Atlas: Storing, Updating and Querying RDF(S) Data on Top of DHTs [1]

Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Iris Miliaraki [2], Matoula Magiridou, Antonios Papadakis-Pesaresi

Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Athens, Greece

Abstract

The RDF(S) data model has been proposed for encoding metadata about Web resources. As more and more Web resources are annotated using RDF(S), there is an urgent need for efficiently dealing with this large volume of data. In this paper, we present Atlas, a peer-to-peer system for storing, updating and querying RDF(S) data. The Atlas system has been built using the distributed hash table Bamboo. Atlas was developed in the context of project OntoGrid, where it was used as a distributed repository for RDF(S) metadata describing Grid services and resources. The development of Atlas continues in other projects that our group participates currently. This paper gives an overview of the most recent version of Atlas and discusses a representative application.

Key words: Semantic web, RDF(S), peer-to-peer networks, distributed hash tables, query processing

1. Introduction

With advances in the Semantic Web, Web 2.0, E-science data repositories and the Linked Data initiative, more and more resources are becoming semantically annotated with RDF causing huge amounts of RDF metadata to be generated. Managing this avalanche of RDF metadata is becoming a challenge for existing RDF stores such as Sesame, Jena, Virtuoso, RSSDB, Oracle 11g RDF and others. This trend has necessitated the careful performance evaluation of existing RDF stores on appropriately designed benchmarks and very big data sets [15,38,8] and the development of novel implementations based on efficient indexing techniques and relational-style statistics-based query optimization [30,44,31]. Performance results published very recently indicate that state-of-the-art systems like RDF-3X [31] can execute complex join queries on RDF data sets containing close to a billion triples in a few seconds.

Although some existing RDF stores have excellent performance, they can be overwhelmed by user requests when used in wide-area network applications such as content-sharing, Web/Grid service registries, distributed digital libraries, social networks etc. Example of such systems are presented in [12,37,19,41,40,27] [3]. More generally, since cen-

Email addresses: zoi@di.uoa.gr (Zoi Kaoudi), koubarak@di.uoa.gr (Manolis Koubarakis), kkyzir@di.uoa.gr (Kostis Kyzirakos), iris@di.uoa.gr (Iris Miliaraki), matoula@di.uoa.gr (Matoula Magiridou), apapadak@di.uoa.gr (Antonios Papadakis-Pesaresi).

[1] This work was partially funded by FP6/IST project OntoGrid, project "Peer-to-Peer Techniques for Semantic Web Services" funded from the Greek General Secretariat for Research and Technology and FP7/ICT project SemsorGrid4Env.

[2] Supported by Microsoft Research through its European PhD Scholarship Programme.

[3] Although RDF is the chosen metadata model for resource annotation only in [12,19], we can easily see how it could

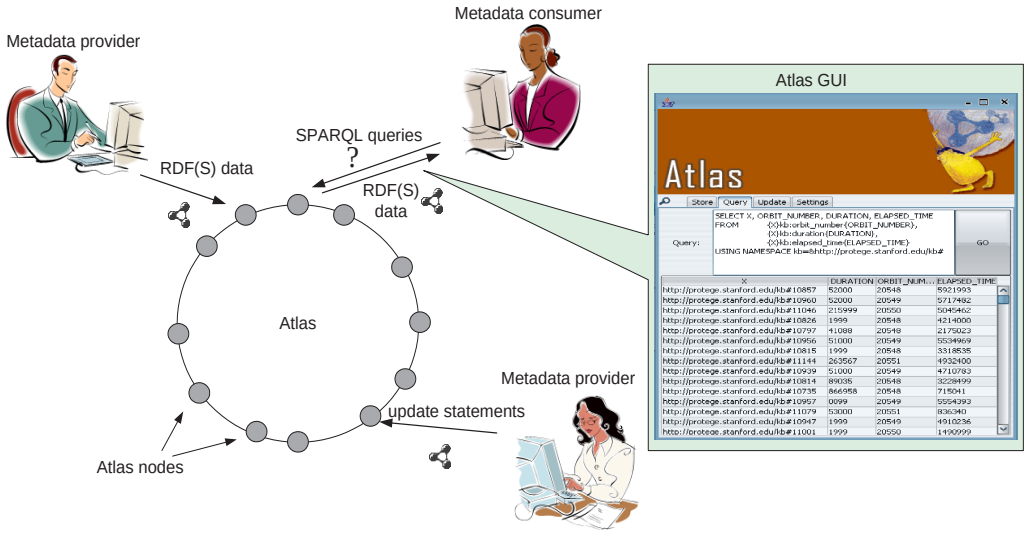


Fig. 1. Using Atlas for storing, updating and querying RDF(S) data

tralized RDF stores are lacking the reliability properties typically associated with large distributed systems [6] (e.g., fault-tolerance, load balancing, availability etc.), researchers have also studied parallel and distributed solutions for RDF(S) reasoning and query processing. Recent proposals in this area have been based on distributed hash tables (DHTs) [9,24,4,5,19,18], distributed computing platforms based on powerful clusters [32,33,16,14] and cloud computing platforms using MapReduce [28].

