3rdf: Storing and Querying RDF Data on top of the 3nuts Overlay Network

Liaquat Ali, Thomas Janson, and Georg Lausen
University of Freiburg
Email: {ali, janson, lausen}@informatik.uni-freiburg.de

***Abstract*—In current research Peer-to-Peer (p2p) based Semantic Web systems mainly use distributed hash table (DHT) based networks. These networks provide good load balancing by applying uniform hash functions with the drawback that they destroy possible semantic relations between data elements. But mapping the data semantics on the network structure could improve the routing time in the network and consequently the RDF query latency on application layer. In this paper, we present *3rdf*, a distributed RDF system for storing and querying RDF data. The 3rdf system has been built on top of the 3nuts p2p network. The 3nuts network improves on reducing the query response time and bandwidth usage in our system by adapting the network structure to the semantics of the RDF data. In addition, we study how the evaluation of SPARQL BASIC graph patterns in existing distributed RDF repositories can be extended for other graph patterns such as OPTIONAL and UNION in our 3rdf system.**

I. Introduction

With rapidly rising interest in the Semantic Web the problem of storing and querying RDF [1] data is a key issue. The current centralized RDF storage and query engines like Sesame [2], Jena [3], and 3store [4] have limitations both in their failure tolerance and in their scalability. Moreover, due to their limited capacities, they will not be able to handle the anticipated load of Semantic web information available in the future. Thus, efficiently distributed databases are a necessary precondition for the acceptance of the Semantic Web. Peer-to-Peer networks (p2p) can offer a foundation layer for such distributed databases.

Distributed Hash Tables (DHTs), earlier introduced in [5] for relieving hot spots in the Internet, have become the data distribution method of choice for such distributed database systems. Existing systems like RDFPeers [6], Atlas [7], [8], [9], and BabelPeers [10], [11] use DHTs to store and query RDF triples in a distributed manner. These RDF database systems store three copies of each triple indexed by the subject, predicate, and object to achieve an efficient search for triples by subject, predicate, or object. Triples with the same index key, such as the subject, are on the same peer. The use of hash functions in DHTs for load balancing (of triples in this case) has the drawback that it destroys possible semantic relations between data with similar index keys on application level. Thus, data with different index keys stored on the same peer is usually unrelated. Without application-based data placement, it is not possible to organize the triples in the network in such a way that triples which tend to be queried combined in queries are stored on nearby peers in the network for fast communication with as small traffic as possible.

GridVine [12] addresses this by using the p2p network P-Grid [13]. P-Grid provides a distributed search tree for order-preserving indexing. Domain-related prefixes in subjects, predicates, and objects ensure with their ordering that semantically related data within the same domain is stored on nearby peers or even on the same peer.

This paper presents a scalable and distributed RDF triple repository named *3rdf* for storing and querying RDF data. 3rdf is built on top of the 3nuts [14] p2p network. Like P-Grid, 3nuts offers a distributed search tree and a distributed data storage usually for extended meta information besides search keys which we call *index data*. Besides this same base, 3nuts comes with further features we want to exploit in 3rdf to reduce traffic and response time of SPARQL queries. These offer a link structure optimization on the network layer for small latencies between peers during the search. The network also allows applications like 3rdf to adapt the network structure according to the attempted search behavior. For this we continuously analyze the nature of upcoming requests in order to optimize the routing structure dynamically for a speed-up of future requests. In our 3rdf system we use the approved distributed SPARQL query evaluation scheme of [7] and extend it for SPARQL query fragments like OPTIONAL and UNION graph patterns, which have not been addressed in a p2p framework so far.

The remainder of the paper is organized as follows: We start with an overview of recent work in the field of distributed RDF stores in Section II. Our 3rdf system uses the 3nuts p2p network for a distributed application which is briefly described in Section III. Section IV then presents our 3rdf system architecture and the mechanisms used to store and query RDF data. Finally, in Section V we make a conclusion and offer suggestions for future work directions.

