 do not satisfy the unique binding property for this type of query [71]. The idb relation `newTriple` appears with different adornments if we apply the rules using the

**Table 5** Magic rules

| Rule | Head | Body |
|---|---|---|
| 1 | `m_newTriple(P)` | $\texttt{sup}_{61}$`(P, Y)` |
| 2 | `m_newTriple(P)` | $\texttt{sup}_{71}$`(P, Y)` |
| 3 | `m_subProperty(Z)` | $\texttt{sup}_{21}$`(Z,Y)` |
| 4 | `m_subProperty(P)` | `m_newTriple(P)` |
| 5 | `m_subClass(Z)` | $\texttt{sup}_{41}$`(Z,Y)` |
| 6 | `m_type(Z)` | $\texttt{sup}_{51}$`(Z,Y)` |
| 7 | $\texttt{sup}_{21}$`(Z, Y)` | `m_subProperty(Y), triple(Z, sp, Y)` |
| 8 | $\texttt{sup}_{31}$`(P1, P)` | `m_newTriple(P), subProperty(P1, P)` |
| 9 | $\texttt{sup}_{41}$`(Z, Y)` | `m_subClass (Y), triple(Z, sc, Y)` |
| 10 | $\texttt{sup}_{51}$`(Z, Y)` | `m_type(Y), triple(Z, sc, Y)` |
| 11 | $\texttt{sup}_{61}$`(P, Y)` | `m_type(Y), triple(P, dom, Y)` |
| 12 | $\texttt{sup}_{71}$`(P, Y)` | `m_type(Y), triple(P, range, Y)` |
| 13 | `newTriple(X, P ,Y)` | `m_newTriple(P), triple(X, P, Y)` |
| 14 | `subProperty(X, Y)` | `m_subProperty(Y), triple(X, sp, Y)` |
| 15 | `subProperty(X, Y)` | $\texttt{sup}_{21}$`(Z,Y), subProperty(X, Z)` |
| 16 | `newTriple(X, P, Y)` | $\texttt{sup}_{31}$`(P1,P), triple(X,P1,Y)` |
| 17 | `subClass(X, Y)` | `m_subClass(Y), triple(X, sc, Y)` |
| 18 | `subClass(X, Y)` | $\texttt{sup}_{41}$`(Z,Y), subClass(X, Z)` |
| 19 | `type(X, Y)` | `m_type(Y), triple(X, type, Y)` |
| 20 | `type(X, Y)` | $\texttt{sup}_{51}$`(Z, Y), type(X, Z)` |
| 21 | `type(X, Y)` | $\texttt{sup}_{61}$`(P, Y), newTriple(X, P, Z)` |
| 22 | `type(X, Y)` | $\texttt{sup}_{71}$`(P, Y), newTriple(Z, P, X)` |
| 23 | `m_type(a)` | |

backward chaining algorithm. For example, consider the execution of query `newTriple(X, type, a)`. Using rule 5, the idb relation `newTriple` appears with bound the second and third argument, while using rules 6 or 7 and after passing values from the edb relation `triple`, it appears with bound only the second argument. Hence, the rules cannot be transformed into a set of magic rules, unless the predicate `newTriple` is split into two different predicates. However, this would introduce twice the number of rules imposing unnecessary overhead. For this reason, we prefer to use the $\rho df$ inference rules as we presented them in Table 3. We transform the above rules into a set of magic rules for query $q$ using the techniques of [8]. The rules are shown in Table 5.

For each idb predicate `p`, we create a magic predicate `m_p` which has as arguments the bound arguments of the unique binding appearing in `p` (i.e., the unique adornment). For each rule, we introduce a number of magic supplementary predicates associated with this rule. A supplementary predicate $\texttt{sup}_{ij}$ denotes that it is the supplementary predicate for rule $i$ and the predicate $j$ of the body of rule $i$ with $j = 0, \ldots, n$. For example, $\texttt{sup}_{21}$ denotes the supplementary predicate for predicate `subProperty` of the body of rule 2 in Table 3. Rules 1–6 define the magic predicates, rules 7–12 show the supplementary predicates and rules 13–22 show the initial rules modified to include the supplementary predicates. All these rules together with the fact `m_type(a)` (rule 23), which is used as the initialization rule, form the complete rule set after the magic set transformation. Note that the same set of rules depicted in Table 5 together with the fact `m_newTriple(p)` enables answering queries of the form `newTriple(*, p, *)`.

This set of rules ensures that only inferences related with class `a` will be generated. The role of magic predicates is that `m_p(v)` should be true if and only if in the top-down evaluation value `v` is passed as a binding to predicate `p`. Similarly, supplementary predicates represent the bound variables that have either been bound by the rule's head or by predicates evaluated at a previous step. Note here that we present an optimized version of the magic set transformation where zeroth supplementary predicates have been replaced by the magic predicates so that extra computations are avoided. For more information on the magic sets transformation, the interested reader might refer to [71].

## 6.2 Algorithm Description

Let us now describe how our new algorithm, which we call MS, works. Generally, MS works as the forward chaining algorithm presented in Sect. 4 with the only difference

**Algorithm 3**: MS: Magic sets algorithm

```
event $n$.MSREQ ($id$, $pred$) from $m$
    Magic($id$, $pred$);
end event

procedure $n$.Magic ($id$, $pred$)
    $localTriples$ = GETTRIPLESFROMDB ($pred.argument1$);
    INFER ($localTriples$, $pred$);
    $pairs = \{\}$;
    forall $t \in \{newTriples \setminus infTriples \setminus localTriples\}$ do
        if $t.property \in \rho df$ then
            $pairs.put(t.object, t)$;
        else
            $pairs.put(t.subject, t)$;
            $pairs.put(t.property, t)$;
            $pairs.put(t.object, t)$;
    forall $k' \in pairs.keys()$ do
        $id'$ =HASH ($k'$);
        $triples' = pairs.get(k')$;
        sendto $id'$.STOREMSG($id'$, $triples'$, $k'$, $true$);
        $infTriples$.add($triples'$);
    forall $pred' \in \{supPredicates \setminus infPreds\}$ do
        $id'$ =HASH ($pred'.argument1$);
        sendto $id$.MSREQ($id'$, $pred'$);
        $infPreds$.add($pred'$);
end procedure

event $n$.STOREMSG ($id$, $t$, $k$, $inf$) from $m$
    if $inf == true$ then Magic($id$, $pred$);
    INSERTTODB ($t$, $inf$);
end event
```

that more predicates are inferred from the rules. In MS, apart from the edb relation `triple` and the idb relations `subProperty`, `newTriple`, `subClass` and `type`, we also have the idb relations of the *magic* and the *supplementary* predicates. The relations of the supplementary predicates are also indexed in the network based on the values of their arguments so that appropriate values can be found locally at the corresponding nodes. Note that it is not necessary to index magic predicates since they can be reproduced by the supplementary predicates and rules 1–6 of Table 5.

Assume that a node needs to find the instances of class `a`. It sends a message containing the magic predicate `m_type(a)` to the node responsible for value `a` using the hash value of `a` as an identifier. The node that receives the magic predicate `m_type(a)` starts a bottom-up evaluation of the rules of Table 5 and sends the new inferred predicates to the network. Each time a node receives a new predicate, it computes the closure locally according to the rules of Table 5 and distributes the newly inferred facts in the network.

The pseudocode of the algorithm is presented in Algorithm 3. The node that receives the query transforms it to the corresponding magic predicate $pred$. Then, it creates an identifier $id$ from the only argument of the magic predicate $pred$ by hashing the value of the argument and sends a MSREQ($id$, $pred$) message to the network. When a node $n$ receives such a message, it calls local procedure `Magic`. This procedure works as FC* with two differences.

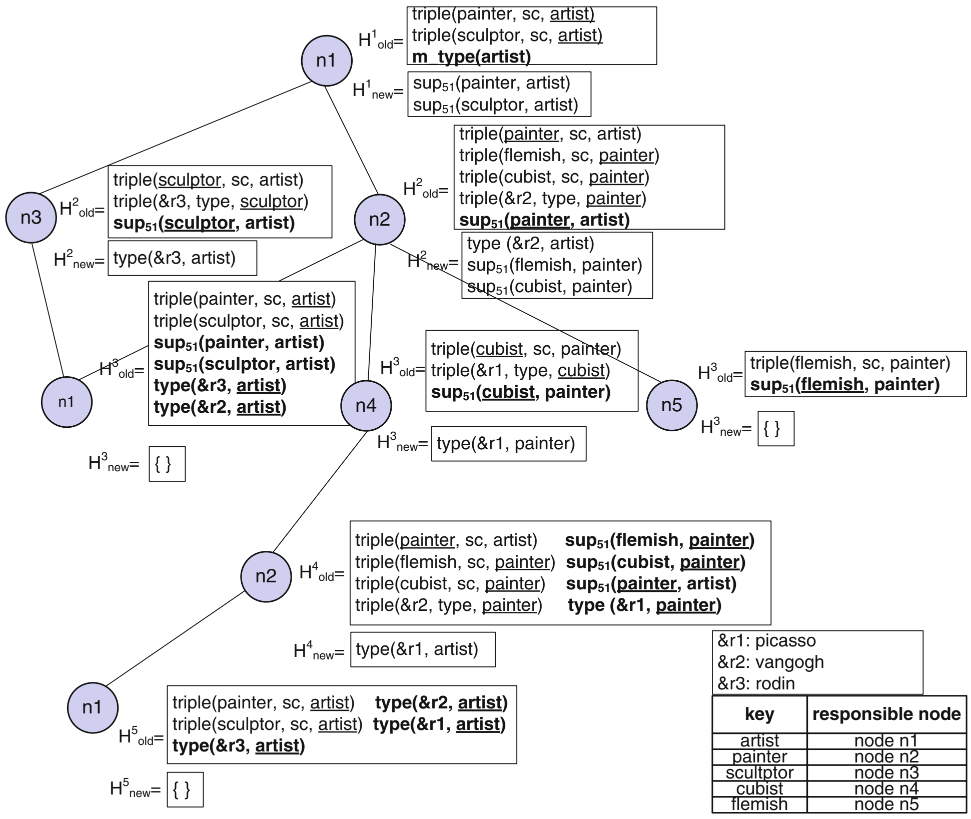
First, newly inferred triples may be indexed once in the network. For the idb predicates `subProperty`, `subClass` and `type`, only the object of the corresponding triple is used for indexing the triple. This indexing scheme ensures that triples will be sent to the appropriate nodes so that the reasoning process is complete. For the idb predicate `newTriple`, all three arguments are used as an identifier ensuring that all triple patterns of the form `(*, p, *)` can be answered after MS has taken place. The inferred triples are stored in the local database of the nodes they are indexed.

Second, inferred *supplementary* predicates are also sent in the network to the corresponding nodes. For each newly inferred supplementary predicate $pred'$, an identifier $id'$ is created by hashing the value of the first argument of $pred'$. Then, a message MSREQ($id'$, $pred'$) is sent to the responsible node in the network. We only use the first argument of $pred'$ since we are already at the node responsible for the value of the second argument and the required reasoning has been already completed. Procedure `Magic` is called when a node receives either a newly inferred triple or a new supplementary predicate value. The algorithm terminates when all nodes have reached a fixpoint and neither new triples nor new supplementary predicate values are generated.

The soundness and completeness of this algorithm follow from the equivalence of the original rules to the rewritten ones given the query [71] and the soundness and completeness results from FC*.

In Fig. 8, we demonstrate an example of the algorithm MS using the computation tree as defined in Sect. 4, where we have the edb and idb relations instead of triples. We use the RDFS example of Fig. 2 plus the triples `(picasso, type, cubist)`, `(vangogh, type, painter)` and `(rodin, type, sculptor)`. For readability reasons, in Fig. 8, we abbreviate the resources `picasso`, `vangogh` and `rodin` with `&r1`, `&r2` and `&r3`, respectively. We want to compute all instances of class `artist`. Node $n_1$ receives a MSREQ message with the predicate `m_type(artist)`. Using the magic set rules of Table 5, the supplementary predicates shown in $H^1_{\text{new}}$ of node $n_1$ are generated and sent to nodes $n_2$ and $n_3$ which are responsible for keys `painter` and `sculptor`, respectively. These nodes apply the magic sets rules using as input data the data appearing in $H^2_{\text{old}}$ and generate new values for the idb relations. The procedure continues until all nodes have reached a fixpoint (i.e., $H^i_{\text{new}}$ is empty). In this case, node $n_1$ has received all three instances of class `artist`. Then, the answer to the query can be found locally at node $n_1$.