In this paper we present Atlas, a system built on top of the Bamboo DHT [35] for the distributed storage, update and querying of RDF data and RDFS ontologies. The implementation of Atlas started in project OntoGrid [4] where it has been used for implementing a Semantic Grid service registry [22]. The Atlas prototype developed in OntoGrid has been further refined in the Greek project "Peer-to-Peer Techniques for Semantic Web Services". In this project Atlas formed the core of a distributed service matchmaker developed for a subset of OWL-S [3]. Currently, Atlas is being used in project SemsorGrid4Env [5] where it is extended with the capability to store temporal and spatial metadata expressed in an appropriate extension of RDF that our group has defined [21]. Atlas is publicly available as open source under the LGPL license [6].

Research in Atlas has resulted in novel distributed RDFS reasoning algorithms [18], and efficient strategies for RDF query processing on top of DHTs which cover *one-time* [24,19] and *continuous* queries [23]. We call one-time queries the standard queries asked in a typical RDF store (or a typical DBMS in general). The answer to a one-time query is returned to the user once, immediately after the query is executed. In contrast, continuous queries are long standing and have a new answer every time a relevant update is executed (e.g., "notify me whenever a new version of the Atlas system becomes available"). The query language of the newest version of Atlas (v1) is SPARQL [13] [7]. Atlas has been tested successfully on wide-area networks of many nodes including local clusters and PlanetLab [8]. In this paper we only discuss the one-time query functionality of Atlas since this functionality has been tested more heavily in the projects mentioned above and in various performance experiments we have carried out.

The rest of the paper is organized as follows. Section 2 gives a short introduction to the Bamboo DHT, the RDF(S) data model and the query language supported by Atlas. In Section 3, we describe the architecture of Atlas and briefly refer to the algorithms for RDF(S) storage, querying and updating. We also present the Atlas API. Further, in Section 4, we describe the use of Atlas in a distributed digital library application and in Section 5 we present

have been used in [37,41,40,27]. See also Section 4.

[4] `http://www.ontogrid.net`

[5] `http://www.semsorgrid4env.eu/`

[6] `http://atlas.di.uoa.gr`

[7] Previous versions of Atlas supported RQL [20].

[8] `http://www.planet-lab.org/`

a brief experimental evaluation. Finally, Section 6 concludes the paper. This paper gives a system-level description of Atlas. Details of algorithms and other results presented only briefly can be found in our publications [18,24].

2. Background

In this section, we give a very short introduction to the Bamboo DHT, the RDF(S) data model and the SPARQL query language.

Bamboo [1,35] is a DHT based on Pastry [36], from where it inherits the circular identifier space and the routing protocols. Bamboo improves on Pastry by being able to withstand very dynamic changes to the network [35]. Like most implementations of DHTs, Bamboo offers a simple interface consisting of two operations: `put(ID, item)` and `get(ID)`. The `put` operation inserts an item with key `ID` and value `item` in the DHT. The `get` operation returns the value `item` from the DHT node responsible for key `ID`. In Bamboo, a `remove(ID, secret)` operation is also provided. This operation removes the item with key `ID`. The parameter `secret` is used for authorizations purposes. For this reason, Bamboo also offers an extended `put(ID, item, secret)` operation where one can define a secret value to ensure that the item can be removed only by an authorized user.

The data model supported by Atlas is RDF(S). Resources are described by RDF(S) documents written either in the Notation3 (N3) format or the RDF/XML format. Each resource description can be decomposed into a collection of triples. Each triple has the form $(subject, predicate, object)$ and represents a relationship, denoted by the predicate, that holds between the subject and the object. The subject and the predicate in a triple are URIs and the object is either a URI or a literal.

The query language offered to users of Atlas is SPARQL [13]. The syntax of SPARQL is SQL-like (`SELECT-FROM-WHERE` queries) and its most basic construct is that of triple pattern. A triple pattern has the form $(subject, predicate, object)$ where the subject and predicate can be either URIs or variables and the object can be either a URI, a literal or a variable. Atlas supports the subset of SPARQL which consists of conjunctions of triple patterns (i.e., basic graph patterns in the SPARQL terminology).