II. Related Work

Existing systems like RDFPeers [6], Atlas [7], [8], [9] and BabelPeers [10], [11] use distributed hash table (DHT) based networks for distributed storage and querying of RDF data. The basic idea here is to store each triple at three locations using the hash value of subject, predicate, and object. Triples with a specific subject, predicate, or object are obtained during query evaluation by computing the hash value of that specific key again to resolve the peer providing these triples.

RDFPeers [6] was the first work to consider the storage and querying of RDF data on top of a DHT. In this system the evaluation of only atomic triple patterns and triple patterns with the same variable subject and possibly different constant predicates have been developed. The authors in [7] extended this work in [6] and presented two novel algorithms for the evaluation of conjunction of RDF triple patterns. They further improved their system in [9] with new query optimization techniques for reducing query response time and bandwidth usage. In their greedy optimization algorithms they tried to minimize the size of the intermediate relation produced during the query evaluation using selectivity based heuristics.

Traditional DHT-based networks such as Chord [15] or Pastry [16], which are used as an underlying network in these systems, apply uniform hash functions to map data keys to the peers in the network. This achieves good storage load balancing but sacrifices the relationship of the keys (attributes) based on their order. Keys which are semantically close at the application level are heavily fragmented in the DHT. Since semantically close data items are stored in a highly fragmented manner in DHTs, the efficiency of range queries or queries posed on semantically related attributes is significantly spoiled.

GridVine [12] is another distributed RDF system proposed for the storage and querying of RDF data. GridVine uses the P-Grid p2p network [13] to provide an order-preserving search tree instead of a DHT-based search structure. The ordering in the tree can represent the semantical proximity of closely related RDF triples (e.g. predicates with the same prefix will be organized in the same subtree). In contrast to other networks like Pastry [16] and 3nuts [14], P-Grid does not provide a routing structure with latency-optimized links for reducing the search time in the network.

III. The 3nuts P2P Network

Our proposed 3rdf system for a distributed RDF database is build on top of the 3nuts peer-to-peer network [14]. This overlay network establishes a distributed search tree providing point and range queries in $\mathcal{O}(\log n)$ routing hops[1] with high probability where n denotes the number of peers in the network. There are two reasons for choosing this network. First, there is an implementation in *Java* that we can use for our system. Most other semantic networks that provide range queries except for PGrid are only theoretical. Secondly, the 3nuts network provides further features which allow to adapt the network structure to the search structure for reducing the communication time and traffic on the application layer. While other RDF systems aim to achieve the principle of data independence [17] and focus on enhancing the query processing on application layer, we see real potential in the interaction of application and network. For this, 3nuts provides three types of locality:

With *network locality* the routing structure of the network is optimized for links with low turn-around-times (ping), e.g. peers choose communication partners with a low ping for short latencies.

The distributed search tree of 3nuts preserves key ordering. Each peer manages a continuous part of the key space, e.g. all data elements share the same prefix. The similarity of data elements can be mapped one-dimensionally to the data ordering, which we call *information locality*, e.g. nearby elements get the same prefix key (see Fig. 1). The benefit here is that a lookup between two peers sharing the same prefix takes less hops. So if a SPARQL query contains data related to several keys sharing the same prefix, the number of hops required to reach all these keys is reduced (at best, some keys are managed by the same peer).

The 3nuts network also allows a peer to have additional routing structures in certain paths in the tree. In this so called *interest locality*, peers with special interest for a certain search key or prefix range can voluntary manage data there or simply have fast routing in these paths with an additional routing structure. Of course, this benefit comes with the extra costs of maintaining additional routing links there. So how can we exploit this feature in a RDF system? In a nutshell, if we know that certain routing paths between some RDF keys are frequently used, we can establish routing shortcuts with interest locality to reduce traffic and query response time. Of course, one could also keep such shortcut links between applications on the RDF layer. But on the network layer the shortcuts are automatically maintained in the dynamic network scenario and integrated in the query algorithm of the network with all its backup techniques for failed routing.