**Fig. 8** Example for MS algorithm


n1
H1old= triple(painter, sc, artist) triple(sculptor, sc, artist) m_type(artist)
H1new= sup51(painter, artist) sup51(sculptor, artist)
n3
H2old= triple(sculptor, sc, artist) triple(&r3, type, sculptor) sup51(sculptor, artist)
H2new= type(&r3, artist)
n2
H2old= triple(painter, sc, artist) triple(flemish, sc, painter) triple(cubist, sc, painter) triple(&r2, type, painter) sup51(painter, artist)
H2new= type (&r2, artist) sup51(flemish, painter) sup51(cubist, painter)
n1
H3old= triple(painter, sc, artist) triple(sculptor, sc, artist) sup51(painter, artist) sup51(sculptor, artist) type(&r3, artist) type(&r2, artist)
H3new= { }
n4
H3old= triple(cubist, sc, painter) triple(&r1, type, cubist) sup51(cubist, painter)
H3new= type(&r1, painter)
n5
H3old= triple(flemish, sc, painter) sup51(flemish, painter)
H3new= { }
n2
H4old= triple(painter, sc, artist) triple(flemish, sc, painter) triple(cubist, sc, painter) triple(&r2, type, painter) sup51(flemish, painter) sup51(cubist, painter) sup51(painter, artist) type (&r1, painter)
H4new= type(&r1, artist)
n1
H5old= triple(painter, sc, artist) triple(sculptor, sc, artist) type(&r3, artist) type(&r2, artist) type(&r1, artist)
H5new= { }
&r1: picasso
&r2: vangogh
&r3: rodin
key | responsible node
artist | node n1
painter | node n2
scultptor | node n3
cubist | node n4
flemish | node n5

6.3 Handling Node Failures

MS works similarly with FC*. However, less nodes are involved in the reasoning process since the generated triples concern only part of the total data stored in the network. Triples generated by MS are sent and stored in the system as the initial triples and thus, the redundancy mechanisms of the underlying DHT take over. We note that at any moment, only one node in the network is responsible for a specific key and thus, for computing the inferences that concern this key.

If a node that is part of the computation tree of MS fails then the nodes can take the following actions. If it is the root node that failed, the requestor node resends its request after a certain timeout has passed and it received no acknowledgment. For the rest cases and to ensure complete results, supplementary and magic predicates are replicated together with the stored triples. This means that whenever a node $n$ produces a magic or a supplementary predicate for a specific key $k$, then the node sends these predicates also to a neighbour node of $n$ which will be responsible for the key $k$ in case $n$ fails. In this way, if a node fails and its parent node does not receive an ACK after a certain timeout, the parent node resends the message and the new node that is now responsible for the specific key is able to continue the reasoning process. The local lists used in the algorithm (e.g., $infTriples$) are only used for preventing the computation of information that has been already computed. If they are not replicated, it does not affect the completeness of the algorithm but may lead to computing the same information more than once.

## 7 An Analytical Cost Model

In this section, we present an analytical cost model for algorithms FC*, BC and MS presented earlier. We will show in the experimental evaluation that our implementation follows the predictions of this cost model. The results of our cost model could be used by users to determine which of the reasoning algorithms would be suitable for their application and available resources, as well as it could be used in the optimization phase of a distributed query processing algorithm for choosing an optimal query plan.

We focus on the frequently used query types `(X, type, a)`, which asks for all the instances of class `a` in an RDFS hierarchy, and `(X, p, Y)` which asks all subjects and objects of property `p`. As we have already seen, the algorithms are able to answer any type of queries considered in the paper.

We assume $mrdf$ graphs. Triples containing $\rho df$ vocabulary except from `type` form our RDF Schema $S$ and we

**Table 6** Notations used in the cost model

| Symbol | Explanation |
|---|---|
| $\|S\|$ | Number of schema triples |
| $\|D\|$ | Number of data triples |
| $d$ | depth of a class (property) hierarchy |
| $b$ | branching factor of a class (property) hierachy |
| $\|H\|_S^{init}$ | Number of schema triples concerning a hierarchy $H$ before any inference has taken place |
| $\|H\|_D^{init}$ | Number of data triples concerning a hierarchy $H$ before any inference has taken place |
| $n$ | Number of nodes in a class (property) hierarchy |
| $n^\ell$ | Number of descendant nodes of a node at level $\ell$ in a hierarchy $H$, including itself |
| $ch$ | Number of class hierarchies in $S$ |
| $ph$ | Number of property hierarchies in $S$ |
| $r$ | rank of a class (property) in a hierarchy with a Zipfian distribution |
| $\|I_u\|$ | Number of instances of a certain class (property) in a hierarchy (with uniform distribution) |
| $\|I_r\|$ | Number of instances of a certain class (property) with rank $r$ in a hierarchy (with Zipfian distribution) |
| $\|H\|_S^{\text{closure}}$ | Number of schema triples concerning a hierarchy $H$ after the transitive closure computation of $H$ |
| $\|H\|_D^{\text{closure}}$ | Number of data triples concerning a hierarchy $H$ after the transitive closure computation of $H$ |

call them *schema triples*, while the rest of the triples (including `type` triples) form our RDF data $D$ and we call them *data triples*. The RDFS $S$ consists of class and property hierarchies. For simplicity, we assume tree-shaped hierarchies including degenerated trees (i.e., trees that resemble a linked list). Table 6 summarizes the notation used in our cost model.

Let $d$ be the depth of a hierarchy $H$ and $b$ its branching factor. The number of classes (properties) of the hierarchy is $n = \sum_{i=0}^{d} b^i$ with $b \geq 1$ in the worst case scenario, where the tree is complete. If $b > 1$, we have $n = \frac{b^{d+1}-1}{b-1}$, while if $b = 1$, we have $n = d + 1$.

If $H$ is a class hierarchy, we call *instances of* $H$ or *data triples concerning* $H$, the triples of the form `(r, type, c)` where `c` belongs to the class hierarchy $H$. If $H$ is a property hierarchy, we call *instances of* $H$ or *data triples concerning* $H$, the triples of the form `(s, p, o)` where property `p` belongs to the property hierarchy $H$. For a uniform distribution, given the total number of initial instances $|H|_D^{init}$, the number of instances under *each* class (property) is $|I_u| = \frac{|H|_D^{init} \times (b-1)}{b^{d+1}-1}$ if $b > 1$ and $|I_u| = \frac{|H|_D^{init}}{d}$ if $b = 1$. Considering a Zipfian distribution of instances with a skew parameter of 1, a tree node with rank $r$ has $|I_r| = \frac{|H|_D^{init}}{r \times h}$ instances where $h = \sum_{j=1}^{n} \frac{1}{j}$ for $n$ tree nodes. Leaf nodes are given a lower rank.

In the following, we constantly use the result that the total number of subclasses (subproperties) of a class (property) at level $\ell$ of a hierarchy $H$, including itself, is at most $n^\ell = \frac{b^{d-\ell+1}-1}{b-1}$ if $b > 1$ and equal to $n^\ell = d - \ell + 1$ if $b = 1$. The proof is straightforward and is omitted. Furthermore, we utilize the fact that the reasoning and query answering algorithms for the type of queries and rules we consider are essentially transitive closure computations [32].

In the analytical calculations presented below, we start with an RDF(S) database. Then, we apply the FC*, MS and BC algorithms to be able to answer a query of the type mentioned above, and estimate its cost.

## 7.1 Storage Cost Model

We first estimate analytically the costs associated with the storage of triples (given and inferred triples) in all algorithms. These costs are captured by two parameters that we define below: database storage load and number of store messages. Both parameters are measured using a uniform as well as a Zipfian distribution of instances under the hierarchies.

### *7.1.1 Storage Load*

We define as *database storage load* the total number of triples that are stored in the network. In BC, the storage load ($\text{SL}_b$) is three times the number of RDF(S) triples that were inserted in the network based on our indexing scheme. In FC*, triples initially inserted in the network, as well as inferred triples, are stored three times. Therefore, it is sufficient to compute the total number of triples that results from the transitive closure computations of the hierarchy (triples initially in the database plus inferred ones). Then, the database storage load incurred in FC* ($\text{SL}_f$) is three times this total number of triples.

**Lemma 1** *The total number of schema triples of a class (property) hierarchy $H$ after the computation of the tran-*

*sitive closure of H (given triples plus inferred triples) is at most* $|H|_S^{\text{closure}} = \sum_{i=1}^{d} b^i \times i$ *with* $b \geq 1$.

*Proof* For each level $i$ of the tree, we have at most $b^i$ classes (properties), and for each class (property) `a` we have one triple of the form `(a, sc, b)` (`(a, sp, b)`) that links this class (property) with its superclass (superproperty) and we infer $i-1$ triples of the form `(a, sc, z)` (`(a, sp, z)`) for the upper levels of the tree. □

**Lemma 2** *The total number of data triples concerning a class (property) hierarchy H after the generation of the inferred instances of H (given triples plus inferred triples) is at most* $|H|_D^{\text{closure}} = \sum_{i=1}^{d} b^i \times |I_u| \times (i+1)$ *with* $b \geq 1$.

*Proof* Each class (property) has $I_u$ direct instances. For each level $i$ of the tree, we have at most $b^i$ classes, and for each class (property) we have $I_u$ triples and we infer $I_u \times i$ triples of the form `(r, type,` $c_j^i$`)` (`(s,` $p_j^i$`, o)`) for its $i$ superclasses (superproperties). □

Apart from the class or property hierarchies that lead to inferred triples, `dom` and `range` triples may also lead to the inference of new data triples. More specifically, this kind of triples contributes to the generation of `type` triples (e.g., see rules 6, 7 of Table 2). However, `type` triples may have already been computed from the `sc` relations as discussed in Sect. 4.5. Therefore, we have to account for the triples that are not already generated from the transitive closure of the data triples.

**Lemma 3** *Assume a triple* $t$ *=*`(p, dom, a)` *(or* `(p, range, a)`*) where* `p` *is at the level* $\ell_p$ *of a property hierarchy and* `a` *is at the level* $\ell_c$ *of a class hierarchy H. If* $|I_u|$ *is the total number of instances under each property* $\text{p}_i$ *of the property hierarchy (i.e., triples of the form* `(s,` $\text{p}_i$`, o)`*), assume that* $|I_u|^-$ *is the number of instances of* $\text{p}_i$ *which have subjects (or objects) which are instances of a class* `b` *which is at a higher level* $\ell'_c$ *of the same hierarchy H* ($\ell'_c < \ell_c$)*. The total number of new data triples inferred because of t is* $|H|_D^{\text{closure}} = |I_u| \times n^{\ell_p} \times (\ell_c + 1) - |I_u|^- \times n^{\ell_p} \times (\ell'_c + 1)$.

*Proof* The triples inferred because of the data triples concerning the hierarchy under property `p` is equal to $|I_u| \times n^{\ell_p} \times (\ell_c + 1)$, since class `a` has $\ell_c$ ancestor classes and property `p` has $n^{\ell_p}$ descendant properties including itself. However, the set of these inferred triples include triples that are computed from the transitive closure of the data triples of class `b`. Therefore, we subtract the number of triples that have been already inferred for the ancestor classes of class `b` which has $\ell'_c$ ancestor classes. □

Based on the above lemmas, the total number of triples stored in the nodes of the network after FC* has terminated depends (i) on the number of class and property hierarchies the RDFS ontology contains as well as the `dom` and `range` triples that connect a property hierarchy with a class hierarchy. Therefore, the total number of triples stored and the storage load will be at most three times the sum of the above formulas.

So far we assumed a uniform distribution of instances. However, depending on the distribution of instances, the storage load of FC* changes based on the number of instances per class (property) (i.e., $|I_u|$ and $|I_r|$). For the Zipfian distribution, we also made use of the following proposition.

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

For a query of type `type(a, X)` (or `newTriple(*, p, *)`), the database storage load for the MS algorithm ($\text{SL}_m$) depends on the level of the class (property) that is queried and the number of inferred instances for this class (property).

**Lemma 4** *The total number of inferred instances of a class (property) at level* $\ell$ *of a hierarchy H is at most* $|H^\ell|_D^{\text{closure}} = |I_u| \times \sum_{i=1}^{d-\ell} b^i \times i$.