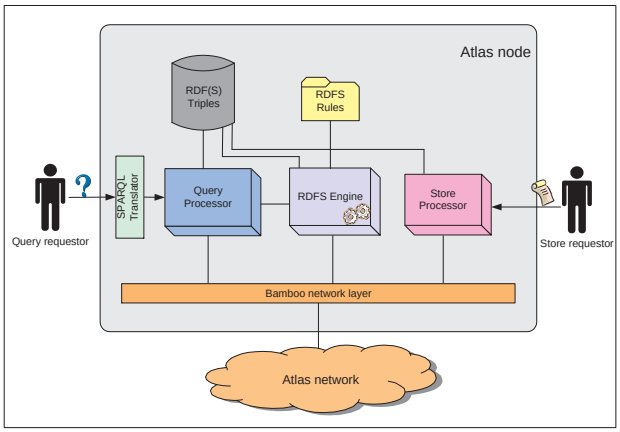


Fig. 2. Architecture of an Atlas node

3. Atlas in detail

In this section we present a simple scenario demonstrating how Atlas can be used, the Atlas architecture and API, and sketch the algorithms for storage, querying and updating RDF(S) data.

In the application scenarios supported by Atlas and RDF(S) stores in general, we can identify two parties, the *metadata provider* and the *metadata consumer* (Figure 1). The metadata provider provides resource descriptions by submitting RDF(S) data to the system. The metadata consumer wants to discover resources by submitting SPARQL queries to the system. The process of annotating and querying Web resources is by itself distributed and therefore fits well with the P2P paradigm of Atlas. A screenshot of the graphical user interface (GUI) supported by Atlas is also depicted in Figure 1.

System Architecture. Atlas inherits the programming architecture used in the implementation of Bamboo which is called SEDA (Staged Event-Driven Architecture)[45]. When using SEDA, the functional units of a program are represented as separate stages. Atlas inherits some basic stages from Bamboo and has its own stages for implementing additional functionalities. In the current implementation of the Atlas system, several stages have been designed so that the functionalities of publishing RDF(S) descriptions, updating an RDF(S) description and posing a one-time query to an Atlas node are supported.

Figure 2 depicts the architecture of an Atlas node. We distinguish between the following components:

– the *Bamboo network layer*, which is responsible

for routing and handling network messages,
- the *local database*, which is where the triples are stored locally at each node,
- the *SPARQL translator*, which is responsible for parsing SPARQL queries and transforming them to the equivalent internal representation handled by the *query processor*,
- the *query processor*, which is responsible for the distributed evaluation of one-time queries,
- the *RDFS reasoning engine*, which is responsible for computing the appropriate inferences in order to give the answer to the queries taking into account the RDFS entailment rules, and
- the *update processor*, which is responsible for storing and updating RDF documents, and executing atomic updates.

Storing and Updating RDF Data. When a node wants to store RDF(S) data, it submits it in the form of an RDF(S) document. The document can be in either RDF/XML or RDF/N3 format. This document is decomposed into a collection of RDF triples. Then, each triple of form $(subject, predicate, object)$ is stored in Atlas three times, once for its subject, once for its predicate and once for its object. This *triple-indexing* approach was originally introduced in [10]. Each of these storage operations is implemented using the underlying `put` operation provided by the Bamboo DHT where the key is the subject, predicate or object value respectively and the item is the triple itself.

As URIs and literals may consist of long strings, we have implemented a mapping dictionary similarly to centralized RDF stores [30,11]. URIs and literals are mapped to integer identifiers and then, triple storage and query evaluation is performed using these identifiers.

Each node stores the triples it receives in its *local database*. We have used SQLite [9] for this purpose. SQLite is a software library which can be intergraded in an application program and used as a light-weight relational DBMS [10]. A database relation with three columns is built when a node joins the network for the first time. The three columns correspond to the subject, predicate and object of the triples stored. When a triple arrives at a node, it is stored as a tuple in this relation.

[9] `http://www.sqlite.org/`

[10] In previous versions of Atlas, the Berkeley DB database [2] 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 a lightweight relational database instead.

Atlas also supports updating resource descriptions using atomic update operations. Atomic updates can be *insertions*, *deletions* and *replacements* of a single triple. An example of such an update statement might be *"Modify the affiliation of Zoi Kaoudi from National Technical University of Athens to National and Kapodistrian University of Athens"*. Our current implementation of atomic updates relies directly on the underlying `put` and `remove` operations provided by the Bamboo API. More details about the implementation of updates can be found in [17]. Set-oriented updates (i.e., operations where a set of triples is updated) as expressed in languages SPARUL [39] and RUL [26] have not been implemented in Atlas so far, since supporting more complex updates is currently an open problem in DHTs in general.