The latter two localities require interaction of the application and the network. For information locality we have to preset the ordering of the keys given for the RDF data which have to fit to the data correlation in the SPARQL requests. To use interest locality, we have to analyze the SPARQL queries online and establish/destroy routing shortcuts dynamically.

3nuts supports two basic operations: *Get(key)* for searching a certain key (or key range) and retrieving the associated data items and *Put(key, value)* for storing new data items. Optional operations for interest locality are *AddVolunteer(key)* and *RemoveVolunteer(key)* for establishing and destroying voluntary routing structures for shortcuts.

IV. 3rdf in detail

In this section we elaborate on the 3rdf architecture and API and sketch the algorithms for storing and querying RDF data. In addition, we will show how the localities (network, information, and interest locality) provided by 3nuts help us to improve the performance in terms of query response time and bandwidth usage.

A. System architecture

We have implemented a prototype of the 3rdf system for initial tests. Figure 1 illustrates a node in the distributed system. It is a two layer model with the 3nuts network layer as the basis for the distributed application at the bottom and the 3rdf layer for RDF storing and querying on top. In the 3nuts

[1]Multi-hop routing in an overlay network: a request for a search key routed over several peers (hops) from the requesting peer to the target peer which is responsible for the search key and generates the response.

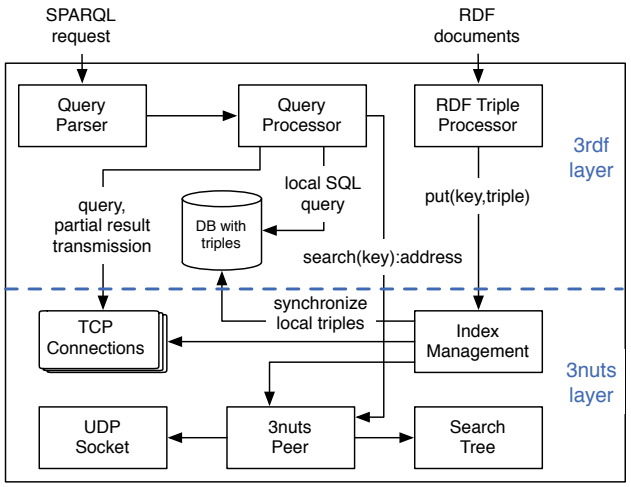

Fig. 1. Implementation overview of a 3rdf node in a distributed environment.

network, each 3nuts peer has a local view on the *Search Tree* which enables the peer the search in the distributed tree of the entire network. For routing, the peers use the UDP protocol. Based on the overlay network, there is a distributed *Index Management* which provides operations for putting and getting triples from the distributed network. Here, the search functionality of the 3nuts network in the search tree is used to place triples at the correct peers responsible for the corresponding index keys and on the other hand for downloading triples for a certain index key from the responsible peers. TCP/IP connections are used here for triple exchange. The same network connections are shared by the 3rdf query processing for exchanging queries and results between 3rdf nodes.

Input for the distributed RDF storage are RDF documents which are converted to tuples $(\text{key}, \text{triple})$ in the *RDF Triple Processor* in order to inject them into the Index Management with the Put-operation. Three tuples are created for each triple with the different keys for subject, predicate, and object to index all three parameters. Each 3rdf node is then responsible for a range of index keys, and the Index Management stores the corresponding tuples for these index keys. As we cannot directly perform SPARQL queries on the internal data structures of the Index Management, we synchronize the triples from the Index Management with a local database. This enables us to state SQL queries on the triples in the database.

To perform SPARQL queries, we first transform a SPARQL statement into a sequence of so called *triple patterns* in the *Query Parser*. This separation of the query into smaller partial queries reflects the single steps of execution at different 3rdf nodes only with their local database and some intermediate results. The triple pattern sequence is then passed to the *Query Processor* which controls the distributed execution of the query. There are basically two cases in the distributed execution. In the first case, the 3rdf node will execute the next triple pattern in the sequence if the 3rdf node is capable of resolving it with its local database because the node is responsible for a given subject, predicate, or object and has the corresponding triples in its database. Otherwise it will use the search operation of the 3nuts peer to find the peer that can execute the query and transmit the query and some intermediate results to that 3rdf node. For transmission it uses the TCP connections of the 3nuts network layer which can be established between the peers in the network on demand.