*Proof* We can think of a hierarchy under a class at level $\ell$ as a hierarchy with depth $d-\ell$. Therefore, the result follows directly from Lemma 2. □

Based on the above Lemma, the database storage load for MS for a uniform distribution is at most $3 \times (|S| + |D| + \sum_{i=1}^{d-\ell} b^i \times i)$. For a Zipfian distribution, we would have $\sum_{r=1}^{N_\ell} |I_r| \times \ell_r$ inferred instances, where ($N_\ell = b^{d-\ell+1} - 1)/(b-1)$.

Table 7 summarizes the database storage load of FC*, BC and MS for both kinds of instance distributions.

### 7.1.2 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 database storage load incurred, i.e., $SM_b = 3 \times (|S| + |D|)$. It is also independent of the instance distribution.

FC* may generate duplicate triples and, therefore, sends more messages than the storage load. For example, redundant messages may be sent in the case where we have `dom` and `range` statements which connect a property with a class hierarchy. The number of messages sent because of the transitive closure computation of schema and data triples of either class or property hierarchies is the same as the storage

**Table 7** Storage cost summary

| Storage cost | Uniform | Zipfian |
|---|---|---|
| $\mathrm{SL}_b$ | $3 \times (\lvert S\rvert + \lvert D\rvert)$ | $3 \times (\lvert S\rvert + \lvert D\rvert)$ |
| $\mathrm{SL}_f$ | $3 \times \Big[\sum_{j=0}^{ch+ph}\Big(\sum_{i=1}^{d_j}(b_j^i \times i + b_j^i \times \lvert I_u\rvert_j \times (i+1)) + \sum_{j=0}^{dr}(n_j^{\ell p_j} \times (\lvert I_u\rvert_j \times (\ell c_j + 1) - \lvert I_u\rvert_j^- \times (\ell c'_j + 1))\Big)\Big]$ | $3 \times \Big[\sum_{j=0}^{ch+ph}\Big(\sum_{i=1}^{d_j} b_j^i \times i + \sum_{r=1}^{N_j}(\lvert I_r\rvert_j \times \ell_r)) + \sum_{j=0}^{dr}(\sum_{r=1}^{N_j}(\ell_r \times (\lvert I_r\rvert_j \times (\ell c_j + 1) - \lvert I_r\rvert_j^- \times (\ell c'_j + 1)))\Big)\Big]$ |
| $\mathrm{SL}_m$ | $3 \times (\lvert S\rvert + \lvert D\rvert + \sum_{i=1}^{d-\ell} b^i \times i)$ | $3 \times (\lvert S\rvert + \lvert D\rvert + \sum_{r=1}^{N_\ell} \lvert I_r\rvert \times \ell_r)$ |
| $SM_b$ | $3 \times (\lvert S\rvert + \lvert D\rvert)$ | $3 \times (\lvert S\rvert + \lvert D\rvert)$ |
| $SM_f$ | $3 \times \Big[\sum_{j=0}^{ch+ph}\Big(\sum_{i=1}^{d_j}(b_j^i \times i + b_j^i \times \lvert I_u\rvert_j \times (i+1)) + \sum_{j=0}^{dr}(n_j^{\ell p_j} \times \lvert I_u\rvert_j \times (\ell c_j + 1))\Big)\Big]$ | $3 \times \Big[\sum_{j=0}^{ch+ph}\Big(\sum_{i=1}^{d_j} b_j^i \times i + \sum_{r=1}^{N_j}(\lvert I_r\rvert_j \times \ell_r)) + \sum_{j=0}^{dr}(\sum_{r=1}^{N_j}(\ell_r \times \lvert I_r\rvert_j \times (\ell c_j + 1))\Big)\Big]$ |
| $SM_m$ | $3 \times (\lvert S\rvert + \lvert D\rvert) + \sum_{i=1}^{d-\ell} b^i \times i$ | $3 \times (\lvert S\rvert + \lvert D\rvert) + \sum_{r=1}^{N_\ell} \lvert I_r\rvert \times \ell_r$ |

load. However, in the presence of `dom` and `range` statements, the number of store messages sent by FC* will be as follows.

**Lemma 5** *Assume a triple $t$ = (`p, dom, a`) ((`p, range, a`)) where `p` is at the level $\ell_p$ of a property hierarchy and `a` is at the level $\ell_c$ of a class hierarchy $H$. The total number of messages sent because of $t$ is at most $|H|_D^{\text{closure}} = |I_u| \times n^{\ell_p} \times (\ell_c + 1)$.*

*Proof* We have $|I_u| \times n^{\ell_p}$ instances of property `p`, i.e., $|I_u|$ instances for each subproperty of `p` including itself. Class `a` has $\ell_c$ ancestor classes. For each instance of `p`, one triple is generated that declares an instance of class `a` and one for each ancestor of `a`. □

The number of messages sent by MS ($SM_m$) is equal to the database load incurred and depends on the level of the class whose the instances are asked.

The number of messages sent is depicted in Table 7 for both kinds of instance distribution.

### 7.2 Querying Cost Model

In this section, we estimate the cost of answering the query `(X, type, c)` or `newTriple(*, p, *)` where class `c` or property `p` is at level $\ell$ of a class (property) hierarchy.

#### 7.2.1 Query Messages

We define as *query messages* the messages sent while answering a query. The cases of FC/FC* and MS is straightforward since just one message is sent to the node responsible for class `c` (property `p`). In BC, the number of messages sent ($QM_b$) is as many as the number of the subclasses of class `c` (subproperties of property `p`). Therefore, we have $QM_b = n^\ell - 1$. The distribution of the instances does not affect the number of messages sent for the query answering.

For the `type` queries of the form `(X, type, a)`, if we also have `dom`, `range` triples more messages may be sent. In this case, let $dr$ be the number of such triples that connect a property `p` with a class in the hierarchy of `a`. Then, the number of messages sent by BC is $QM_b = n^\ell - 1 + dr \times n^{\ell p}$. This is because it sends one message for each such triple found and then as many messages as the number of descendant nodes of property `p`.

## 8 Experiments

In [35], we showed that a forward chaining algorithm is constrained by a small number of triples in a DHT environment. In this section, we present an experimental evaluation of the backward chaining and magic sets algorithms described in this paper. All algorithms have been implemented in our system Atlas, which is built on top of the Bamboo DHT[11] [59]. In our algorithms, we have also utilized the dictionary encoding implemented in Atlas, where URIs and literals are mapped to integer identifiers. We do not elaborate on this method since this is out of the scope of this paper. More details can be found in [37].

### 8.1 Experimental Setup and Datasets

We tested our system in a local shared cluster[12] consisting of 41 commodity machines with two processors at 2.6GHz and 4GB memory each. We used 39 of these machines where we run multiple instances of Atlas on each machine (i.e., up to 4 Atlas nodes per machine). This allowed us to build networks of up to 156 Atlas nodes in total.

The datasets we use are taken from two different benchmarks as well as from real datasets. The first benchmark

[11] http://bamboo-dht.org/

[12] http://www.grid.tuc.gr/

**Table 8** LUBM atomic queries

| Query notation | Query |
|---|---|
| LQ1 | `X: (X, type, Student)` |
| LQ2 | `X: (X, type, Faculty)` |
| LQ3 | `X: (X, type, Organization)` |
| LQ4 | `X: (X, type, Publication)` |
| LQ5 | `X, Y: (X, degreeFrom, Y)` |
| LQ6 | `X, Y: (X, memberOf, Y)` |

**Table 9** DBpedia queries

| Query notation | Query |
|---|---|
| DQ1 | `X: (X, type, Band)` |
| DQ2 | `X: (X, type, Organisation)` |
| DQ3 | `X: (X, type, Politician)` |
| DQ4 | `X: (X, type, Work)` |
| DQ5 | `X: (X, type, Politician) ∧ (X, birthPlace, Y)` |
| DQ6 | `X, Y: (X, type, Work) ∧ (X, genre, Y)` |
| DQ7 | `X: (X, type, Work) ∧ (X, genre, Pop)` |
| DQ8 | `X, Y: (X, type, GrandPrix) ∧ (X, location, Y)` |
| DQ9 | `X: (X, type, Event)` |

we used is the RBench generator[13] [70] which produces RDF(S) data synthetically. 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 generated datasets contain only `type` and `sc` triples. The queries we measure are queries that ask for all the transitive instances of the root class of the RDFS hierarchy. We used class hierarchies of depth 2-6 (corresponding to 7-127 RDFS classes). 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.

The second benchmark we use the Lehigh University benchmark (LUBM) [22] that provides synthetic RDF datasets of arbitrary sizes[14]. LUBM consists of a university domain ontology modeling an academic setting and is widely used for testing RDF stores. Each dataset can be defined by the number of universities generated and is expressed in RDF. For example, the dataset LUBM-1 involves one university, while the dataset LUBM-10 involves 10 universities. The more universities are involved in the data generation the more triples are produced. In all datasets generation, we set the seed to 0 which is used for random number generation. Since our intention in this paper is to compare the reasoning algorithm, apart from the queries provided by the benchmark, we also use atomic queries. These queries are shown in Table 8. Namespaces are omitted. Therefore, we use as a query workload queries that ask for instances of certain classes included in the LUBM ontology or queries that ask for triples with a specific property. For completeness, we also present some results of the benchmark queries. However, the algorithms that can process conjunctive queries can be found in [37,36].

Finally, we use real data extracted from DBpedia[15] an initiative which provides structured information in RDF from Wikipedia. We randomly picked data from the infobox types and properties of DBpedia, while we used the whole DBpedia ontology as our RDFS. The dataset we use consists of 2,372,539 triples and 5,633 RDFS triples (3,899,020 inferred triples). We have hand-picked a set of 9 queries which require reasoning. The queries are shown in Table 9 in their conjunctive form. In the following experiments, all query results are averaged over 10 runs using the geometric mean which is more resilient to outliers.

## 8.2 Storing RDF(S) Data

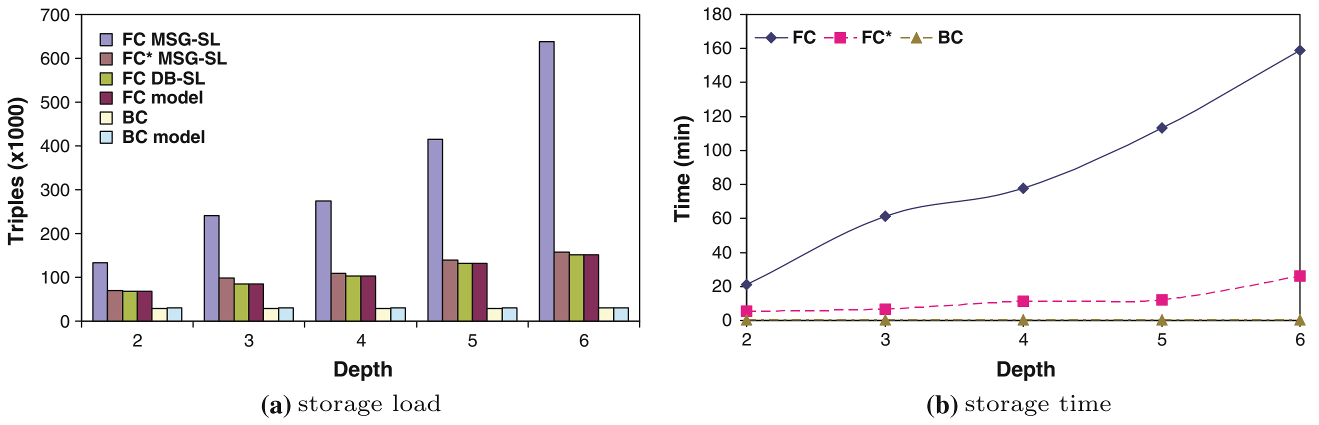
First, we compare the performance of the forward chaining algorithm (FC*) with the backward chaining algorithm (BC) when storing RDF(S) data in the network. Apart from the forward chaining algorithm described in Sect. 4 (FC*), we present results from the forward chaining algorithm we have presented in [35] (called FC in the graphs) for demonstrating the redundancy phenomenon. For this set of experiments, we have generated 10,000 instances uniformly distributed under an RDFS class hierarchy of varying depth using the RBench generator. We inserted the data in the network together with the corresponding RDFS class hierarchy and measured the following metrics: network traffic, the load incurred at the nodes and the time required for all triples to be stored in the network (i.e., for backward chaining just the given triples and for forward chaining the given and the inferred triples).