Query Processing. Let us now discuss some algorithms for one-time query processing in Atlas. As an example, consider the query *"Select all articles that are published in WWW 2009 and contain keyword RDF"*. This query can be expressed in SPARQL as follows:

```
PREFIX swrc:<http://swrc.ontoware.org/ontology#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX confwww:<http://data.semanticweb.org/conference/www/>
SELECT ?articleTitle
WHERE { ?article rdf:type swrc:InProceedings .
?article swrc:title ?articleTitle .
?article swrc:keyword "RDF" .
?article swrc:presentedIn confwww:2009 . }
```

This query will be processed by Atlas as follows. The node that posed the query is responsible for parsing it and producing an equivalent internal representation using the local SPARQL translator. Then, the Atlas nodes cooperate to find the RDF(S) data that satisfy the query. We have currently implemented two algorithms for query processing: Query Chain (QC) and Spread By Value (SBV), originally proposed in [24].

The main characteristic of QC is that the query is evaluated by a chain of nodes. Intermediate results flow through the nodes of this chain and finally the last node in the chain delivers the result back to the node that submitted the query.

SBV extends the ideas of QC to achieve a better distribution of the query processing load. It does not create a single chain for a query as QC does, but by exploiting the values of matching triples found while processing the query incrementally, it rewrites the query and distributes the responsibility of evaluat-

ing it to more nodes than QC (in other words, SBV is constructing multiple chains for each query). For more details about algorithms QC and SBV, the interested reader might refer to [24]. More recently, QC and SBV have been supplemented with query optimization techniques that have substantially improved the performance of Atlas.

RDFS Reasoning. We have implemented and evaluated forward and backward chaining algorithms for RDFS reasoning in Atlas [18]. While the general idea of forward chaining is that all inferred triples are precomputed and stored in the network a priori, backward chaining is a goal-driven algorithm starting from a given request and deriving all possible answers in a top down way. RDFS reasoning in Atlas is based on the *ρdf* fragment of the RDFS vocabulary studied formally in [29]. Each Atlas node keeps locally a copy of the relevant RDFS entailment rules [29] in a datalog-like notation and uses it in a forward or backward manner.

In the forward chaining algorithm, each time a node receives a triple to be stored in its local database, it computes all inferred triples and sends them to the network to be stored to the appropriate nodes. The indexing scheme we use assures the soundness and completeness of the algorithm. Then, answering a query can be done using the algorithms QC or SBV presented above.

In contrast to the data driven nature of forward chaining, backward chaining starts from the given query and tries to find rules that can be used to derive answers. Thus, each time a node receives a request for evaluating a query, it should also use the RDFS entailment rules to compute the answer. We utilize and extend the rule adornment concept from recursive query processing [42] to exploit the distributed nature of the DHT and show that the algorithm is sound and complete. The interested reader might refer to [18] for more details on the RDFS reasoning algorithms of Atlas. The backward chaining algorithm of [18] has been integrated with QC and SBV so that SPARQL queries are evaluated under the RDFS entailment regime [7].

Atlas API. We have developed an API for storing RDF(S) data in Atlas and posing SPARQL queries and atomic updates to an Atlas node. The following functions are provided by the Atlas API:

- `store(file)`, stores an RDF/XML or RDF/N3 `file` in the Atlas network. Each RDF(S) document is decomposed into a collection of RDF(S) triples and the indexing scheme described previously is used to store these triples in various nodes of the Atlas network.
- `store(file, secret)`, stores a `file` using the client-chosen `secret` value so that the triples derived from the specific file are removable. The secret value is used to provide a basic authorization mechanism. This authorization mechanism is inherited from Bamboo for supporting updates [1]. A triple may be removed or updated from the client that inserted it, since it is the only one to know the secret value.
- `store_ontology(url)`, stores an RDFS ontology in the Atlas network specified by `url`. Depending on the reasoning scheme supported, the appropriate algorithms described in [18] are used.
- `query(string)`, poses an one-time SPARQL query to the Atlas network. Each time a node poses a one-time query, the network nodes cooperate to find RDF(S) data that form the answer to the query using the algorithms found in [24,18].
- `update(string, secret)`, poses an atomic insert or delete to an Atlas node.
- `replace(string, old_secret, new_secret)`, replaces a triple that was previously stored to the Atlas network with a new one. The client should also provide the secret value (`old_secret`) that was used when the triple to be removed was inserted, and a new one (`new_secret`) for the triple to be inserted.