B. Storing RDF Triples

Each node in our system can publish RDF resources in the network. For this it has a *RDF Triple Processor* which supports the encoding of different types of RDF resources into metadata in form of RDF triples. The RDF triples are then inserted into the distributed triple storage. In the RDF data model, resources are expressed as subject-predicate-object expressions, called triples in RDF terminology. The subject in a RDF triple denotes the resource, and the predicate expresses a relationship between the subject and the object[1].

```
@prefix bench: <http://localhost/vocabulary/bench/>
@prefix ub: <http://www.lehigh.edu/zhp2/2004/univ-bench#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix u0: <http://www.Department0.University0.edu#>
u0:G7   rdf:type   bench:GraduateStudent.
u0:G7   ub:name    Jim.
u0:G7   ub:email   Jim@ub.com.
u0:G8   rdf:type   bench:GraduateStudent.
u0:G8   ub:name    Pet.
```

Listing 1. Example RDF statements about a resource *GraduateStudent48* encoded in RDF/N3 format.

RDF data is inserted as RDF documents into the 3rdf system. Each document is decomposed into a collection of RDF triples to store them distributedly in the network. Since the majority of RDF query languages, including SPARQL, are based on constraints-search of the triple's subject, predicate or object, we index and thus store each triple three times using its subject, predicate, and object as different keys. Triples are stored in the 3nuts network by using the *Put(key, value)* operation. This operation finds the responsible peer for the key and transmits the triple there. The responsible peer then manages the triple and inserts it into its database ready for query evaluations.

RDF resources are normally represented by Uniform Resource Identifiers (URIs) and hence closely related resources share a common prefix (e.g. the 'name' and 'email' predicates of the subject 'G7' in Listing 1 share the same prefix 'ub'). The information locality of the 3nuts search tree then guarantees that the triples corresponding to the closely related keys are stored on nearby nodes or at best on the same node.

C. Resolving SPARQL Queries in 3rdf

We extend the distributed evaluation of SPARQL BASIC graph pattern queries considered by the existing approaches RDFPeers, Atlas, BabelPeers, and GridVine with the graph patterns OPTIONAL and UNION. SPARQL is the W3C candidate recommendation query language for RDF [18].

To understand the evaluation of SPARQL BASIC, OPTIONAL, and UNION graph patterns in 3rdf, we will first briefly describe the semantics of SPARQL graph pattern

```
@prefix bench: <http://localhost/vocabulary/bench/>
@prefix ub: <http://www.lehigh.edu/zhp2/2004/univ-bench#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix u0d0: <http://www.Department0.University0.edu#>
SELECT ?Y1 ?Y2
WHERE {
  ?X rdf:type  bench:GraduateStudent.
  ?X ub:name ?Y1.
  OPTIONAL { ?X ub:email  ?Y2 }
}
```

Listing 2. SPARQL query returning names and emails of all graduate students.

expressions. The definition of so-called *mappings* and *compatibility* of mappings is helpful to understand the semantics of SPARQL. In the following we use the notation from [19].

Let UL represent the set of URIs and literals in RDF and V be a set of variables disjoint from UL. A *mapping* is a partial function $\mu : V \rightarrow UL$ from set V to UL. The domain $dom(\mu)$ of a mapping μ is defined as the subset of V for which μ is defined. Two mappings μ_1 and μ_2 are said to be *compatible* if $\mu_1(?X) = \mu_2(?X)$ for all $?X \in dom(\mu_1) \cap dom(\mu_2)$. We denote by $var(t)$ all variables in triple pattern t, and $\mu(t)$ represents the triple pattern obtained when replacing all variables in triple pattern t according to the mapping μ. We now define the SPARQL expression semantics for 3rdf based on the definition of [19].

