RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network

Min Cai
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
mcai@isi.edu

Martin Frank
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
frank@isi.edu

ABSTRACT

Centralized Resource Description Framework (RDF) repositories have limitations both in their failure tolerance and in their scalability. Existing Peer-to-Peer (P2P) RDF repositories either cannot guarantee to find query results, even if these results exist in the network, or require up-front definition of RDF schemas and designation of super peers. We present a scalable distributed RDF repository ("RDFPeers") that stores each triple at three places in a multi-attribute addressable network by applying globally known hash functions to its subject, predicate, and object. Thus, all nodes know which node is responsible for storing triple values they are looking for, and both exact-match and range queries can be efficiently routed to those nodes. RDFPeers has no single point of failure nor elevated peers, and does not require the prior definition of RDF schemas. Queries are guaranteed to find matched triples in the network if the triples exist. In RDFPeers, both the number of neighbors per node and the number of routing hops for inserting RDF triples and for resolving most queries are logarithmic to the number of nodes in the network. We further performed experiments that show that the triple-storing load in RDFPeers differs by less than an order of magnitude between the most and the least loaded nodes for real-world RDF data.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems – Distributed Applications, Distributed Databases; C.2 [**Network Protocols**]: Routing Protocols; H.2.3 [**Database Management**]: Languages – Query Languages

General Terms

Algorithms, Design

Keywords

Semantic Web, Peer-to-Peer, Distributed RDF Repositories

1. INTRODUCTION

RDF [1] meta-data makes flexible statements about resources that are uniquely identified by URIs. RDF statements are machine-processable and machine-understandable, and statements about the same resource can be distributed on the Web and made by different users. RDF schemata [2] are extensible and evolvable over time by using a new base URI every time the schema is revised. The distribution of RDF statements provides great flexibility for annotating resources. However, distributed RDF documents on Web pages are hard to discover. That is, obviously, just because you put an RDF document on your Web site does not mean that others can find it, much less issue structured queries against it. One approach is to crawl all possible Web pages and index all RDF documents in centralized search engines, "RDF Google" if you wish, but this approach makes it difficult to keep the indexed RDF up to date. For example, it currently takes Google many days to index a newly created Web page. Further, this approach has a large infrastructure footprint for the organization providing the querying service, and is a centralized approach on top of technologies (RDF, the Internet itself) that were intentionally designed for decentralized operation. One choice for non-centralized RDF repositories is Edutella [12] which provides an RDF-based meta-data infrastructure for P2P applications. It uses a Gnutella-like [17] unstructured P2P network which has no centralized index or predictable location for RDF triples. Instead, RDF queries are flooded to the whole network and each node processes every query. Measurement studies [20, 21] show that Gnutella-like unstructured P2P networks do not scale well to a large number of nodes. This is because their flooding mechanism generates a large amount of unnecessary traffic and processing overhead on each node, unless a hop-count limit is set for queries – but then the queries cannot guarantee finding results, even if these results exist in the network. An Edutella successor [13] provides better scalability by introducing super-peers and schema-based routing; however, it requires up-front definition of schemas and designation of super peers. This paper presents a scalable and distributed RDF triple repository named RDFPeers for storing, indexing and querying individual RDF statements, and which does not require the definition of an RDF schema before inserting RDF triples into the network. RDF triple storage providers self-organize into a cooperative structured P2P network based on randomly chosen node identifiers. When an RDF triple is inserted into the network, it will be stored three times, based on applying a globally-known hash function to its subject, predicate, and object values. Queries can then efficiently be routed to those nodes in the network where the triples in question are known to be stored if they exist.

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

2. RDFPEERS ARCHITECTURE

Our distributed triple repository consists of many individual nodes called RDFPeers that self-organize into a multi-attribute addressable network (MAAN) [4] which extends Chord [22] to efficiently answer multi-attribute and range queries. However, MAAN only supported predetermined attribute schemata with a fixed number of attributes. RDFPeers exploits MAAN as the underlying

network layer and extends it with RDF-specific storage, retrieval, and load balancing techniques. Figure 1 shows the architecture of RDFPeers. Each node in RDFPeers consists of five components: the MAAN network layer, the RDF triple loader, the local RDF triple storage, the native query resolver and the RDQL-to-native-query translator. The underlying MAAN protocol contains three classes of messages for (a) topology maintenance, (b) storage, and (c) search. (a) The topology maintenance messages are used for keeping the correct neighbor connections and routing tables and include *JOIN/LEAVE*, *KEEPALIVE* and other network-structure-stabilizing messages. (b) The *STORE* message inserts triples into the network. (c) The *SEARCH* message visits the nodes where the triples in question are known to be stored, and returns the matched triples to the requesting node. The RDF triple loader reads an RDF document, parses it into RDF triples, and uses MAAN's *STORE* message to store the triples into the RDFPeers network. When an RDFPeer receives a *STORE* message, it stores the triples into its Local RDF Triple Storage component such as a relational database. The native query resolver parses native RDFPeers queries and uses MAAN's *SEARCH* message to resolve them. There can be a multitude of higher-level query modules on top of the native query resolver which map higher-level user queries into RDFPeers' native queries, such as an RDQL to Native Query Translator.