4. Applications

We can think of several application scenarios which feature Atlas as a distributed repository for storing and retrieving metadata describing resources: content-sharing, Web/Grid service registries, distributed digital libraries and social networks. To demonstrate how Atlas can be used in such application scenarios, we consider the case of distributed digital libraries (DDLs) that our group has previously studied in [41] using P2P systems based on data models from the area of Information Retrieval. The use of Atlas to implement a distributed Grid service registry in the context of project OntoGrid has already been discussed in [19].

According to [41], there are two kinds of basic functionality that one can expect by a DDL: *information retrieval* and *publish/subscribe*. In the information retrieval scenario, a user poses a query (e.g., "I am interested in papers on the Semantic Web") and the DDL returns matching resources. In the publish/subscribe scenario, a user posts a *sub-*

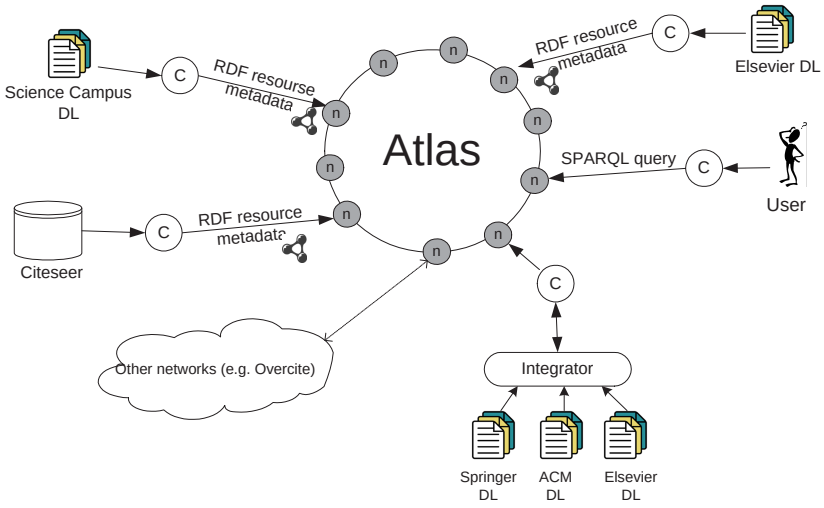


Fig. 3. Using Atlas to implement a distributed digital library

scription to the DDL to receive notifications whenever certain events of interest take place (e.g., when a paper on the Semantic Web becomes available). We concentrate on the information retrieval scenario only; the publish/subscribe scenario can be realized similarly using the continuous query functionality of Atlas that we have not discussed in this paper.

The main components of the DDL architecture shown in Figure 3 are: *Atlas nodes* (denoted by n), *Atlas clients* (denoted by c), *providers* and *consumers*. Providers own information sources that are part of the DDL, while consumers are interested in the DDL content. Atlas nodes form an overlay network that offers a robust, fault-tolerant and scalable means for routing messages and managing DDL resource meta-data encoded in RDF. Atlas clients are software components that enable providers and consumers to communicate with the Atlas network. Clients can connect to any Atlas node of the network. [11]

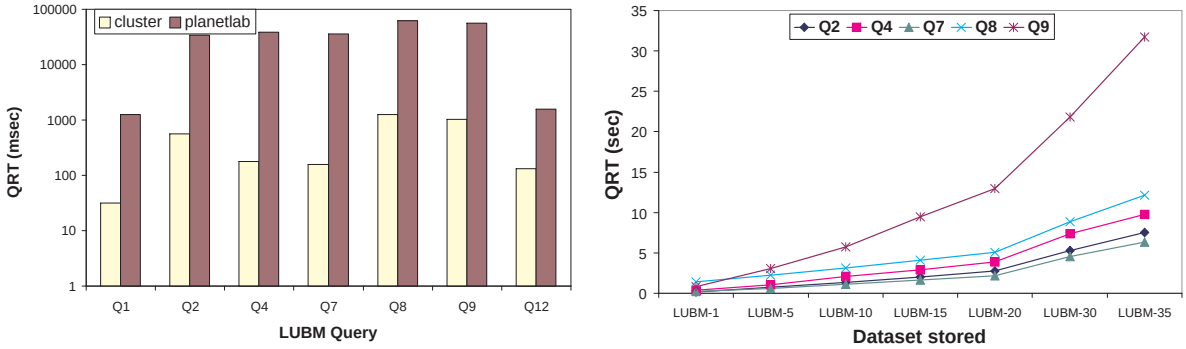
Figure 3 gives a realistic use case of this architecture. We assume a university with 3 geographically distributed campuses (e.g., Arts, Sciences and Medicine) and a local DL in each campus (only the Science DL is shown graphically in the figure). Each local DL is a provider in terms of our architecture. Providers maintain an Atlas client which offers them an access point to the Atlas network. Providers use this client to publish RDF metadata about the resources they hold in the Atlas network. Consumers (e.g., users) can also deploy an Atlas client to pose queries about available resources. Various interesting extensions to this basic setting are possible as shown in Figure 3. For example, the university might be interested in making available to its students and staff, in a timely way, the content provided by other publishers (e.g., CiteSeer, ACM, Springer, Elsevier). Figure 3 shows how our architecture can be used to fulfill this requirement. A client is used to publish in the overlay network metadata about resources available in CiteSeer. The content of the ACM, Springer and Elsevier DLs is similarly made available. In this case, an integration layer is used to unify different DLs. We omit discussion of the relevant interoperability issues since they can be solved at the level of providers with well-known integration techniques. We also leave open the question of how to interoperate with other relevant P2P networks e.g., the DHT-based digital library OverCite [40].