Definition 1: Let D be an RDF database, t be a triple pattern, Q_1, Q_2 represent SPARQL expressions, and $v \subset V$ be a finite set of variables. Then the evaluation of a SPARQL expression over D, denoted by $[[.]]_D$, is recursively defined as follows:

1) $[[t]]_D = \{\mu \mid dom(\mu) = var(t) \text{ and } \mu(t) \in D\}$.
2) $[[\text{SELECT}_v(Q_1)]]_D = \pi_v([[Q_1]]_D)$
3) $[[Q_1 \text{ AND } Q_2]]_D = [[Q_1]]_D \bowtie [[Q_2]]_D$.
4) $[[Q_1 \text{ OPTIONAL } Q_2]]_D = [[Q_1]]_D \rtimmod [[Q_2]]_D$.
5) $[[Q_1 \text{ UNION } Q_2]]_D = [[Q_1]]_D \cup [[Q_2]]_D$.

For the evaluation of a SPARQL expression $(Q_1 \text{ OPTIONAL } Q_2)$ we consider the mapping μ_1 in $[[Q_1]]_D$ and determine if there is a mapping μ_2 in $[[Q_2]]_D$ such that μ_1 and μ_2 are compatible. If so, $\mu_1 \cup \mu_2$ belongs to $[[Q_1 \text{ OPTIONAL } Q_2]]_D$. In the absence of such a mapping μ_2, μ_1 belongs to $[[Q_1 \text{ OPTIONAL } Q_2]]_D$.

Example 1: We consider the evaluation of the SPARQL query in Listing 2 over the RDF database D in Listing 1. The query extracts all graduate students including the name plus the email address optional. By applying Definition 1 to the given query we obtain the expression:

$$E := \pi_{?Y1,?Y2}((([[?X, \text{rdf:type}, \text{bench:GraduateStudent}]]_D \bowtie [[?X, \text{ub:name}, ?Y1]]_D) \rtimmod [[(?X, \text{ub:email}, ?Y2)]]_D).$$

After applying the $\bowtie$ (join) and $\rtimmod$ (leftjoin) operators, expression E evaluates to the mapping set

$$\{\{?Y1 \rightarrow \text{Jim}, ?Y2 \rightarrow \text{Jim@ub.com}\}, \{?Y1 \rightarrow \text{Pet}\}\}$$

1) Query processing: We have adopted the query processing algorithm originally presented in [7], where the triple patterns contained in the query are iteratively resolved by a chain of nodes. In the process, each node adds to an intermediate result all triples of its local database, which are qualified for the evaluated query so far. The last node in the line of nodes then has the complete result and returns it to the requesting node. The query request takes the parameters $(id, t_i, Q, I, type, IP(p))$ where *id* denotes the query ID, t_i is the currently active triple pattern, Q is the list of remaining triple patterns, I is the relation that will accumulate triples (intermediate results generated so far), $type$ represents the query type i.e. BASIC or OPTIONAL graph patterns, and *IP(p)* is the IP address of node *p* that posed the query. Initially, t_i is the first triple pattern in the given query with $i = 1$, Q contains all triple patterns except t_i, and I is empty.

The node, which initiates a new SPARQL query request, first reorders the triple patterns to avoid the computation and transformation of Cartesian products through the network and then routes the query request to node r_i through prefix search using one of the constants in t_i. When node r_i receives the query request, it first resolves the triple pattern t_i with a local relational query on the local triple table R, i.e. it computes the relation $T = \pi_X(\sigma_{SC}(R))$ where SC is a selection condition and X represents the positions of variables. For example, if t_i is $(?s_i, p_i, o_i)$ then $T = \pi_{\text{subject}}(\sigma_{\text{predicate}=p_i \wedge \text{object}=o_i}(R))$.