Figure 9a shows the total load incurred at all nodes of the network when storing RDF(S) data. We define as *database storage load* of a node *n* the number of triples that *n* stores locally in its database. We also define as *store message load*

13 http://athena.ics.forth.gr:9090/RDF/RBench/

14 http://swat.cse.lehigh.edu/projects/lubm/

15 http://dbpedia.org



**(a)** storage load **(b)** storage time

**Fig. 9** Storage load and time

of a node $n$ the number of triples that $n$ receives to store in its local database. If the triples that are sent to node $n$ to be stored are not already stored in its local database, then the database storage load is equal to the store message load. If at least one triple is already stored at $n$, then the store message load is greater than the database storage load. The difference of these two metrics allow us to estimate the redundant triples that are sent to node $n$. The total database storage load (DB-SL) is the total number of triples stored in the network and the total store message load (MSG-SL) is the sum of the store message load incurred at all nodes of the network. Figure 9a shows DB-SL and MSG-SL for algorithms FC, FC*, and BC. DB-SL and MSG-SL of BC are equal and we only depict them by the single bar BC. The bars FC model and BC model depict the total database storage load as computed by the analytical model of Sect. 7.1.1 and demonstrate its accuracy.

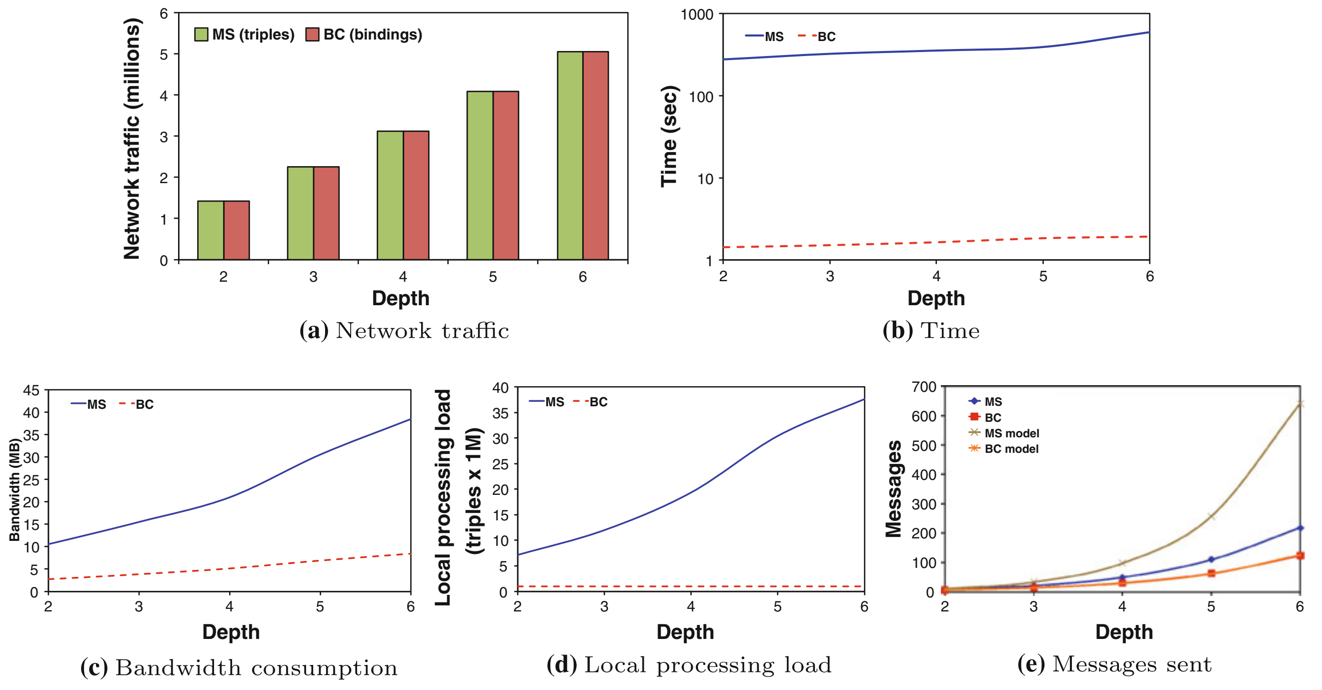
BC's storage load is significantly lower than FC and is independent of the tree-depth. Both FC and FC* cause the same database storage load and, therefore, we depict it with bar FC DB-SL. However, the store message load of FC and FC* has a significant difference. While MSG-SL of FC grows very abruptly with the tree depth, MSG-SL of FC* increases more gently and is very close to the database storage load incurred in the network. This means that FC* does not produce the amount of redundant triples generated by FC. The only redundant triples that are generated by FC* in this experiment are `sc` triples which are generated by two different nodes because of our triple indexing scheme. These triples, however, are much fewer than the total number of triples stored and generated and thus the redundancy shown by the graph is very low.

In Fig. 9b, we show the time needed by each approach to complete the insertion of RDF(S) data. In BC, this time represents the time needed until all given triples are stored at the respective nodes. In FC and FC*, this time represents the time required for the algorithm to terminate, i.e., to reach a fixpoint. Certainly, the time required by BC to store RDF(S) data is independent of the tree depth. The time is negligible and thus the line is very close to the $x$ axis. On the contrary, FC and FC* require a time proportional to the tree depth. FC requires a larger amount of time to reach a fixpoint than FC*, a phenomenon that is magnified as the tree depth grows. Generally, both forward chaining algorithms require an enormous amount of time to complete which made the measurement of inserting more than 10,000 triples of the RBench dataset infeasible. This is mainly due to the number of messages sent in the network which causes bandwidth congestion as well as overload to the nodes that have to process this amount of messages.

## 8.3 Comparing Backward Chaining with Magic Sets

In this section, we compare the backward chaining algorithm (BC) with the algorithm using the magic sets transformation (MS). In the following set of experiments, we stored 1,000,000 instances of the RBench dataset for RDFS class hierarchies of varying depth using a uniform distribution. Then, we send a request with the predicate `m_type` and argument the root class for MS, while we run the query that asks for all instances of the root class for BC.

Figure 10a shows the network traffic in terms of the number of triples transferred in the network for MS and in terms of the number of bindings transferred for BC. Both of these metrics depict the number of generated inferences from the two algorithms. We observe that the total number of triples generated from MS and the total number of bindings generated from BC are equal. This shows that the magic sets rewriting algorithm is equivalent with a backward chaining algorithm where we materialize the produced inferences, with the difference that the former one works bottom-up while the latter one top-down. In fact, in the literature, the magic sets rewrit-



**(a)** Network traffic **(b)** Time

**(c)** Bandwidth consumption **(d)** Local processing load **(e)** Messages sent

**Fig. 10** MS vs. BC for RBench dataset

ing algorithm is often considered as an efficient algorithm for creating materialized views and for propagating changes to the views through incremental maintenance [65].

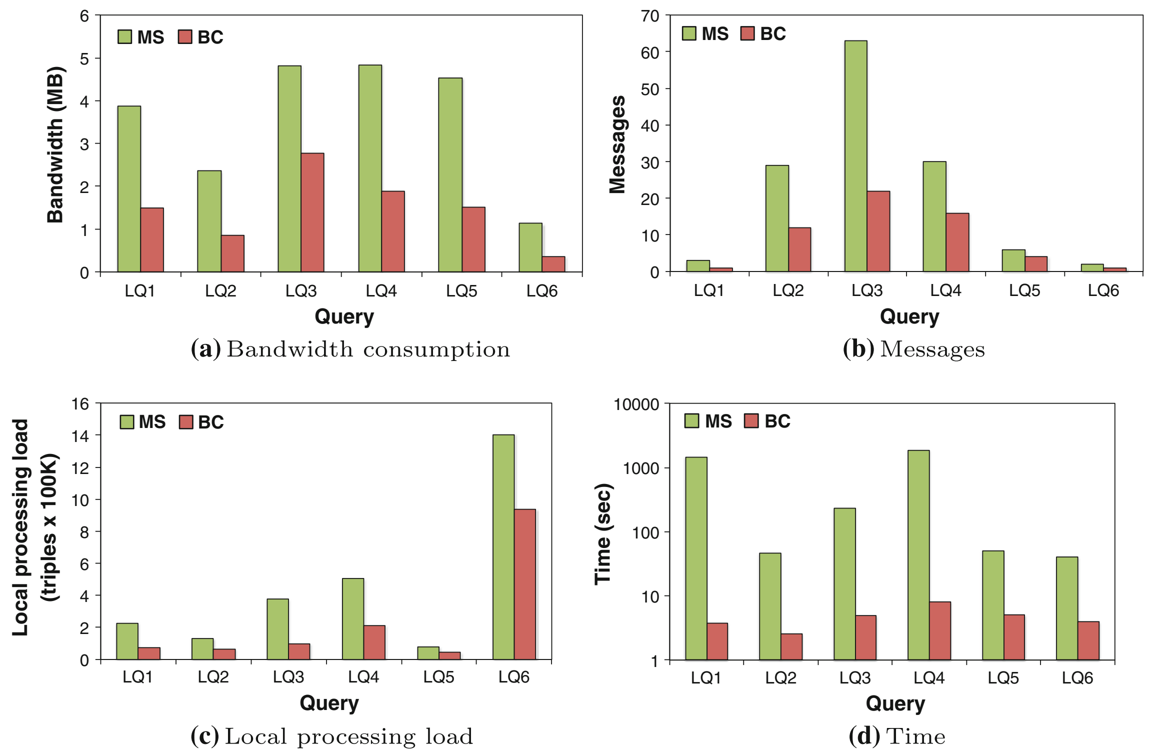
Although the two algorithms are equivalent in terms of the number of inferences, in our experiments, the amount of time required from MS to terminate is greater than the time required from BC to answer the corresponding query. Figure 10b depicts the time difference. Note that *y* axis shows the time in seconds on a logarithmic scale. We observe that BC outperforms MS by two orders of magnitude. The reasons for this are explained below.

Firstly, the messages sent during MS contain whole triples, while the messages sent during BC contain only bindings, i.e., only the object of the matching triples. This fact is depicted in Fig. 10c by the bandwidth consumption of the two algorithms. We observe that the total bandwidth consumed by MS is about three times greater than the total bandwidth spent by BC.

Secondly and most notably, the load incurred at each node for processing its local triples is greater for MS resulting in a considerable time difference. We define as *local processing load* of a node the number of triples that the node retrieves from its local database and should process to determine if new triples can be generated (for the case of MS) or new queries should be sent (for the case of BC). Figure 10d shows the *total local processing load* incurred in all the nodes of the network for both algorithms, and thus, demonstrates the total work required by the two algorithms. While BC incurs a constant load regardless of the depth of the RDFS hierarchy, the total load in MS is increasing significantly with the depth of the RDFS class hierarchy and is much greater than in BC.

The reason behind this is that MS sends many small messages, while BC sends only a few large messages and thus, MS requires more local processing effort from the nodes of the network. Although Fig. 10a demonstrates that the total number of values (i.e., triples or bindings) sent by both algorithms is equal, Fig. 10e shows that the number of store messages (StoreMsg) sent by MS containing the inferred triples is greater than the number of response messages (BCRDFSResp) sent by BC containing the bindings of matching triples. In BC, a node sends a response back to its parent node only after it has collected all answers from its children. On the contrary, in MS whenever a node receives a MSReq, it sends a StoreMsg message if any new triples are generated. As a consequence, the local load incurred at each node is affected, as in both algorithms, each time a node receives a message, it retrieves from its local database matching triples to process them. Figure 10e also shows the predicted numbers of messages of both algorithms computed by the analytical model. BC sends exactly the number of messages computed by the analytical model and thus, the two lines in the graph overlap, while MS sends less messages compared to the number of messages computed by the analytical model due to the MultiPut functionality.