We close by pointing out that P2P architectures of digital libraries have been very popular lately starting with [41]. See for example, [34,43,25].

[11] This architecture differs from the one in [41] in two ways. Firstly, in [41], super-peers are used to form the overlay network. Secondly, two assymetric kinds of clients are used to capture the behaviour of providers vs. consumers. Here we have only one type of node (client) that offers both functionalities.

5. Experiments

In this section, we present a brief experimental evaluation of the Atlas system. In the latest version of Atlas, algorithms QC and SBV have been sup-



(a) Cluster vs. PlanetLab performance

(b) Varying dataset size in cluster

Fig. 4. Atlas performance

plemented with query optimization techniques that have substantially improved the performance of the system. Below we present some of the results of algorithm QC using the mapping dictionary.

We have done experiments in PlanetLab [12] as well as in the more controlled environment of a local cluster [13] . The cluster consists of 30 machines where we run 4 Atlas nodes in each machine, i.e., 120 Atlas nodes in the network. In PlanetLab, we created a network of 281 Atlas nodes in 281 machines that were available and lightly loaded at the time of the experiment. For our evaluation we used the LUBM benchmark [15] that provides RDF datasets of arbitrary sizes and 14 SPARQL queries. In the following, we show our results using the most representative LUBM queries. The results are similar for the rest of the queries. All measurements are averaged over 10 runs.

In Figure 4(a), we show the time required to answer a query (query response time) in the local cluster and in PlanetLab. The x-axis shows the different LUBM queries, while the y-axis shows the query response time (QRT) in msec on a logarithmic scale. Due to the long response times we observed in PlanetLab, Figure 4(a) shows the results for the dataset LUBM-1 which consists of only 144, 819 triples. As expected, the performance of Atlas in the cluster is substantially better than in PlanetLab. This results from the availability of the cluster machines as well as the huge bandwidth provided by a local cluster. For example, the query response time of query $Q2$ is 0.569 seconds in the cluster, while it is 33.9 seconds in PlanetLab. The bad response times in PlanetLab are caused by the heavy load of PlanetLab nodes and bandwidth congestion. PlanetLab nodes are used simultaneously by many users which share the nodes' resources. This very often results in overloaded nodes and bandwidth congestion. In addition, PlanetLab nodes are spread across four continents and hence bandwidth is lower than the bandwidth provided by a local cluster. However, PlanetLab offers real-world testing of distributed systems thus it has been a "must" in our testing of Atlas.

Let us now briefly explain the differences in the query response time of the queries. A significant factor that affects the query response time of a query is the number of triple patterns it contains; the more triple patterns the query has, the more join operations are required and the more messages are sent during the query evaluation. Query $Q1$ contains only 2 triple patterns, queries $Q4$ and $Q8$ contain 5 triple patterns, queries $Q7$ and $Q12$ contain 4 triple patterns, and queries $Q2$ and $Q9$ contain 6 triple patterns. Queries $Q1$ and $Q12$ have a significant smaller query response time in both settings compared to the other queries. $Q1$ consists only of two triple patterns and thus only one join operation is required, while query $Q12$ returns an empty result set because of its first triple pattern which does not match any triples. On the contrary, query $Q9$, which consists of 6 triple patterns, produces a large intermediate result set twice. As a result, it requires computing two expensive join operations and transferring two large messages through the network. This makes query $Q9$ more expensive to evaluate than the other queries, as we will also see below.