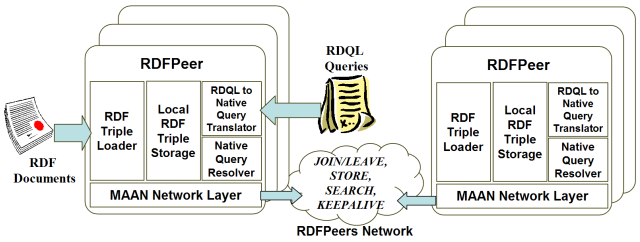

Figure 1: The Architecture of RDFPeers

3. MAAN AS USED FOR RDFPEERS

MAAN [4] uses the same one-dimensional modulo-2^m circular identifier space as Chord, where m is the number of bits in node identifiers and attribute hash values. Every node in MAAN is assigned a unique m-bit identifier, called the node ID, and all nodes self-organize into a ring topology based on their node IDs. The node ID can be chosen locally, for example by applying a hash function to the node's IP address and port number. In Chord, bundles of related attribute-value pairs such as "name: John, age: 27" are called "resources", a term we will avoid in this paper because of its different meaning in RDF. Note that for RDFPeers' use of MAAN, a "bundle of related attribute-value pairs" is always synonymous with "an RDF triple". Unlike Chord in which these bundles can only be stored and looked up by one unique key, they can be stored and looked up by any attribute value in MAAN. Chord uses SHA1 hashing [11] to assign each key a unique m-bit identifier. MAAN uses the same hashing for string-valued attributes. However, for numeric attributes MAAN uses locality preserving hash functions to assign each attribute value an identifier in the m-bit space. Here, we refer to the hashing image of the key in Chord as well as to the hashing image of the attribute value in MAAN as "the key" which is an identifier in the circular m-bit space. Suppose we have an attribute a with numeric values $v \in [v_{\min}, v_{\max}]$. (Note that in RDFPeers, the only attributes that can have numeric values are the objects given that subjects and predicates are always non-numeric URIs in RDF.) A simplistic locality preserving hash function we could use is $H(v) = (v - v_{\min}) \times (2^m - 1)/(v_{\max} - v_{\min})$, where $v \in [v_{\min}, v_{\max}]$. Key k is assigned to the first node whose identifier is equal to or follows k in the identifier circle. This node is called the successor node of key k, denoted by $successor(k)$. Similar to Chord, each node in MAAN maintains two sets of neighbors, the *successor list* and the *finger table*. The nodes in the successor list immediately follow the node in the identifier space, while the nodes in the finger table are spaced exponentially around the identifier space. The finger table has at most m entries. The i-th entry in the table for the node with ID n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle, i.e. $s = successor(n+2^{i-1})$, where $1 \leq i \leq m$ and all arithmetic is modulo 2^m. MAAN uses Chord's successor routing algorithm to forward a request of key k to its successor node. If a node n receives a request with key k, the node searches its successor list for the successor of k and forwards the request to it if possible. If it does not know the successor of k, it forwards the request to the node j whose identifier most immediately precedes k in its finger table. By repeating this process, the request gets closer and closer to the successor of k. Since the fingers on each node are spaced exponentially around the identifier space, each hop from node n to the next node covers at least half the identifier space (clockwise) between n and k. The average number of hops for this routing is $O(\log N)$ for a network with N nodes.

MAAN stores each bundle of attribute-value pairs on the successor nodes of the keys for all its attribute values. Suppose each bundle has M pairs $<a_i, v_i>$ and $H_i(v)$ is the hash function for attribute a_i (Note that M is always 3 in RDFPeers, a_1 is always *subject*, a_2 is always *predicate*, and a_3 is always *object*.) Each bundle of attribute-value pairs will be stored at node $n_i = successor(H(v_i))$ for each attribute value v_i, where $1 \leq i \leq M$. A *STORE* message for attribute value v_i is routed to its successor node using the above successor routing algorithm. M nodes store the same bundle consisting of M attribute-value pairs, each by keying on a different attribute. Thus, the routing hops for storing a bundle of attribute-value pairs is $O(M \log N)$ for bundles with M attributes.

Since numeric attribute values in MAAN are mapped to the m-bit identifier space using locality preserving hash function H, numerically close values for the same attribute are stored on nearby nodes. Given a range query $[l, u]$ where l and u are the lower bound and upper bound respectively, nodes which contain attribute value $v \in [l, u]$ must have an identifier equal to or larger than $successor(H(l))$ and equal to or less than $successor(H(u))$.

Suppose node n wants to search for bundles with attribute value $v \in [l, u]$ for attribute a. Node n composes a *SEARCH* message and uses the successor routing algorithm to route it to node n_l, the successor of $H(l)$. The search message has parameters k, a, R, and X. k is the key used for successor routing, initially $k = H(l)$. a is the name of the attribute we are interested in, R is the desired query range $[l, u]$ and X is the list of bundles of attribute-value pairs discovered in the range. Initially, X is empty. When node n_l receives the search message, it searches its local sets and appends those sets that satisfy the range query for attribute a to X in the message. Then it checks whether it is the successor of $H(u)$ also. If true, it sends back the search result in X to the requesting node n. Otherwise, it forwards the search message to its immediate successor n_i. Node n_i repeats this process until the message reaches node n_u, the successor of $H(u)$. Thus, routing the search message to node n_l via successor routing takes $O(\log N)$ hops for N nodes. The next sequential forwarding from n_l to n_u takes $O(K)$, where K is the number of nodes between n_l and n_u. So there are total $O(\log N + K)$ routing hops to resolve a range query for one at-

tribute. Given that the nodes are uniformly distributed in the m-bit identifier space, K is $N \times s$ where s is the selectivity of the range query and $s = (l - u)/(v_{\max} - v_{\min})$.