The RBench dataset involves only one RDFS class hierarchy and the potentials of MS are not fully exploited. In the next experiment, we use the LUBM dataset whose schema contains several independent class and property hierarchies



**(a)** Bandwidth consumption **(b)** Messages **(c)** Local processing load **(d)** Time

**Fig. 11** MS vs. BC for LUBM-20

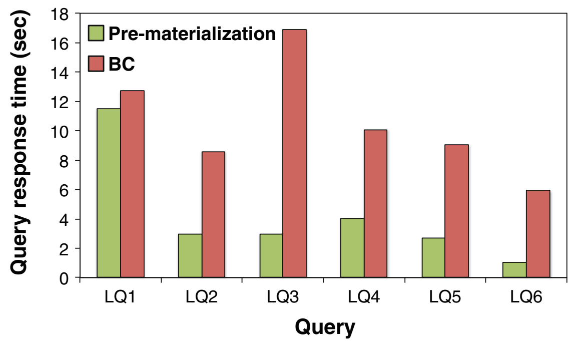
which are linked through `dom` and `range` statements. We create a network of 156 nodes in the cluster and store the complete LUBM-20 dataset consisting of 2,782,435 triples. The queries we use for evaluation are shown in Table 8. For MS, we sent a request with the predicate `m_type` or `m_newTriple` and argument the class name or the property name, while for BC we run the respective query. Figure 11 shows the bandwidth consumption, the total number of messages sent, the total local processing load and the time required for each algorithm to terminate. In this experiment as well, we observe that BC outperforms MS. Since we deal with a bigger dataset, the advantage of using BC is more evident, as shown in Fig. 11d. As a conclusion, BC outperforms MS for two reasons: (1) due to the increased local load incurred at each node and (2) due to the bandwidth consumed during MS.

## 8.4 Query Performance

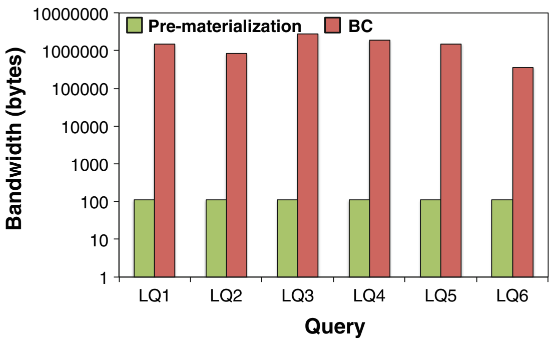
In this section, we explore the query performance of the backward chaining compared to the case where inferred triples have been materialized and stored in the network. This can be done either by precomputing the full RDFS closure or by precomputing only the triples that concern a specific query using the magic sets algorithm.

In this set of experiments, we use both the LUBM-50 and the DBpedia dataset. In the first case, we store all inferred triples of the datasets and then run the query without any reasoning involved. In the second case, we store only the initial dataset and use backward chaining during query evaluation to return a complete answer to the query. The queries we used are shown in Tables 8 and 9. Figure 12 depicts the results of different metrics for these queries.

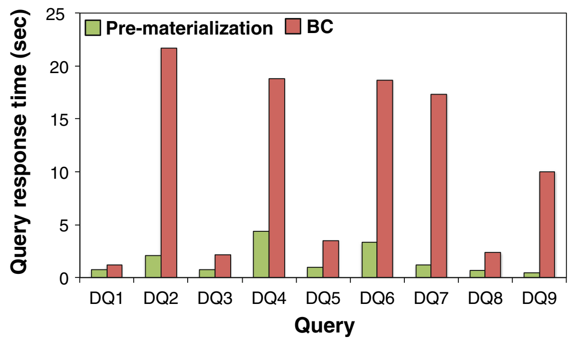
In Fig. 12a, c, we depict the time required to answer each query, i.e., the time from the moment a node receives the query request until the moment it receives all the answers. Depending on the RDFS schema, BC sends a different number of messages to retrieve the required inferences. On the contrary, one message is sent during query evaluation of a pre-materialized dataset to a single node which retrieves all results from its local database. Table 10 shows the total number of messages sent at each case for the different queries. For example, the LUBM schema does not contain many inferences for class `Student` and thus for query LQ1 that asks for the instances of class `Student` the number of messages sent by BC is only 1. On the other hand, for query LQ2 that asks for the instances of class `Faculty`, BC sends 13 messages to retrieve all inferences. Certainly, this leads to a greater query response time difference between the algorithm that uses backward chaining and pre-materialization. We also observe from Table 10 that because the DBpedia ontology is richer and more expressive than the one of LUBM, more messages are sent.



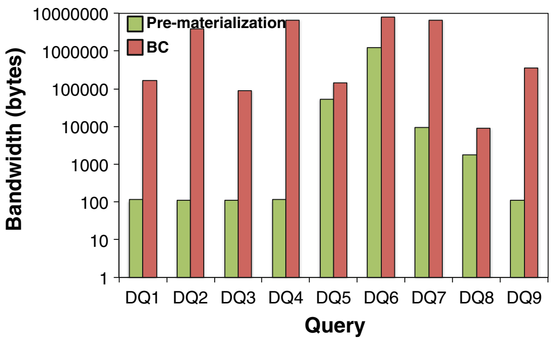
**(a)** Query Response time



**(b)** Bandwidth consumption



**(c)** Query Response time



**(d)** Bandwidth consumption

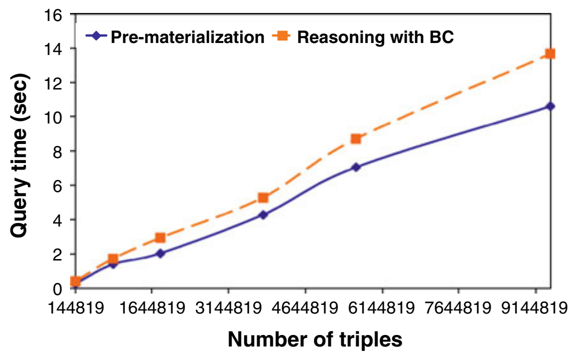
**Fig. 12** Query performance of BC for LUBM (*top*) and DBpedia (*bottom*)

**Table 10** Messages sent for various queries

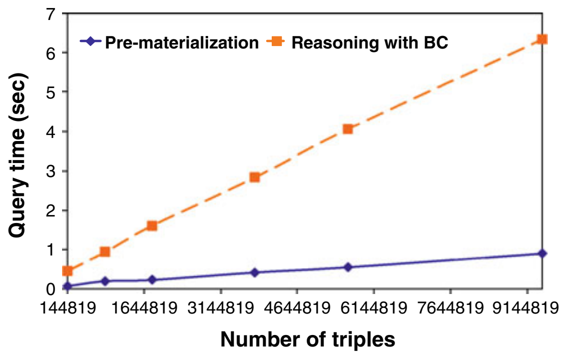
| LUBM query | Pre-materialization | BC |
|---|---|---|
| LQ1 | 1 | 1 |
| LQ2 | 1 | 13 |
| LQ3 | 1 | 22 |
| LQ4 | 1 | 16 |
| LQ5 | 1 | 4 |
| LQ6 | 1 | 20 |
| LQ7 | 1 | 41 |
| **DBpedia query** | **Pre-materialization** | **BC** |
| DQ1 | 1 | 2 |
| DQ2 | 1 | 258 |
| DQ3 | 1 | 12 |
| DQ4 | 1 | 110 |
| DQ5 | 2 | 13 |
| DQ6 | 2 | 111 |
| DQ7 | 2 | 114 |
| DQ8 | 2 | 46 |
| DQ9 | 1 | 124 |

As a result of the number of messages sent, bandwidth consumption is also greater when backward chaining is used. Figure 12b, d shows the bandwidth consumed because of the partial answers that are transferred through the network. Note that these graphs do not contain the bandwidth consumed for returning the final results to the query requestor node since it is the same for both approaches. Another parameter that affects query response time is the selectivity of the query, i.e., the number of results that each query returns. We see that the query response time for class `Student` is greater than the query response time for class `Faculty`. This is because the number of results for query `Student` is 430,114 while it is 35,973 for `Faculty`.

We have also experimented with different dataset sizes. In a network of 156 nodes, we store datasets from LUBM-1 to LUBM-50. Every time we measure the query response time of queries LQ1 and LQ2 that ask for the instances of `Student` and `Faculty`, respectively. Figure 13 shows the behavior of our system with and without the reasoning process as the dataset stored in the network grows. Table 11 shows the sizes of all the datasets used together with the number of results that each query returns. In the graphs of Fig. 13, we show that query response time scales in a linear fashion with the number of triples stored in the network. This is a result of two factors. First, the local database of each node grows and as a result local query processing becomes more time-consuming. Second, the size of the answer set of the queries grows as the number of triples is increasing resulting in a greater bandwidth consumption. As we mentioned before, BC performs close to the query evaluation without any reasoning for the query that asks for the instances of



**(a)** Query LQ1 (Student)



**(b)** Query LQ2 (Faculty)

**Fig. 13** Increasing the number of triples stored

**Table 11** LUBM datasets and queries

| Dataset | Triples | Inferred triples | Answers of query `ub:Student` | Answers of query `ub:Faculty` |
|---|---|---|---|---|
| LUBM-1 | 103,413 | 144,819 | 6,463 | 540 |
| LUBM-5 | 646,144 | 887,461 | 40,087 | 3,373 |
| LUBM-10 | 1,317,009 | 1,806,023 | 82,507 | 6,843 |
| LUBM-20 | 2,782,435 | 3,812,865 | 174,750 | 14,457 |
| LUBM-30 | 4,109,311 | 5,629,144 | 256,919 | 21,440 |
| LUBM-50 | 6,890,949 | 9,437,221 | 430,114 | 35,973 |

`Student`, while the difference becomes more evident for the query that asks for the instances of class `Faculty`. In all cases, the increase of query response time of backward chaining remains linear with respect to the triples stored.

Finally, in Fig. 14, we show the query response time for all queries provided in the LUBM benchmark using the LUBM-50 dataset. These queries consist from one to six triple patterns. For the conjunctive queries we utilize the QC algorithm of Atlas described in [37] which splits a conjunctive query to its atomic parts and evaluates them in a sequential way. For each triple pattern, the BC algorithm takes place. The overhead of BC is now more evident as the more triple patterns in a query require reasoning the more overhead is added to the final response time. Studying algorithms for a parallel evaluation of conjunctive queries like in [46] is part of our future work and is not the focus of this paper.

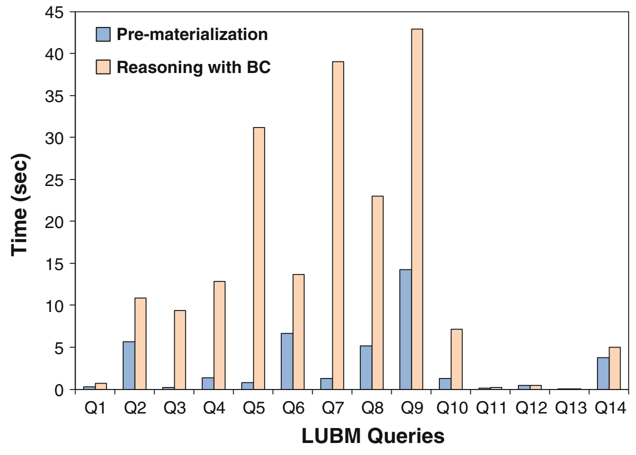
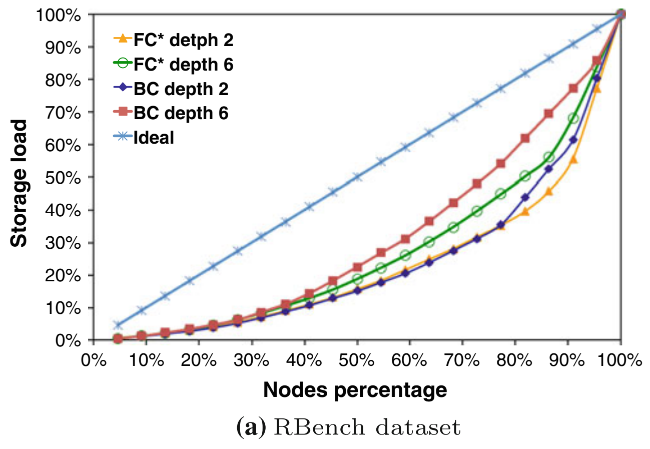


**Fig. 14** BC performance for all LUBM queries

## 8.5 Data Skewness

In this section, we study how the data skewness affects our algorithms. DHTs can suffer from load imbalances [69]. Load imbalances can appear at the level of the key distribution, i.e., keys are not evenly distributed among the nodes of the network, and at the level of the item distribution, i.e., items are not evenly shared among the nodes. The first phenomenon can be easily solved with a consistent hash function and using virtual nodes [38,67]. The second problem occurs mainly due to data skewness. RDF data is highly skewed since various values, and mostly properties such as `rdf:type`, `rdfs:label`, appear very frequently in a dataset. Therefore, most DHT systems which use the triple indexing scheme of [12] suffer from load imbalances concerning storage load.