Figure 4(b) shows the behavior of our system in the cluster as the dataset stored in the network grows. In a network of 120 peers, we stored the datasets LUBM-1 (144, 819 triples),

[12] http://www.planet-lab.org/
[13] http://www.grid.tuc.gr/

LUBM-5 (887,461 triples), LUBM-10 (1,806,023 triples), LUBM-15 (2,771,847 triples), LUBM-20 (3,812,865 triples), LUBM-30 (5,629,144 triples) and LUBM-35 (6,586,652 triples). Each time, we measured the query response time of LUBM queries consisting of more than 4 triple patterns. We observe from Figure 4(b) that the query response time increases as the number of triples stored in the network grows. This is caused by two factors. Firstly, the local database of each peer grows and as a result local query processing becomes more time-consuming. Secondly, the result sets of some of the queries vary as the dataset changes. For example, for LUBM-1 the result set of Q9 consists of 134 answers, while for LUBM-35 the result set of Q9 contains 5989 answers. This results in transferring larger intermediate result sets through the network which also affects the query response time of the query.

Other experiments done in PlanetLab regarding our RDFS reasoning algorithms can be found in [18], while simulation results comparing algorithms QC and SBV can be found in [24]. Ongoing experimental evaluation of Atlas v1 running the QC algorithm shows that Atlas can answer queries involving millions of RDF triples efficiently, but cannot yet scale to billions of triples as e.g., YARS2 [16]. However, YARS2 and other approaches as [33,14] are not based on general DHT-based platforms that have been the focus of our work on Atlas. It is still an open question how to achieve greater scalability of RDF query processing in these platforms and thus, Atlas is the only first step towards this direction.

6. Conclusions

We presented Atlas, a distributed system for storing, updating and querying RDF(S) data on top of DHTs. We described briefly the algorithms implemented in Atlas and the architecture of each Atlas node. In addition, we presented an application where Atlas is used for building a distributed digital library. Finally, we presented a very brief experimental evaluation of Atlas.

References

[1] Bamboo DHT, `http://bamboo-dht.org/`.

[2] BerkeleyDB, `http://sleepycat.com/`.

[3] OWL-S: Semantic Markup for Web Services, `http://www.w3.org/Submission/OWL-S/`.

[4] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, T. V. Pelt, GridVine: Building Internet-Scale Semantic Overlay Networks, in: ISWC 2004.

[5] D. Battre, A. Hoing, F. Heine, O. Kao, On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores, in: DBISP2P 2006.

[6] K. P. Birman, Reliable Distributed Systems: Technologies, Web Services, and Applications, Springer Verlag, 2005.

[7] Birte Glimm and Bijan Parsia, SPARQL 1.1 Entailment Regimes, `http://www.w3.org/TR/2010/WD-sparql11-entailment-20100126/` (2010).

[8] C. Bizer, A. Schultz, Benchmarking the Performance of Storage Systems That Expose SPARQL Endpoints, in: SSWS 2008.

[9] M. Cai, M. Frank, RDFPeers: A Scalable Distributed RDF Repository Based on A Structured Peer-to-Peer Network, in: WWW 2004.

[10] M. Cai, M. R. Frank, B. Yan, R. M. MacGregor, A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management, Journal of Web Semantics: Science, Services and Agents on the World Wide Web 2 (2) (2004) 109–130.

[11] E. I. Chong, S. Das, G. Eadon, J. Srinivasan, An Efficient SQL-based RDF Querying Scheme, in: VLDB 2005.

[12] P. Cudré-Mauroux, A. Budura, M. Hauswirth, K. Aberer, PicShark: Mitigating Metadata Scarcity Through Large-Scale P2P Collaboration, The VLDB Journal 17 (6) (2008) 1371–1384.

[13] E. Prud'hommeaux and A. Seaborn, SPARQL query language for RDF, W3C Recommendation `http://www.w3.org/TR/rdf-sparql-query/` (2008).

[14] O. Erling, I. Mikhailov, Towards Web Scale RDF, in: 4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008), 2008.

[15] Y. Guo, Z. Pan, J. Heflin, LUBM: A Benchmark for OWL Knowledge Base Systems, J. Web Sem. 3 (2-3) (2005) 158–182.
URL `http://swat.cse.lehigh.edu/projects/lubm/`

[16] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A Federated Repository for Querying Graph Structured Data from the Web, in: ISWC/ASWC 2007.

[17] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, M. Magiridou, I. Miliaraki, A. Papadakis-Pesaresi, 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.

[18] Z. Kaoudi, I. Miliaraki, M. Koubarakis, RDFS Reasoning and Query Answering on Top of DHTs, in: ISWC 2008.