MAAN supports multi-attribute and range queries using a single-attribute-dominated query resolution approach. Suppose X are the bundles of attribute-value pairs satisfying all sub-queries, and X_i are the bundles satisfying the sub-query on attribute a_i, where $1 \leq i \leq M$. So we have $X = \bigcap X_i$ and each X_i is a superset of X. This query resolution approach first computes a X_k which satisfies one sub-query on attribute a_k. Then it applies the sub-queries for other attributes on these candidate bundles and computes the intersection X which satisfies all sub-queries. Here, we call attribute a_k the dominant attribute. In order to reduce the number of the candidate sets which do not satisfy other sub-queries, we carry all other sub-queries in the *SEARCH* message, and use them to filter out the unqualified bundles of attribute-value pairs locally at the nodes visited. Since this approach only needs to do one iteration around the Chord identifier space for the dominant attribute a_k, it takes $O(\log N + N \times s_k)$ routing hops to resolve the query, where s_k is the selectivity of the sub-query on attribute a_k. We can further minimize the routing hops by choosing the attribute with minimum selectivity as the dominant attribute, presuming, of course, that the selectivity is known in advance; in that case, the routing hops will be $O(\log N + N \times s_{\min})$, where $s_{\min}$ is the minimum range selectivity for all attributes in the query.

Although the simplistic locality preserving hash function above keeps the locality of attribute values it does not necessarily produce uniform distributions of hashing values if the distribution of attribute values is not uniform. Consequently, the load balancing of resource entries can be poor across the nodes. To address this problem, we proposed a uniform locality preserving hashing function in MAAN which always produces uniform distribution of hashing values if the distribution function of input attribute values is continuous and if the distribution is known in advance. (The former condition is satisfied for many common distributions, such as Gaussian, Pareto, and Exponential distributions.) Suppose attribute value v conforms to a certain distribution with continuous and monotonically increasing distribution function $D(v)$ and possibility function $P(v) = \frac{dD(v)}{dv}$, and $v \in [v_{\min}, v_{\max}]$. We can design a uniform locality preserving hashing function $H(v)$ as follows: $H(v) = D(v) \times (2^m - 1)$.

4. STORING RDF TRIPLES

RDF documents are composed of a set of RDF triples. Each triple is in the form of *subject, predicate, object*. The *subject* is the resource about which the statement was made. The *predicate* is a resource representing the specific property in the statement. The *object* is the property value of the predicate in the statement. The object is either a resource or a literal; a resource is identified by a URI; literals are either plain or typed and have the lexical form of a unicode string. Plain literals have a lexical form and optionally a language tag, while typed literals have a lexical form and a datatype URI. The following triples show three different types of objects, resource, plain literal, and typed literal, respectively.

```
@prefix info: <http://www.isi.edu/2003/11/info#>  .
@prefix dc:   <http://purl.org/dc/elements/1.1/>  .
@prefix foaf: <http://xmlns.com/foaf/0.1/>  .
@prefix xmls: <http://www.w3.org/2001/XMLSchema#> .

<info:rdfpeers> <dc:creator> <info:mincai>  .
<info:mincai>   <foaf:name>  "Min Cai" .
<info:mincai>   <foaf:age>   "28"^^<xmls:integer> .
```

In order to support efficient queries on distributed RDF triples, we exploit the overlay structure of MAAN to build a distributed index for these triples. In the *STORE* message one of the three attribute values is designated as the destination of the routing, and we store each triple three times, once each based on its subject, predicate, and object. Each triple will be stored at the successor node of the hash key of the value of the routing key attribute-value pair. Since the value of attribute "subject" and "predicate" must be a URI which is a string, we apply the SHA1 hash function to mapping the subject value and predicate value to the m-bit identifier space in MAAN. However, the values of attribute "object" can be URIs, plain literals or typed literals. Both URIs and plain literals are strings and we apply SHA1 hashing on them. The typed literal can be either string types or numeric types, such as an enumeration type or a positive integer respectively. As discussed above, we apply SHA1 hashing on string-typed literals and locality preserving hashing on numeric literals. For example, to store the first triple above by subject, RDFPeers would send the following message in which the first attribute-value pair *("subject", info:rdfpeers)* is the routing key pair, and *key* is the SHA1 hash value of the subject value.

```
STORE {key, {("subject", <info:rdfpeers>),
            ("predicate", <dc:creator>),
            ("object", <info:mincai>)}}
where key=SHA1Hash("<info:rdfpeers>")
```

This triple will be stored at the node which is the successor node of *key*. Figure 2 shows how the three triples above are stored into an example RDFPeers network. It also shows the finger tables of example nodes $N6$ and $N14$ for illustration.