Then, depending on the type, i.e. BASIC (AND) or OPTIONAL, r_i computes a new relation I' as follows:

1) If $type = \text{AND}$ then $I' = \pi_S(I \bowtie T)$
2) If $type = \text{OPTIONAL}$ and t_i is not in the optional part of the query then $I' = \pi_S(I \bowtie T)$
3) If $type = \text{OPTIONAL}$ and t_i is in the optional part of the query then $I' = \pi_S(I \rtimmod T)$

The set S identifies the attributes of I and T that exist in answer variables or are needed for the evaluation of the remaining triple patterns. If I' is not the empty relation then r_i sends the query message $(id, t_{i+1}, Q, I', type, IP(p))$ to the node responsible for the evaluation of the next triple pattern t_{i+1}. When the last node is reached and the last triple pattern of the given query is evaluated, this node simply returns relation I' back to the start node p using its IP address *IP(p)*.

Example 2: Consider the evaluation of the OPTIONAL graph pattern in Listing 2 over RDF database D in Listing 1. Let r_1, r_2, and r_3 be the nodes responsible for the evaluation of the triple patterns in Listing 2. Node r_1 evaluates $(?X, \text{rdf:type}, \text{ub:GraduateStudent})$ and finds T={(u0:G7), (u0:G8)}, I'={(u0:G7), (u0:G8)}. Node r_2 evaluates $(?X, \text{ub:name}, ?Y1)$ and finds $T = \{(\text{u0:G7, Jim}), (\text{u0:G8, Pet})\}$, I'={(u0:G7, Jim), (u0:G8, Pet)}. The last node r_3 computes $(?X, \text{ub:email}, ?Y2)$ and finds $T = \{(\text{u0:G7}, \text{Jim@ub.com})\}$, $I' = \{(\text{Jim}, \text{Jim@ub.com}), (\text{Pet})\}$.

For the evaluation of UNION graph patterns we extend the algorithm mentioned above to achieve a better distribution of the query processing load and to improve the query response time and bandwidth usage. When a node p poses

a SPARQL query of the form $(P_1 \text{ UNION } P_2)$, it creates the query request messages $(id, t_{i_1}, Q_1, I_1, type, IP(p))$ and $(id, t_{i_2}, Q_2, I_2, type, IP(p))$ for the graph patterns P_1 and P_2 respectively. These query messages are then processed in parallel by a chain of nodes in the same way as discussed for the BASIC and OPTIONAL graph patterns in the above algorithm, and the results are stored as intermediate results in relations I_1 and I_2 respectively. When the last nodes for the evaluation of P_1 respectively P_2 are reached, they return I_1 respectively I_2 back to the originating node p, which then creates the final result $I' = \pi_S(I_1) \text{ UNION } \pi_S(I_2)$.

2) Exploiting Interest Locality: While we have already placed triples with the same index prefix on nearby nodes in the network with information locality resulting in a fast routing time in between, triple indexes with more diverse prefixes still need routing time of $\mathcal{O}(\log n)$ hops in the p2p network (e.g. 'rdf:type' and 'ub:...' in List. 2). The interest locality supported by the 3nuts network also gives us the opportunity to speed-up the routing between more diverse prefixes by placing routing shortcuts. But there is a trade-off between speed and costs in the form of network traffic. The more routing shortcuts we place, the more the network structure is extended, which results in higher maintenance costs for routing links. But the good news is that the traffic produced in the query execution can be reduced by using the routing shortcuts instead of the original longer routing paths. Thus, when a placed routing shortcut is heavily used by lots of SPARQL queries, the routing time can be shortened and the overall traffic is reduced at the same time.

So to reach the three goals fast query latency, reasonable small traffic and network structure, we can only place a limited amount of routing shortcuts based on the frequency of being used in SPARQL queries. For this we measure the frequency of successive occurring triple patterns in SPARQL queries. In the distributed execution, the 3rdf nodes responsible for an index key and processing queries, that refer to that key, simply measure what are the most frequent next index keys in the queries. On the basis of the query measurement and the knowledge about extra network maintenance costs for routing shortcuts, we can determine which routing shortcuts to create.