In this set of experiments, we conduct some measurements concerning the distribution of the database storage load for both algorithms to investigate how each reasoning algorithm



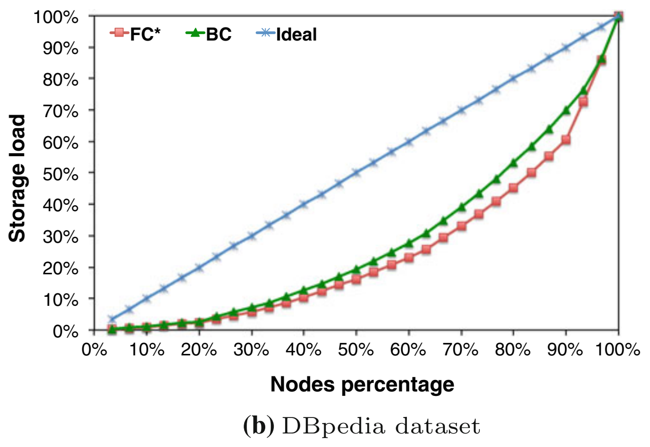


**Fig. 15** Storage load distribution

is affected by the skewness of the data. Figure 15 shows the storage load distribution among network nodes using Lorenz curves as proposed in [57]. Lorenz curves are functions mapping the cumulative percentage of ordered objects to their corresponding cumulative percentage of their size. In our case, the objects are the network nodes and the size is their storage load. We order network nodes from the one that has the least storage load to the one that has the biggest storage load and for each set of nodes we calculate the cumulative storage load (the first set contains only one node, while the last one all nodes). We show in the $x$ axis the *cumulative percentage of the nodes*, while in the $y$ axis *the cumulative percentage of the storage load*. The purpose of Lorenz curves is to show which percentage of the nodes holds which percentage of the total load. In the ideal case where all nodes share the same load, the curve is a straight diagonal line (e.g., 40% of the nodes share 40% of the load). The closer a Lorenz curve is to this diagonal, the better the load distribution is.

In this Fig. 15a, we show the database storage load of BC and FC* for the RBench dataset with tree-depths 2 and 5. As the magic sets algorithm in this experiment generates all the instances of the root class, its storage load distribution is the same with FC* and thus we do not depict it in the graph. For readability reasons, we do not show the intermediate tree-depths, but as the depth of the tree increases the load distribution becomes better for both approaches. For example, for a hierarchy with depth 2, we see that, in BC, around 72% nodes share less than 50% storage load and the other 28% nodes have to deal with the other 50% load. In this graph, we observe that the deeper the hierarchy tree, the better the load distribution is for both backward and forward chaining. This can be explained by the fact that the range of the object value of the triples stored is limited to the number of classes of the hierarchy for this dataset. For a class hierarchy with depth 2, the number of distinct classes is 7 and nodes responsible for these classes are overloaded. As the depth increases, the number of classes increases exponentially and more nodes share the load resulting in a more balanced distribution. Furthermore, while both algorithms have almost the same load distribution for a tree with depth 2, BC distributes the load slightly better than forward chaining when the depth increases to 6. 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 lower levels.

Figure 15b shows the Lorenz curve for the DBpedia dataset. We observe that for both approaches storage load is not equally shared among the nodes. For example, half of the nodes share less than 20% of the load. The most overloaded nodes become even more loaded with the FC* approach as inferred triples are indexed in these nodes.

Load imbalance in a distributed RDF system can have several implications. First, it may lead to the deterioration of the overall system performance. Overloaded nodes need to keep a bigger local database and hence they may become a bottleneck as they will require more time for inserting new data as well as for finding matches to a given triple pattern. Second, if the data is very skewed, some of the nodes may reach their storage capacity limits and thus will not be able to hold any more data.

There are solutions proposed in the literature for load balancing [6,38,69] and both reasoning approaches could benefit from them. In general, load balancing methods can be categorized to data replication and data relocation. Applying data replication to a problem of data skewness does not usually comprise a solution, rather it magnifies the problem by overloading more nodes than necessary. A relocation technique is more suitable for the RDF storage load balancing. In [6], the authors propose a method based on relocation and utilize overlay trees among nodes to keep track of the relocated triples. The set of triples of an overloaded node $n$ is split into two or more equal parts which are then sent to the network nodes $n_i$ that appear to be the least busy. These nodes become children of the original network node in the overlay tree. When a triple pattern should be evaluated at node $n$, it is also broadcasted to all network nodes $n_i$ that are the children of $n$ in the overlay tree. Then, the union of the results from all $n_i$ nodes constitutes the result to the triple pattern submitted to $n$. Another issue that arises in this scenario is the detection of the overloaded nodes. In [6], a sampling technique is proposed where each node sends its statistics to a sample number of nodes and compares its load with the one from the other nodes. If it exceeds a certain percentage threshold, then the node is considered overloaded. Such solutions can be applied to our algorithms as well but we consider this research area out of the scope of this paper.

### 8.6 Lessons Learned

As a conclusion, a forward chaining approach improves the time for answering a query but increases the storage load significantly by generating statements that might never be required by a query. One might prefer to pay this storage cost if enough space is available and fast answering is paramount. Naturally, this method is the preferred one if one wants to compute the complete closure of a given dataset under RDFS entailment. Results from our experimentation show that a simple forward chaining implementation in the DHT as the one we presented in [35] and in Sect. 4 and also presented in BabelPeers [7] suffers from message congestion. While the FC* algorithm we presented in this paper improves on the number of messages sent, we still could not scale to larger datasets mainly due to a straightforward implementation. An appropriate optimization of our implementation would involve optimizing the local database storage at each node (for instance, using an off-the-self RDF store like RDF-3X [54]), and compressing the triples exchanged among the nodes (for instance using a compact representation of RDF such as the ones recently proposed in [19,44]). However, this was not the focus of our work and our implementation served as the targeted proof of concept.

An alternative solution, if materializing the full RDFS closure is paramount, would be to keep all RDFS schema triples at each node locally, and precompute the closure of the data before inserting them into the DHT system. This is possible in the $mrdf$ fragment we consider since schema triples are always generated by two other schema triples and data triples are generated by one schema and one data triple. However, still this solution causes a big storage overhead to the system and is more expensive to maintain in the presence of frequent updates as the whole RDFS closure should be materialized and maintained.

In contrast, a backward chaining approach improves storage load and can scale to bigger datasets in our system. Certainly, this comes at the cost of an increase in query response time. Yet, the query response time of backward chaining increases linearly with the number of triples stored in the network.

The magic sets algorithm constitutes a good compromise between the two algorithms as it exploits the advantages of both approaches and scores between the two. Although in our system, it performs worse than BC, such an algorithm can be proved extremely helpful in application scenarios where the query workload is known a-priori and a pre-computation can be done offline. In such cases, only triples concerning the queries in the workload are inferred and stored in the network which leads to storage savings and a speedup of query performance.

## 9 Related Work

In this section, we survey related work. We cover works on RDFS reasoning in both centralized systems as well as distributed ones of various architectures.

### 9.1 Centralized Systems

A representative centralized RDF store that supports a forward chaining approach is Sesame [11]. Each time an RDF Schema is uploaded in Sesame, an inference module computes the closure of a given dataset under the RDFS entailment and asserts the inferred RDF statements. So, every RDF statement, explicit or implicit, is stored in Sesame's database. Jena [76] has a generic reasoning module designed to allow the usage of any kind of reasoner. Therefore, Jena can eventually support different approaches depending on the underlying reasoner. RDFSuite[16] and specifically RSSDB [4] follows a totally different approach in which the taxonomies are stored using the underlying DBMS inheritance capabilities so that retrieval is more efficient. For example, if a class $c$ is represented as a table (relation) $R$, a subclass of $c$ would be a subtable of $R$ in the underlying DBMS. Nevertheless, this approach is still an *on demand* approach and resembles the backward chaining evaluation algorithm. 3store [24] follows a hybrid approach to gain from the advantages of both approaches and avoid their disadvantages. In [24], the authors have chosen which inference rules will be evaluated a priori using forward chaining when new facts are asserted, and rules which have greater storage load and lower query processing cost will be evaluated on demand with backward chaining and query rewriting. In the Oracle RDBMS [14], RDFS inference is done at query execution

[16] http://139.91.183.30:9090/RDF/

time using appropriate SQL queries to the underlying relations. However, if a rule is used frequently, then the system can determine that inferencing can be done using forward chaining to pre-compute the inferred triples and store them in a separate relation. Virtuoso[17] also provides support for RDF(S) reasoning [17]. Virtuoso's SPARQL implementation supports inference at run time by rewriting the query appropriately to retrieve all inferred answers. Finally, GiaBATA [30] is a prototype system which uses logic programming approaches coupled with a persistent relational database for implementing SPARQL with dynamic rule-based inference. In [31], the authors of GiaBATA employ different optimization techniques like magic set rewriting in an effort to stay competitive as they extend SPARQL to be able to use a custom ruleset for specifying inferences.

## 9.2 Peer-To-Peer Networks

A DHT-based RDF store that is closely related with our work is BabelPeers [7] which was the first system to support RDFS reasoning using DHTs. It is implemented on top of Pastry [60] and only a forward chaining approach is supported. RDF(S) triples are distributed in the network using the DHT protocol. 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 inference rules and sends them to be stored in the network. [7] presents no experimental evaluation of the forward chaining algorithm. The results of our experiments show how expensive a forward chaining algorithm is in terms of storage load, time and bandwidth.

In [18], the authors present a DHT system called DORS for distributed ontology reasoning. All nodes share the same TBox, while the instances of ABox are distributed in the network using a DHT partitioning scheme. Each node uses a DL reasoner to infer the complete subsumption relationships among classes and properties of the TBox. Although TBox reasoning is taking place at each node independently, ABox reasoning is performed in a distributed manner iteratively until no new inferences are produced, similarly with our FC approach. A prefetch procedure retrieves the required data before the rule engine starts the reasoning process. Then, the inferred assertions are distributed according to the DHT partitioning scheme until no new assertions are generated. The experiments presented in [18] are performed in a network of up to only 32 nodes and for a small dataset.

In [3], a peer data management system is presented, called SomeRDFS, where peers are connected through semantic mappings using as a data model RDF. These mappings specify the relationships between RDFS classes. Then, queries are rewritten using these mappings to find the complete answer and evaluated at the appropriate peers. However, the architecture is different from ours, since the mappings are used to locate the peers that should evaluate a query, and the class of queries they support does not allow RDFS classes or properties to be variables.

In addition to the above works that concentrate only on RDFS reasoning, DHTs have also been used as the underlying infrastructure for distributed SPARQL query processing. RDFPeers [12,13] is the first system which focused on distributed RDF query processing but there is no support for RDFS reasoning. In [45], the authors extend the work of RDFPeers and present two algorithms for the distributed evaluation of conjunctions of triple patterns in a simulated environment. [39,46] focus on the efficient evaluation of SPARQL queries using various kinds of optimization techniques such as parallelization. It is interesting to investigate how such optimization techniques and sophisticated query evaluation algorithms can be combined with the RDFS reasoning algorithms we describe in this paper.

Another category of P2P systems which provide an infrastructure for answering RDF queries is semantic overlay networks. GridVine [1] is a semantic overlay network which provides semantic interoperability through schema mappings and supports conjunctive and disjunctive triple pattern queries without supporting RDFS inference. Schema mappings are also used in [23] to provide semantic mediation between disparate data sources. SQPeer [42] is a middleware for efficient routing and planning of complex queries in a P2P database system, exploiting the schemas of peers. However, such works do not focus on RDFS reasoning.

Finally, a different research area that is related with our work is declarative networking. In [48,47], a variation of Datalog is used for expressing routing protocols in a simple and compact way based on the observation that recursive query languages are very suitable for expressing routing protocols. The implementation of the routing protocols in [48,47] can then be done by evaluating recursive queries in a distributed manner.