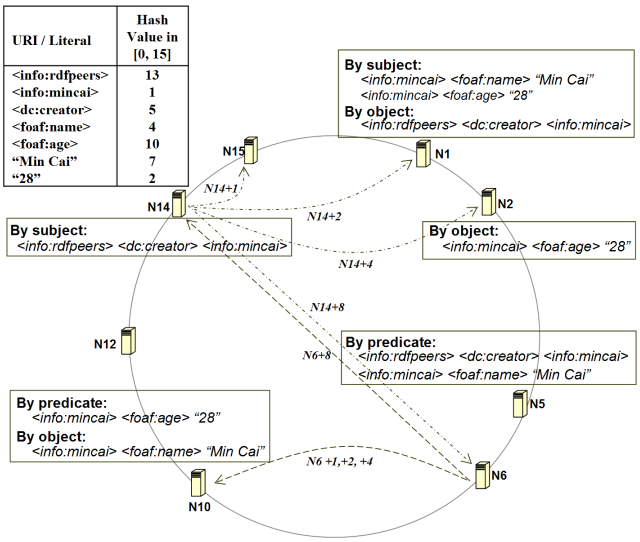

Figure 2: Storing three triples into an RDFPeers network of eight nodes in an example 4-bit identifier space that could hold up to 16 nodes. (In reality a much larger identifier space is used, such as 128 bits.)

Since nodes might fail and network connections might break, the triples stored on its corresponding successor nodes are replicated on its neighbors in Chord identifier space. This can be done by setting the parameter *Replica_Factor* in MAAN. Whenever a node receives a triple storing request, it will not only store the triple locally but also store it to as many of its immediate successors as the above parameter dictates. If any node fails or its connection breaks, its immediate successor and predecessor will detect it by checking the *KEEPALIVE* messages. If the node does not come back to life after a time-out period, nodes will repair the ring structure using the

Chord stabilization algorithm. After stabilization, the immediate successor node of the failed node will restore its replicas to its new predecessor.

5. NATIVE QUERIES IN RDFPEERS

Based on the above triple-storing scheme, we define a set of native queries which can be efficiently resolved via MAAN's multi-attribute range queries. These native queries include atomic triple patterns, disjunctive and range queries, and conjunctive multi-predicate queries.

5.1 Atomic Triple Patterns

An atomic query pattern is a triple pattern in which the subject, predicate, or object can each either be a variable or an exact value. The eight resulting possible queries are shown in Table 1.

Q1 is the most general and most expensive query which matches all triples. Since there is no restriction whatsoever on this triple pattern, we have to propagate this query to all nodes, which takes $O(N)$ routing hops for a network with N nodes.

We can use MAAN's routing algorithm to resolve queries Q2 through Q8 since we store each triple three times based on its subject, predicate, and object hash values. In these seven query patterns, there is always at least one value which is a constant, and we resolve the query by routing it to the node responsible for storing that constant, that node then matches these triples against the pattern locally and returns them to the requesting node.[1] For example, in Figure 2 if node $N6$ asks the native query (<*info:mincai*>, <*foaf:name*>, *?name*), we hash on *info:mincai* and get the hash value "1". Then $N6$ routes it to the corresponding node $N1$ (via $N14$). $N1$ filters triples locally using this pattern, and sends back the matched triple <*info:mincai*>, <*foaf:name*>, *"Min Cai"* to $N6$ (via $N5$).

No.	Query Pattern	Cost	Query Semantics
Q1	$(?s, ?p, ?o)$	$O(N)$	find all possible triples
Q2	$(?s, ?p, o_i)$	$\log N$	given object o_i of any predicate, find the subjects and predicates of matched triples
Q3	$(?s, p_i, ?o)$	$\log N$	given predicate p_i, find the subjects and objects of the triples having this predicate
Q4	$(?s, p_i, o_i)$	$\log N$	given object o_i of predicate p_i, find the subjects of matched triples
Q5	$(s_i, ?p, ?o)$	$\log N$	given subject s_i, find all predicates and objects of the resource identified by s_i
Q6	$(s_i, ?p, o_i)$	$\log N$	given subject s_i, find its predicate which has object o_i
Q7	$(s_i, p_i, ?o)$	$\log N$	given subject s_i, find its object of predicate p_i
Q8	(s_i, p_i, o_i)	$\log N$	return this triple if it exists otherwise return nothing

Table 1: The eight possible atomic triple queries for exact matches. The cost is measured in the number of routing hops needed to resolve each query.

[1]Note that we assume that the value is not "overly popular", in which case we would have to use $O(n)$ messages, see Section 7.2.

5.2 Disjunctive and Range Queries

RDFPeers' native queries support constraints on variables in the triple patterns. Q9 extends the above atomic queries with a constraint list which limits the domain of variables.

```
Q9             ::= TriplePattern 'AND' ConstraintList
TriplePattern  ::= Q1|Q2|Q3|Q4|Q5|Q6|Q7
ConstraintList ::= OrExpression ('&&' OrExpression)*
OrExpression   ::= Expression ('||' Expression)*
Expression     ::= Variable (NumericExpression
                           | StringExpression)+
NumericExpression ::=('>'|'<'|'='|'!='|'<='|'>=')
                     NumericLiteral
StringExpression ::= ('='|'!=')Literal
Literal        ::= PlainLiteral|URI|NumericLiteral
```