Another interesting aspect is that we can also balance faster routing shortcuts and fewer shortcuts with less routing structure. If we know, for instance, that many queries which start with the key 'rdf:type' are followed by a key 'ub:name' or 'ub:email', we can choose between placing two shortcuts from the peer being responsible for 'rdf:type' to 'ub:name' and 'ub:email' or just one shortcut to the prefix 'ub:'.

V. Conclusions

We have presented 3rdf, a distributed system for storing and querying RDF data on top of the 3nuts p2p network. We showed how the network, information, and interest locality provided by the 3nuts network can be exploited to improve the performance in terms of query response time and bandwidth usage. In addition, we studied how the evaluation of the BASIC graph patterns in current distributed RDF systems can be extended for other graph patterns, such as OPTIONAL and UNION graph patterns, in the 3rdf system.

In future work, we would like to run some benchmarks on our 3rdf system and compare the performance of our system with comparable systems. We intend to improve the query performance with more sophisticated algorithms by exploiting parallelism and reducing traffic for intermediate results.

References

[1] C. Gutierrez, C. Hurtado, and A. O. Mendelzon, "Foundations of semantic web databases," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. PODS '04. New York, NY, USA: ACM, 2004, pp. 95–106.

[2] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *Proceedings of the First Internation Semantic Web Conference*, July 2002, pp. 54–68.

[3] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, and J. Database, "Efficient rdf storage and retrieval in jena2," in *EXPLOITING HYPERLINKS 349*, 2003, pp. 35–43.

[4] S. Harris and N. Gibbins, "3store: Efficient bulk rdf storage," in *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems PSSS'03*, 2003.

[5] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663.

[6] M. Cai and M. Frank, "Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network," in *Proceedings of the 13th international conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 650–657.

[7] E. Liarou, S. Idreos, and M. Koubarakis, "Evaluating conjunctive triple pattern queries over large structured overlay networks," in *International Semantic Web Conference*, 2006, pp. 399–413.

[8] Z. Kaoudi, I. Miliaraki, and M. Koubarakis, "Rdfs reasoning and query answering on top of dhts," in *Proceedings of the 7th International Conference on The Semantic Web*, 2008, pp. 499–516.

[9] Z. Kaoudi, K. Kyzirakos, and M. Koubarakis, "Sparql query optimization on top of dhts," in *Proceedings of the 9th international semantic web conference on The semantic web - Vol. Part I*, 2010, pp. 418–435.

[10] D. Battré, F. Heine, A. Höing, and O. Kao, "On triple dissemination, forward-chaining, and load balancing in dht based rdf stores," in *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, 2007, pp. 343–354.

[11] F. Heine, "Scalable p2p based rdf querying," in *Proceedings of the 1st international conference on Scalable information systems*, ser. InfoScale '06. New York, NY, USA: ACM, 2006.

[12] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. V. Pelt, "Gridvine: Building internet-scale semantic overlay networks," in *In International Semantic Web Conference*, 2004, pp. 107–121.

[13] K. Aberer, "P-grid: A self-organizing access structure for p2p information systems," in *Proceedings of the 9th International Conference on Cooperative Information Systems*, ser. CooplS '01. London, UK: Springer-Verlag, 2001, pp. 179–194.

[14] T. Janson, P. Mahlmann, and C. Schindelhauer, "A self-stabilizing locality-aware peer-to-peer network combining random networks, search trees, and dhts," in *ICPADS*, 2010, pp. 123–130.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, August 2001.

[16] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, 2001, pp. 329–350.

[17] J. M. Hellerstein, "Toward network data independence," *SIGMOD Rec.*, vol. 32, pp. 34–40, September 2003.

[18] "Sparql query language for rdf." [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[19] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Trans. Database Syst.*, vol. 34, pp. 16:1–16:45, September 2009.