[19] Z. Kaoudi, I. Miliaraki, M. Magiridou, E. Liarou, S. Idreos, M. Koubarakis, Semantic Grid Resource Discovery in Atlas, Knowledge and Data Management in Grids (2006).

[20] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, RQL: A Declarative Query Language for RDF, in: WWW 2002.

[21] M. Koubarakis, K. Kyzirakos, Modeling and Querying Metadata in the Semantic Sensor Web: the Model stRDF and the Query Language stSPARQL, in: ESWC 2010.

[22] M. Koubarakis, I. Miliaraki, Z. Kaoudi, M. Magiridou, A. Papadakis-Pesaresi, Semantic Grid Resource

Discovery using DHTs in Atlas, in: 3rd GGF Semantic Grid Workshop 2006.
[23] E. Liarou, S. Idreos, M. Koubarakis, Continuous RDF Query Processing over Distributed Hash Tables, in: ISWC 2007.
[24] E. Liarou, S. Idreos, M. Koubarakis, Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks, in: ISWC 2006.
[25] J. Lu, J. Callan, Full-text Federated Search of Text-based Digital Libraries in Peer-to-Peer Networks, Inf. Retr. 9 (4) (2006) 477–498.
[26] M. Magiridou, S. Sahtouris, V. Christophides, M. Koubarakis, RUL: A Declarative Update Language for RDF, in: ISWC 2005.
[27] M. Mani, A.-M. Nguyen, N. Crespi, What's up 2.0: P2P Spontaneous Social Networking, in: IEEE INFOCOM 2009.
[28] P. Mika, G. Tummarello, Web Semantics in the Clouds, IEEE Intelligent Systems 23 (5) (2008) 82–87.
[29] S. Muñoz, J. Pérez, C. Gutierrez, Simple and Efficient Minimal RDFS, Web Semantics: Science, Services and Agents on the World Wide Web 7 (3) (2009) 220–234.
[30] T. Neumann, G. Weikum, RDF-3X: a RISC-style Engine for RDF, in: VLDB 2008.
[31] T. Neumann, G. Weikum, Scalable Join Processing on Very Large RDF Graphs, in: SIGMOD 2009.
[32] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, F. van Harmelen, MARVIN: A platform for large-scale analysis of Semantic Web data, in: WebSci'09.
[33] A. Owens, A. Seaborne, N. Gibbins, mc schraefel, Clustered TDB: A Clustered Triple Store for Jena, Technical Report (Unpublished) (2008).
[34] P. Raftopoulou, E. Petrakis, C. Tryfonopoulos, G. Weikum, Information Retrieval and Filtering over Self-Organising Digital Libraries, in: ECDL 2008.
[35] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, Handling Churn in a DHT, in: USENIX Annual Technical Conference 2004.
[36] A. Rowstron, P. Druschel, Pastry: Scalable, Distributed Object Location and Routing for Large-Scale- Peer-to-Peer Storage Utility, in: Middleware 2001.
[37] M. Schlosser, M. Sintek, S. Decker, W. Neijdl, A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services, in: 2nd IEEE International Conference on Peer-to-Peer Computing 2002.
[38] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, C. Pinkel, An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario, in: ISWC 2008.
[39] A. Seaborne, G. Manjunath, C. Bizer, J. Breslin, S. Das, I. Davis, S. Harris, K. Idehen, O. Corby, K. Kjernsmo, B. Nowack, SPARQL Update: A language for updating RDF graphs, W3C Member Submission `http://www.w3.org/Submission/SPARQL-Update/` (2008).
[40] J. Stribling, I. G. Councill, J. Li, M. F. Kaashoek, D. R. Karger, R. Morris, S. Shenker, OverCite: A Cooperative Digital Research Library, in: IPTPS, 2005.
[41] C. Tryfonopoulos, S. Idreos, M. Koubarakis, LibraRing: An Architecture for Distributed Digital Libraries Based on DHTs, in: ECDL, 2005.
[42] J. D. Ullman, Principles of Database and Knowledge-Base Systems, Volume I, Computer Science Press, 1988.
[43] S. Viglas, T. Dalamagas, V. Christophides, T. K. Sellis, A. Dimitriou, Application of the Peer-to-Peer Paradigm in Digital Libraries, in: DELOS Conference, 2007.
[44] C. Weiss, P. Karras, A. Bernstein, Hexastore: Sextuple Indexing for Semantic Web Data Management, in: VLDB 2008.
[45] M. Welsh, D. Culler, E. Brewer, SEDA: An Architecture for Well-Conditioned, Scalable Internet Services, in: SOSP 2001.