Variables can be either string-valued or numeric. Constraints can limit the domain of string values by enumerating a set of either allowed or forbidden constants. Numeric variables can additionally be limited to a set of disjunctive ranges.

```
(a) (?s, dc:creator, ?c) AND ?c=''Tom'' || ?c=''John''
(b) (?s, foaf:age, ?age) AND ?age > 10 && ?age < 20
```

As discussed in Section 3, MAAN can efficiently resolve range queries by using locality preserving hashing.[2] In addition to specifying a single range, Q9 can also specify a set of disjunctive ranges for attribute values. For example, a user can submit a range query for variable $?x$ and $?x \in \bigcup_{i=1}^{d}[l_i, u_i]$. Obviously, this kind of disjunctive range query could simply be resolved by issuing one query for each contiguous range and by then computing the union of the results. For a query with d disjunctive ranges, this takes $d \times O(\log N + N \times s)$, where s is the aggregate selectivity of the d ranges. So the number of hops in the worst case increases linearly with d and is not bounded by N. We can optimize this by using a range ordering algorithm which sorts these disjunctive query ranges in ascending order. Given a list of disjunctive ranges in ascending order, $[l_i, u_i], 1 \leq i \leq d$ where $l_i \leq l_j$ and $u_i \leq u_j$ iff $i \leq j$, the query request will be first routed to node n_{l1}, the successor node of $H(l_1)$ which is the key corresponding to the lower bound of the first range. Node n_{l1} then sequentially forwards the query to the successor node of the upper bound $H(u_1)$ if it itself is not the successor node of $H(u_1)$. Then node n_{u1} uses successor routing to forward the query to node n_{l2}, the successor node corresponding to the lower bound of the next range $[l_2, u_2]$, which in turn forwards the query to the successor node of $H(u_2)$. This process will be repeated until the query reaches the successor node of $H(u_d)$. This optimized algorithm exploits the locality of numeric MAAN data on the Chord ring and the ascending order of the ranges, reduces the number of routing hops, especially for cases where d is large, and bounds the routing hops to N. Disjunctive exact-match queries such as $?c \in \{Tom, John\}$ present a special case of the above disjunctive range queries where both the lower bound and upper bound of the range are equal to the exact-match value, and we use the same algorithm to resolve them.

5.3 Conjunctive Multi-Predicate Queries

In addition to atomic triple patterns and disjunctive range queries, RDFPeers handles conjunctive multi-predicate queries which describe a non-leaf node in the RDF graph by specifying a list of edges for this node. They are expressed as a conjunction of atomic

[2]Note that this is the one case where RDFPeers would benefit from up-front RDF Schema information: if say an Integer-valued object of some triples in reality only ever has values 1 through 10, RDFPeers can use a hash function that yields better load balancing for these triples.

triple patterns or disjunctive range queries for the same subject variable. Q10 consists of a conjunction of sub-queries where all subject variables must be the same.

```
Q10            := TriplePatterns 'AND' ConstraintList
TriplePatterns := (Q3|Q4|Q9)+
```

In Q10, we restrict the sub-query Q9 to be the Q3-style triple pattern with constraints on the object variable. Thus Q10 describes a subject variable with a list of restricting *predicate, object* or *predicate, object-range* pairs.

```
(?x, <rdf:type>, <foaf:Person>)
(?x, <foaf:name>, "John")
(?x, <foaf:age>, ?age) AND ?age > 35
```

To efficiently resolve these conjunctive multi-predicate queries, we use a recursive query resolution algorithm which searches candidate subjects on each predicate recursively and intersects the candidate subjects inside the network, before returning the search results to the query originator. The search request takes the parameters q, R, C, and I, where q is the currently active sub-query, R is a list of remaining sub-queries, C is a set of candidate subjects matching current active sub-query, and I is a set of intersected subjects matching all resolved sub-queries. Initially, q is the first sub-query in this multi-predicate query, R contains all sub-queries except q, C is empty and I is the whole set. Suppose the sub-query q for predicate p_i is $v_{li} \leq o_i \leq v_{ui}$, where v_{li} and v_{ui} are the lower bound and upper bound of the query range for the object variable o_i, respectively. When node n wants to issue a search request, it first routes the request to node $n_{li} = successor(H(v_{li}))$. The node n_{li} receives the request, searches its local triples corresponding to predicate p_i, appends the subjects matching sub-query q to C, forwards this request to its immediate successor n_{si} unless it is already the $successor(H(v_{ui}))$. Node n_{si} repeats this process until the search request reaches node $n_{ui} = successor(H(v_{ui}))$. When node n_{ui} receives the request, it also searches locally for the subjects matching sub-query q and appends them to C. It then intersects set I with set C, and pops the first sub-query in R to q. If R or I is empty, it sends the query response back with the subjects in I as the result; otherwise, it resolves sub-query q. This process will be repeated until no sub-queries remain or I is empty.

This recursive algorithm takes $O(\sum_{i=1}^{k}(\log N + N \times s_i))$ routing hops in the worst case, where k is the number of sub-queries and s_i is the selectivity of the sub-query on predicate p_i. However, it intersects the search results on different predicates in the network and will terminate the search process before resolving the query on all predicates if there are no matches left, i.e. I is empty. Thus, we can further reduce the average number of expected routing hops by sorting the sub-queries in ascending order of selectivity presuming the selectivity can be estimated in advance. For example, in the above three-predicate query, the sub-query on *rdf:type* might match many subjects, while *foaf:age* matches far fewer and *foaf:name* matches only a handful. After sorting the sub-queries, we resolve *foaf:name* first, then *rdf:age*, and finally *rdf:type*.