## 9.3 Other Distributed and Parallel Architectures

Apart from DHTs which were the first infrastructures proposed for distributed RDF(S) query processing and reasoning, other distributed and parallel computing platforms have been proposed lately. As it has been proven, approaches that are based on distributed computing platforms consisting of powerful clusters and cloud computing platforms using MapReduce can be very scalable for computing the closure of RDF(S) graphs.

Marvin [55,56] is a parallel and distributed platform for RDFS reasoning over large amount of RDF(S) data. Marvin supports a forward chaining approach for RDFS reasoning and runs on DAS-3 (Distributed ASCI Supercom-

[17] http://virtuoso.openlinksw.com/

puter[18]). The creators of MARVIN point out that a distribution of RDF(S) triples based on a DHT can suffer from load imbalances due to the skewness of RDF data [43]. Therefore, they propose an approach of *divide-conquer-swap* where triples are fairly partitioned to all peers, each peer performs the reasoning, repartitions its triples and swaps it with another peer. Although this method produces sound results, it is not complete. In [56], the authors present an analytical model to prove that their system will eventually reach completeness over time. The swapping phase can be either random or using another algorithm called SPEEDDATE in [56] which enables data clustering and thus improves the chances of getting completeness earlier. However, the amount of time required to reach completeness remains questionable. In addition, the system uses an in-memory implementation at each peer which speeds up inferencing significantly but if a node fails and re-joins the network all triples are lost. Datasets used in the experiments of [56] contain up to 14.9 million triples.

In [72], a different forward chaining approach is proposed based on MapReduce [15]. The system is implemented on top of Hadoop[19] and runs on the DAS-3 distributed supercomputer managing to scale to 865 millions of triples. The authors of [72] show that a naive implementation is inefficient due to load-balancing problems and the generation of many duplicate triples and propose three optimizations to achieve an RDFS closure computation more efficiently. Firstly, the RDFS triples are kept in memory since they are fewer than the RDF data triples. Secondly, data are grouped in a way which prevents the generation of duplicate triples and avoids load balancing problems. This is performed using as keys more than one part of the triples in some cases. Finally, the rules are executed in a specific order so that the number of iterations required is limited. Load-balancing issues are also handled by the Hadoop framework which dynamically assigns tasks to optimize the workload of each node. In [72], the authors also utilize a distributed dictionary encoding in MapReduce. More recently in [74], the authors propose QueryPIE, a hybrid rule-based reasoning distributed prototype, which combines forward and backward chaining. The idea behind this method is to precompute the closure of the RDFS schema only, and using this information they perform backward chaining for the OWL Horst fragment. Experimental results are encouraging; however, no formal proofs for the correctness of the methods are given.

Weaver and Hendler [75] considers the problem of producing the full RDFS closure using parallel computing techniques. The authors show that RDFS rules have certain properties that allow for an embarrassingly parallel algorithm. This means that the RDFS reasoning task can be divided into a number of completely independent tasks that can be executed in parallel by separate processes. Similar to [18], a distinction is made between triples that describe RDFS information (referred as ontological triples) and triples that encode RDF data information (referred as assertional triples). The partitioning scheme requires each process to have all ontological triples, while assertional triples are split equally to the processes. Each process iterates over the RDFS rules in the appropriate order until no more inferences are found. The inferred triples produced from a processor are added to the set of triples of the same processor. The authors show that this algorithm is sound and complete with respect to the RDFS rules supported. A disadvantage of this approach is that each process outputs the set of triples to a separate file. This introduces the problem of having different processes producing the same triples and thus the resulting data set contains many duplicate triples. As one would expect, removing duplicates would require much time and would sacrifice the scalability of the algorithm. Experiments were conducted in an Opteron blade cluster using machines with 16GB memory each and the datasets used contained up to $\sim$ 346 million triples from the LUBM benchmark. In [77], the authors show how one can efficiently extract relevant information from the computed RDFS closure of the data.

In [61,62], the authors present the reasoning engine of 4store [25] which runs in a backward chaining fashion using the $mrdf$ fragment of RDFS. 4store is a clustered RDF store which uses the subject of each triple to decide to which cluster node the triple should be stored. Then, a query processing node is responsible for retrieving the required data and compute the answer to a query. The reasoning engine of 4store works in a backward chaining fashion but keeps all RDFS information at one cluster node. The experiments presented in [62] were conducted in 5 Dell PowerEdge R410 machines, each of them with 4 dual core processors at 2.27 GHz, 48GB memory and 15k rpm disks scaling to 138 million triples.

In [20,21], the authors demonstrate how Cray XMT, a shared-memory supercomputer with multithreaded processors, can be used for managing billions of triples. They study the computation of the RDFS closure, dictionary encoding and query processing. All these operations are completely performed in-memory and are specific to this infrastructure.

Other related approaches consider OWL reasoning in parallel platforms. One such approach is presented in [64], where two partitioning approaches are studied. The first one partitions the data which are then processed independently. The second one partitions the rules and each process applies its rules to the complete data set. The authors of [72] go one step further using MapReduce to compute the closure of RDF graphs under the OWL Horst semantics in [73]. In [53], a MapReduce algorithm is presented for classifying $EL^+$ ontologies, following the paradigm of [72,73] for RDFS ontologies. Finally, SAOR is an OWL reasoner using forward chaining with best-effort semantics [28]. In [28], the

[18] http://www.cs.vu.nl/das3/

[19] http://hadoop.apache.org/

authors generalize to arbitrary rule sets for distributed reasoning and show when the system maintains completeness.

## 10 Conclusions

We presented and evaluated both forward and backward chaining algorithms for RDFS reasoning and query answering on top of the Bamboo DHT [59]. We proved the correctness of our algorithm utilizing the minimal deductive system *mrdf* presented in [51]. We revised the backward chaining algorithm of [35] to take into account the inference rules presented in the minimal RDFS fragment of [51]. In addition, we designed and implemented an algorithm which works in a bottom-up fashion using the magic sets transformation technique [8]. We provided a comparative study of our algorithms both analytically and experimentally. The analytical cost model could be used by users to determine which of the reasoning algorithms would be suitable for their application and available resources. Also it can be used in the optimization phase of a distributed query processing algorithm where, for instance, the number of messages sent is essential for choosing optimal query plans. In the experimental evaluation, we deployed our system in a local shared cluster, compared the performance of the algorithms from various perspectives and verified the accuracy of our analytical cost model.

As a conclusion, the forward chaining approach improves the time for answering a query but increases the storage load significantly by generating statements that might never be required in a query and is difficult to update. In contrast, the backward chaining approach improves storage load and can scale to millions of triples. Certainly, this comes at the cost of an increase in query response time. Yet, the query response time of backward chaining increases linearly with the number of triples stored in the network. A magic sets algorithm tries to exploit the advantages of both approaches and scores between the two.

As it has been proven recently, approaches that are based on parallel computing platforms consisting of powerful clusters based on MPI [75] and cloud computing platforms using MapReduce [72] can be very scalable for computing the closure of RDF(S) graphs and for backward reasoning [74], hence they should be preferred to DHT-based approaches if enormous datasets are to be used and appropriate cluster/cloud infrastructures are available. However, none of these approaches has dealt in depth with the theoretical aspects (soundness and completeness of algorithms) we have examined in this paper, thus, they also stand to benefit from the techniques discussed in this paper.

Interestingly, many of the existing cloud-based key-value stores also adopt the hash-based partitioning and replication mechanisms provided by DHTs, such as DynamoDB[20] and Apache Cassandra[21]. They can be viewed as one-hop DHTs, where each node routes a request to the appropriate node directly by maintaining enough routing information [16]. Our RDFS reasoning algorithms can be easily adopted in such key-value stores and scale to a large amount of data.

## 11 Future Directions and Open Issues

One direction to improve our methods is to incorporate load balancing methods so that storage load can be evenly shared among the network nodes. Load balancing methods can be distinguished to data replication and data relocation. Applying data replication to a problem of data skewness does not usually comprise a solution, rather it magnifies the problem by overloading more nodes than necessary. A relocation method might be more suitable for the storage load balancing in a DHT setting such as the one proposed in [6]. Investigating similar techniques for load balancing in combination with node failures for a DHT-based RDF stores remains an open issue.

In addition, an interesting future direction is the adoption of hybrid inference techniques in a DHT, similar to [74], where the schema is shared to all nodes. In such a case, the node that receives a query could directly rewrite it based on the schema and then send the subqueries of the rewritten query to be evaluated in the network. However, we believe that in some simple cases (as in the case of queries of the form `(X, type, a)`), the number of nodes that will have to be visited to evaluate the subqueries is the same as the number of nodes visited in our backward chaining algorithm and, therefore, the gain of such an approach will not be noticeable. However, we believe that there are queries that can benefit from having the schema locally at each node. We plan to investigate the gain of following such an approach as part of our future work.

Finally, in this work, we have assumed a simple query processing algorithm for conjunctive queries which splits the query into its atomic parts and then operate on its atomic part separately. However, more elegant algorithms such as [39,42, 46] are interesting to be investigated and explored towards the direction of incorporating the reasoning approaches we propose in this paper.

## References

1. Aberer K, Cudre-Mauroux P, Hauswirth M, Pelt TV (2004) GridVine: building internet-scale semantic overlay networks. In:

[20] http://aws.amazon.com/dynamodb/

[21] http://cassandra.apache.org/

Proceedings of the 3rd international semantic web conference (ISWC 2004), Hiroshima, Japan
2. Abiteboul S, Hull R, Vianu V (1995) Foundations of Databases. Addison-Wesley, Boston
3. Adjiman P, Goasdou F, Rousset MC (2007) SomeRDFS in the semantic web. J Data Semant 8
4. Alexaki S, Christophides V, Karvounarakis G, Plexousakis D (2001) On storing voluminous RDF descriptions: the case of web portal catalogs. In: Proceedings of the 4th international workshop on the web and databases (WebDB (2001) co-located with SIGMOD 2001), Santa Barbara, California, USA
5. Balakrishnan H, Kaashoek MF, Karger DR, Morris R, Stoica I (2003) Looking up data in P2P systems. Commun ACM 46(2):43–48
6. Battre D, Heine F, Hoing A, Kao O (2006a) Load-balancing in P2P based RDF stores. In: Proceedings of the 2nd international workshop on scalable semantic web knowledge base systems (SSWS 2006, co-located with ISWC 2006), Athens, Georgia, USA
7. Battre D, Hoing A, Heine F, Kao O (2006b) On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores. In: DBISP2P 2006 (co-located with VLDB 2006), Seoul, Korea
8. Beeri C, Ramakrishnan R (1987) On the power of magic. In: PODS '87: proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. ACM, New York, pp 269–284. http://doi.acm.org/10.1145/28659.28689
9. Brickley D, Guha R (2000) Resource description framework (RDF) schema specification 1.0. Technical Report, W3C Recommendation
10. Brickley D, Guha R (2004) RDF vocabulary description language 1.0: RDF schema. Technical Report, W3C Recommendation
11. Broekstra J, Kampman A (2002) Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Proceedings of the 1st international semantic web conference (ISWC 2002), Sardinia, Italy
12. Cai M, Frank M (2004) RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: Proceedings of the 13th world wide web conference (WWW 2004), New York, USA
13. Cai M, Frank MR, Yan B, MacGregor RM (2004) A subscribable peer-to-peer RDF repository for distributed metadata management. J Web Semant Sci Serv Agents World Wide Web 2(2):109–130
14. Chong EI, Das S, Eadon G, Srinivasan J (2005) An efficient SQL-based RDF querying scheme. In: Proceedings of the 31st very large data bases conference (VLDB 2005), Trondheim, Norway
15. Dean J, Ghemawat S (2004) Mapreduce: simplified data processing on large clusters. In: Proceedings of the USENIX symposium on operating systems design and implementation (OSDI), pp 137–147
16. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W (2007) Dynamo: Amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles, SOSP '07, pp 205–220
17. Erling O, Mikhailov I (2009) RDF support in the virtuoso DBMS. In: Networked knowledge–networked media, pp 7–24
18. Fang Q, Zhao Y, Yang G, Zheng W (2008) Scalable distributed ontology reasoning using DHT-based partitioning. In: ASWC '08: Proceedings of the 3rd Asian semantic web conference on the semantic web. Springer, Berlin, pp 91–105
19. Fernandez JD, Martinez-Prieto MA, Gutierrez C, Polleres A (2011) Binary RDF representation for publication and exchange (HDT). http://www.w3.org/Submission/2011/SUBM-HDT-20110330/
20. Goodman E, Mizell D (2010) Scalable in-memory RDFS closure on billions of triples. In: Proceedings of the 4th international workshop on scalable semantic web knowledge base systems, Shanghai, China
21. Goodman EL, Jimenez E, Mizell D, al Saffar S, Adolf B, Haglin D (2011) High-performance Computing Applied to Semantic Databases. In: Proceedings of the 8th extended semantic web conference (ESWC 2011), Crete, Greece
22. Guo Y, Pan Z, Heflin J (2005) LUBM: a benchmark for OWL knowledge base systems. J Web Semant 3(2–3):158–182. http://dblp.uni-trier.de/db/journals/ws/ws3.html#GuoPH05
23. Halevy AY, Ives ZG, Mork P, Tatarinov I (2003) Piazza: data management infrastructure for semantic web applications. In: Proceedings of the 12th international conference on world wide web, Budapest, Hungary, WWW '03, pp 556–567
24. Harris S, Gibbins N (2003) 3Store: efficient bulk RDF storage. In: Proceedings of the 1st international workshop on practical and scalable semantic systems(PSSS 2003), Sanibel Island. Florida, USA
25. Harris S, Lamb N, Shadbolt N (2009) 4store: the design and implementation of a clustered RDF store. In: 5th international workshop on scalable semantic web knowledge base systems (SSWS2009), Washington DC, USA
26. Hayes P (2004) RDF semantics. W3C Recommendation. http://www.w3.org/TR/rdf-mt/
27. Heine F, Hovestadt M, Kao O (2005) Processing complex RDF queries over P2P networks. In: Proceedings of workshop on information retrieval in peer-to-peer-networks (P2PIR 2005), Bremen, Germany
28. Hogan A, Pan JZ, Polleres A, Decker S (2010) SAOR: template rule optimisations for distributed reasoning over 1 billion linked data triples. In: Proceedings of the 9th international semantic web conference (ISWC 2010), Shanghai, China
29. ter Horst HJ (2005) Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Web Semant 3(2–3):79–115. http://dx.doi.org/10.1016/j.websem.2005.06.001
30. Ianni G, Krennwallner T, Martello A, Polleres A (2009a) A rule system for querying persistent RDFS data. In: Proceedings of the 6th European semantic web conference (ESWC 2009), Heraklion, Greece (Demo paper)
31. Ianni G, Krennwallner T, Martello A, Polleres A (2009b) Dynamic querying of mass-storage RDF data with rule-based entailment regimes. In: Proceedings of the 8th international semantic web conference (ISWC 2009), Washington DC, USA
32. Jagadish HV, Agrawal R, Ness L (1987) A study of transitive closure as a recursion mechanism. SIGMOD Rec 16(3):331–344. http://doi.acm.org/10.1145/38714.38750
33. Kaoudi Z, Miliaraki I, Magiridou M, Liarou E, Idreos S, Koubarakis M (2006) Semantic grid resource discovery in atlas. In: Talia D, Bilas A, Dikaiakos MD (eds) Knowledge and data management in grids, Springer, Berlin
34. Kaoudi Z, Koubarakis M, Kyzirakos K, Magiridou M, Miliaraki I, Papadakis-Pesaresi A (2007) Publishing, discovering and updating semantic grid resources using DHTs. In: CoreGRID workshop on grid programming model, grid and P2P systems architecture, grid systems, tools and environments, Heraklion, Crete, Greece
35. Kaoudi Z, Miliaraki I, Koubarakis M (2008) RDFS reasoning and query answering on top of DHTs. In: Proceedings of the 7th international conference on the semantic web (ISWC 2008), Karlsruhe, Germany
36. Kaoudi Z, Koubarakis M, Kyzirakos K, Miliaraki I, Magiridou M, Papadakis-Pesaresi A (2010a) Atlas: storing, updating and querying RDF(S) data on top of DHTs. J Web Semant
37. Kaoudi Z, Kyzirakos K, Koubarakis M (2010b) SPARQL Query optimization on top of DHTs. In: Proceedings of the 9th interna-

tional conference on the semantic web (ISWC 2010), Shanghai, China
38. Karger DR, Ruhl M (2004) Simple efficient load balancing algorithms for peer-to-peer systems. In: Proceedings of the 16th ACM symposium on parallelism in algorithms and architectures (SPAA 2004), Barcelona, Spain
39. Karnstedt M, Sattler KU, Hauswirth M, Schmidt R (2008) A DHT-based infrastructure for Ad-hoc integration and querying of semantic data. In: Proceedings of IDEAS'08, Coimbra, Portugal
40. Karvounarakis G, Alexaki S, Christophides V, Plexousakis D, Scholl M (2002) RQL: a declarative query language for RDF. In: Proceedings of the 11th world wide web conference (WWW 2002), Honolulu, Hawaii, USA
41. Kobilarov G, Scott T, Raimond Y, Oliver S, Sizemore C, Smethurst M, Bizer C, Lee R (2009) Media meets semantic web—how the BBC uses DBpedia and linked data to make connections. In: Proceedings of the 6th European semantic web conference (ESWC), Heraklion, Crete, Greece
42. Kokkinidis G, Sidirourgos L, Christophides V (2006) Query processing in RDF/S-based P2P database systems. In: Semantic web and peer-to-peer. Springer, Berlin
43. Kotoulas S, Oren E, van Harmelen F (2010) Mind the data skew: distributed inferencing by speeddating in elastic regions. In: Proceedings of the WWW 2010, Raleigh NC, USA
44. Leblay J (2012) SPARQL query answering with bitmap indexes. In: Proceedings of the 4th international workshop on semantic web information management (SWIM 2012) Scottsdale. AZ, USA
45. Liarou E, Idreos S, Koubarakis M (2006) Evaluating conjunctive triple pattern queries over large structured overlay networks. In: Proceedings of 5th the international semantic web conference (ISWC (2006) Athens, GA, USA
46. Lohrmann B, Battré D, Kao O (2009) Towards parallel processing of RDF queries in DHTs. In: Proceedings of the 2nd international conference on data management in grid and peer-to-peer systems, Linz, Austria
47. Loo BT, Hellerstein JM, Stoica I, Ramakrishnan R (2005) Declarative routing: extensible routing with declarative queries. In: Proceedings of the 2005 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '05). ACM, New York, pp 289–300. http://doi.acm.org/10.1145/1080091.1080126
48. Loo BT, Condie T, Garofalakis M, Gay DE, Hellerstein JM, Maniatis P, Ramakrishnan R, Roscoe T, Stoica I (2006) Declarative networking: language, execution and optimization. In: Proceedings of the 2006 ACM SIGMOD international conference on management of data (SIGMOD '06). ACM, New York, pp 97–108. http://doi.acm.org/10.1145/1142473.1142485
49. Lynch NA (1996) Distributed algorithms. Morgan Kaufmann Publishers Inc., San Francisco
50. Manola F, Miller E (2004) RDF primer. W3C Recommendation. http://www.w3.org/TR/rdf-mt/
51. Muñoz S, Pérez J, Gutierrez C (2009) Simple and efficient minimal RDFS. Web Semant Sci Serv Agents World Wide Web 7(3):220–234. http://dx.doi.org/10.1016/j.websem.2009.07.003
52. Munoz S, Perez J, Gutierrez C (2007) Minimal deductive systems for rdf. In: Proceedings of the 4th European semantic web conferene (ESWC 2007), pp 53–67. http://www.informatik.uni-trier.de/ley/db/conf/esws/eswc2007
53. Mutharaju R, Maier F, Hitzler P (2010) A MapReduce Algorithm for EL+. In: Proceedings of the 23rd international workshop on description logics (DL2010), Waterloo, Canada
54. Neumann T, Weikum G (2008) RDF-3X: a RISC-style engine for RDF. In: Proceedings of 34th international conference on very large data bases (VLDB 2008), Auckland, New Zealand, vol 1, pp 647–659
55. Oren E, Kotoulas S, Anadiotis G, Siebes R, ten Teije A, van Harmelen F (2009a) MARVIN: a platform for large-scale analysis of semantic web data. In: Proceedings of web science conference
56. Oren E, Kotoulas S, Anadiotis G, Siebes R, ten Teije A, van Harmelen F (2009b) Marvin: distributed reasoning over large-scale semantic web data. Web Semant Sci Serv Agents World Wide Web 7(4):305–316
57. Pitoura T, Triantafillou P (2007) Load distribution fairness in P2P data management systems. In: Proceedings of the 21st interntational conference on data engineering (ICDE 2007), Tokyo, Japan
58. Prud'hommeaux E, Seaborn A (2005) SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/
59. Rhea S, Geels D, Roscoe T, Kubiatowicz J (2004) Handling churn in a DHT. In: USENIX annual technical conference
60. Rowstron A, Druschel P (2001) (2001) Pastry: scalable. Distributed object location and routing for large-scale-peer-to-peer storage utility. In: Middleware 2001
61. Salvadores M, Correndo G, Omitola T, Gibbins N, Harris S, Shadbolt N (2010) 4s-reasoner: RDFS backward chained reasoning support in 4store. In: Web-scale knowledge representation, retrieval, and reasoning (Web-KR3), Toronto, Canada
62. Salvadores M, Correndo G, Harris S, Gibbins N, Shadbolt N (2011) The design and implementation of minimal RDFS backward reasoning in 4store. In: Proceedings of the 8th extended semantic web conference (ESWC 2011), Crete, Greece, pp 139–153
63. SHA-1 (1995) Secure hash standard. Publication 180-1
64. Soma R, Prasanna VK (2008) Parallel inferencing for OWL knowledge bases. In: ICPP '08: Proceedings of the (2008) 37th international conference on parallel processing. IEEE Computer Society, Washington, DC, USA, pp 75–82
65. Staudt M, Jarke M (1996) Incremental maintenance of externally materialized views. In: Proceedings of 22th international conference on very large data bases (VLDB 1996), Mumbai (Bombay), India, pp 75–86
66. Stocker M, Seaborne A, Bernstein A, Kiefer C, Reynolds D (2008) SPARQL basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th international world wide web conference (WWW 2008), Beijing, China
67. Stoica I, Morris R, Liben-Nowell D, Karger D, Kaashoek MF, Dabek F, Balakrishnan H (2003) Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans Netw 11(1):17–32
68. Stuckenschmidt H, Broekstra J, (2005) Time-space trade-offs in scaling up RDF schema reasoning. In: Proceedings of web information systems engineering workshop (WISE, 2005) New York, NY, USA
69. Surana S, Godfrey B, Lakshminarayanan K, Karp R, Stoica I (2006) Load balancing in dynamic structured peer-to-peer systems. Perform Eval 63(3):217–240. http://dx.doi.org/10.1016/j.peva.2005.01.003
70. Theoharis Y, Christophides V, Karvounarakis G (2005) Benchmarking database representations of RDF/S stores. In: Proceedings of the 4th international semantic web conference (ISWC 2005), Galway, Ireland
71. Ullman JD (1988) Principles of database and knowledge-base systems, Vol I. Computer Science Press, Rockville
72. Urbani J, Kotoulas S, Oren E, van Harmelen F (2009) Scalable distributed reasoning using MapReduce. In: Proceedings of the 8th international semantic web conference (ISWC2009)
73. Urbani J, Kotoulas S, Maassen J, Harmelen FV, Bal H (2010) OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In: Proceedings of the 8th extended semantic web conference (ESWC2010), Heraklion, Greece
74. Urbani J, van Harmelen F, Schlobach S, Bal H (2011) QueryPIE: backward reasoning for OWL horst over very large knowledge

bases. In: Proceedings of the 10th international semantic web conference (ISWC 2011), Bonn, Germany
75. Weaver J, Hendler J (2009) Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In: 8th international semantic web conference (ISWC2009)
76. Wilkinson K, Sayers C, Kuno HA, Raynolds D (2003) Efficient RDF storage and retrievalin Jena2. In: Proceedings of the 1st international workshop on semantic web and databases (SWDB 2003, co-located with VLDB 2003), Berlin, Germany
77. Williams GT, Weaver J, Atre M, Hendler JA (2010) Scalable reduction of large datasets to interesting subsets. J Web Semant 8(4):365–373