6. RESOLVING RDQL QUERIES

RDQL [10] is a query language for RDF proposed by the developers of the popular Jena Java RDF toolkit [8]. RDQL operates at the RDF triple level, without taking RDF Schema information into account (like RQL [7] does) and without providing inferencing capabilities. As such, it is the type of low-level RDF query language that we want RDFPeers to support well. It is our intuition that it is possible to translate all RDQL queries into combinations of the native RDFPeers queries above; however, we have not yet written such a translator and it may be inefficient for some queries, especially for joins. This section informally describes how the example RDQL queries from the Jena tutorial (http://www.hpl.hp.com/semweb/doc/tutorial/RDQL) would be resolved.

```
(1) SELECT ?x WHERE  (?x, <vcard:FN>, "John Smith")
(2) SELECT ?x, ?fname WHERE  (?x, <vcard:FN>, ?fname)
(3) SELECT ?givenName
    WHERE  (?y, <vcard:Family>, "Smith"),
           (?y, <vcard:Given>, ?givenName)
(4) SELECT ?resource
    WHERE (?resource, <inf:age>, ?age) AND ?age>=24
(5) SELECT ?resource, ?givenName
    WHERE (?resource, <vcard:N>, ?z),
          (?z, <vcard:Given>, ?givenName)
(6) SELECT ?resource, ?familyName
    WHERE (?resource, <inf:age>, ?age),
          (?resource, <vcard:N>, ?y),
          (?y, <vcard:Family>, ?familyName) AND ?age>=24
```

Query (1) translates directly into Q4, so that it can be resolved in $\log N$ routing hops in a network of N nodes. Similarly, query (2) translates directly into Q3, taking $\log N$ hops. To resolve query (3) , we first issue a Q4-style query and then use its query result as constraint to issue a Q9-style disjunctive query with Q3-style triple patterns. Since all the predicate values in the two triple patterns are known, these two native queries can be resolved in $2 \times \log N$ hops. Query (4) is a typical Q9-style range query with the constraint on the object value. Since its predicate value is known, we can route the query to the node which stores the triples with predicate *inf:age* in $\log N$ hops.

Our native queries do not include join operations, so that we decompose join queries into multiple native queries. Query (5) can be resolved via two Q3-style queries, and by then joining the first triple set's object with the second triple's subject, $2 \times \log N$ routing hops. (However, note that these two Q3-style queries might generate large-size messages if the predicates *vcard:N* or *vcard:Given* are popular.). Query (6) can be resolved by first issuing the same query as for the previous RDQL example for the first triple pattern. Then we use the query result as a constraint for variable *?resource* and resolve the second triple pattern as a Q9-style disjunctive range query. Finally, we use the second query result as a constraint for variable *?y* and again resolve the third triple as a Q9-style query, which in the aggregate takes $3 \times \log N$ hops.

7. IMPLEMENTATION AND EVALUATION

We implemented the MAAN layer of RDFPeers in Java and measured its performance on a real-world network of up to 128 nodes in a previous paper [4]. We also measured the number of neighbors per node against the network size. Similar to Chord, the number of neighbors at each node increases logarithmically with the network size, so that the node state in MAAN scales well to a large number of nodes; for example, in a hypothetical network of eight billion nodes (one for each human on Earth) each node would maintain just thirty-three IP connections. We measured the number of routing hops against the network size for both exact-match queries and for range queries. The experiment results show that for exact-match queries, the number of routing hops in the worst case is $O(\log N)$ and the average routing hops is $\log N/2$. However, for range queries whose selectivity $s_i > \varepsilon\%$, meaning that they select more than one node, the routing hops increase linearly with network size. This is optimal in the sense that s_i of total N nodes have to be visited by the search queries presuming we want to evenly balance the load to the nodes. We implemented an RDF/XML triple loader based on the Jena Toolkit 2.0 to measure the number of routing hops in a simulation (measuring the query cost); we also studied the number

of triples stored per node by loading real-world RDF data into our simulator (measuring the storage cost).

7.1 Routing Hops to Resolve Native Queries

The number of routing hops taken to resolve a query is the dominant performance metric for P2P systems. Figure 3 shows our simulation result for atomic triple patterns from 1 node to 8192 nodes on a logarithmic scale, which matches our theoretical analysis.

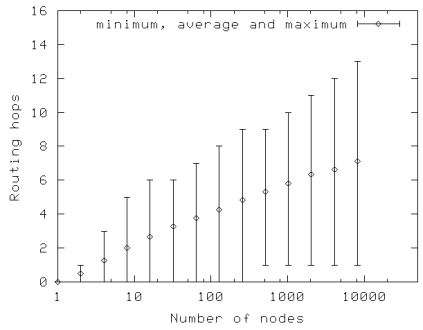

Figure 3: The number of routing hops to resolve atomic triple patterns Q2 through Q8.

We also compared two disjunctive range query resolution algorithms: the simple algorithm vs. range ordering algorithm. Figure 4 shows the simulation result for up to 1000 disjunctive exact-match values ($s_i = \varepsilon\%$) in a network with 1000 nodes.

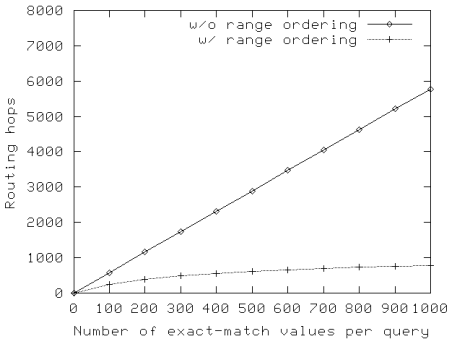

Figure 4: The number of routing hops to resolve disjunctive exact-match queries in a network with 1000 nodes.

Figure 5 shows the result for up to 1000 disjunctive ranges with 0.1% selectivity each in the same network. From these two experiments, we can see that the range ordering algorithm takes less routing hops to resolve a range query than the simple algorithm, and that its routing hops are indeed bounded by N.

7.2 Dealing with Overly Popular URIs and Literals

Even today's cheapest PCs have a surprising storage capacity, each can store well over ten million RDF triples by dedicating 10 Gigabytes of its typical 80-120 GB disk. Nevertheless, some triples in RDF such as those with the predicate *rdf:type* may occur so frequently that it becomes impossible for any single node in the network to store all of them. That is, in practice, triples may not hash around the Chord identifier circle uniformly due to the non-uniform frequency count distribution of URIs and literals. Figure 6 shows the frequency count distribution of the URIs and

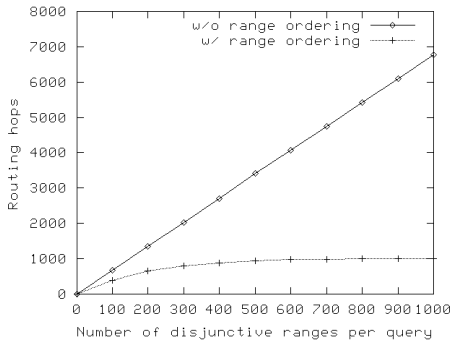

Figure 5: The number of routing hops to resolve disjunctive range queries (0.1% selectivity) in a network with 1000 nodes.

Literals in the RDF dump of the "Kids and Teens" catalog of the Open Directory Project (http://rdf.dmoz.org). There are two RDF files for this catalog: *kt-structure.rdf.u8.gz* and *kt-content.rdf.u8.gz*. The former describes the tree structure of this catalog and contains 19,550 triples. The latter describes all the sites in this catalog and contains 123,222 triples. Figure 6 shows that only 10 to 20 URIs and literals (less than 0.1%) occur more than a thousand times.

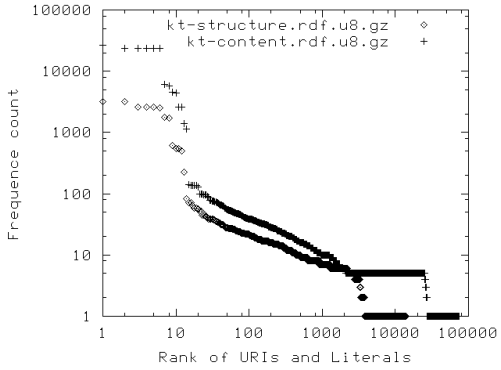

Figure 6: The frequency count distribution of URIs and literals in the ODP Kids and Teens catalog.

Table 2 lists the URIs and literals which occur more than 1000 times in *kt-structure.rdf.u8.gz*. For example, since each URI as a predicate value will be stored at only one node, this node has the global knowledge about the frequency count of this predicate value.

We deal with predicate values that become overly popular by simply no longer indexing triples on them. Each node defines a *Popular_Threshold* parameter based on its local capacity and willingness (subject to some minimum community expectation). Each node keeps counting the frequency of each predicate value. If a predicate value occurs more than *Popular_Threshold* times, the node will refuse to store it and internally makes a note of that. If the node receives a search request with the overly popular value for the predicate, it sends a refusal message back to the requesting node and the requesting node must then find an alternative way of resolving the query by navigating to the target triples though either the subject or object values. This approach will add $O(logN)$ to that node's total query cost in hops. We limit subject and object values in the same way. We are aware that this still makes the node with popular URIs a hotspot for query messages which can be addressed by querying nodes caching which queries were refused in

the past. In essence, this means that you cannot ask e.g. "which instances in the world are the subclass of some class". However, these queries are so general and would return so many triples that we suspect they would rarely be of use in practice anyway (in analogy to the English language, where the words "a" and "the" occur frequently but provide little value as search terms). For the above query, you could alternatively gather the class URIs for which you want to look for instances for, then traverse to the instances via that set of URIs by issuing a Q4-style query.

Frequency	URI or literal	Type
3158	rdf:type	predicate
3158	dc:Title	object
2612	http://dmoz.org/rdf/Topic	object
2612	http://dmoz.org/rdf/catid	predicate
2574	http://dmoz.org/rdf/lastUpdate	predicate
2540	http://dmoz.org/rdf/narrow	predicate
1782	http://dmoz.org/rdf/altlang	predicate
1717	dc:Description	object

Table 2: URIs and literals that occur more than one thousand times in *kt-structure.rdf.u8.gz*

Figure 7 shows the minimum, average, and maximum number of triples per node with *Popular_Threshold* from 500 to 32,000. In this experiment, we store both *ktstructure.rdf.u8.gz* and *ktcontent.rdf.-u8.gz* (total 142,772 triples) into a network of 100 physical nodes (and the standard Chord log(100)=6 virtual nodes per physical node for trading off load balancing against routing hops). When *Popular_Threshold*=32,000, there are no overly popular URIs or literals being removed and there is an average of 4303 triples per node. However, the load is unevenly balanced – the minimum number of triples per node is 700 while the maximum number of triples per node is 36,871. When *Popular_Threshold* is set to 500, there are 20 overly popular URIs and literals being removed from indexing and there are an average of 2352 triples per node. The minimum number of triples per node is 688 while the maximum number of triples per node is reduced to 4900 – which we believe at less than an order of magnitude difference is acceptable load balancing.

7.3 Load Balancing via Successor Probing

Although limiting overly popular URIs and literals greatly reduces the difference between the maximum and minimum number of triples per node, the triples are still not uniformly distributed around all nodes. This is because the frequency count distribution of non-popular URIs and literals remains non-uniform even after removing overly popular values. We propose a preliminary *successor probing* scheme inspired by the "probe-based" node insertion techniques of [5] to further achieve a more balanced triple storage load on each node. In Chord, the distribution of node identifiers is uniform and independent of the data distribution. In this successor probing scheme, we use a sampling technique to generate a node identifier distribution adaptive to the data distribution. When a node joins the network, it will use SHA1 hashing to generate *Probing_Factor* candidate identifiers. Then it uses Chord's successor routing algorithm to find the successors corresponding to these identifiers. All the successors will return the number of triples which would be migrated to the new node if it joined there, and the new node will choose the identifier that gives it the heaviest load. The cost of this technique is that it increases the insertion time of a triple from $\log N$ to $Probing\_Factor \times \log N$. It is our intuition that $\log N$ is a good setting for the probing factor. Figure 8 shows the minimum, average and maximum number of triples per node with *Probing_Factor* from 1 to 9 in a network with 100 physical nodes. The *Popular_Threshold* is set to 1000 in this experiment. If there is no successor probing, the most loaded node has 7.2 times more triples than the least loaded node. If each node probes 9 nodes when it joins, the node with the heaviest load only has 2.6 times more triples than the node with the lightest load – which further reduces load imbalances to much less than an order of magnitude. We can further improve load balancing with a background virtual node migration scheme proposed in [14], subject to the limitation that it cannot distribute the load for a single overly popular value.

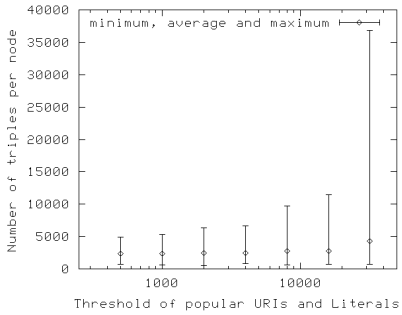

Figure 7: The number of triples per node as a function of the threshold of popular triples (100 physical nodes with 6 virtual nodes per physical node).

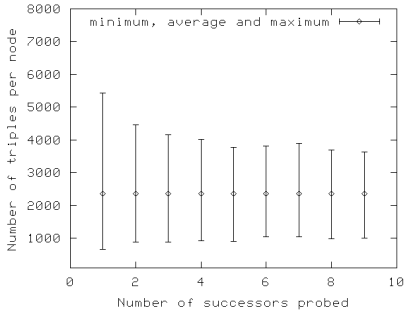

Figure 8: The number of triples per node as a function of the number of successor nodes probed (100 physical nodes, *Popular_Threshold*=1000).

8. RELATED WORK

Many centralized RDF repositories have been implemented to support storing, indexing and querying RDF documents, such as RDFDB [19], Inkling [9], RDFStore [3] and Jena [8]. These centralized RDF repositories typically use in-memory or database-supported processing, and files or a relational database as the back-end RDF triple store. RDFDB supports a SQL-like query language, while Inkling, RDFStore and Jena all support SquishQL-style RDF query languages. Centralized RDF repositories are very fast and can scale up to many millions of triples. However, they have the same limitations as other centralized approaches, such as a single processing bottleneck and a single point of failure. Edutella [12] and its successor super-peer-based RDF P2P network [13] were discussed in Section 1. Super-peers are often desirable in order to place the load unevenly among heterogeneous nodes, but our

scheme can achieve the same effect more flexibly by nodes hosting more or fewer Chord virtual nodes according to their capacity. Recent structured P2P systems use message routing instead of flooding by leveraging a structured overlay network among peers. These systems typically support distributed hash table (DHT) functionality and offer the operation *lookup (key)*, which returns the identity of the node storing the object with the key [16]. Current proposed DHT systems include Tapestry [23], Pastry [18], Chord [22], CAN [15], and Koorde [6]. In these DHT systems, objects are associated with a key which can be produced by hashing the object name. Nodes have identifiers which share the same space as the keys. Each node is responsible for storing a range of keys and corresponding objects. The DHT nodes maintain an overlay network with each node having several other nodes as neighbors. When a *lookup (key)* request is issued from one node, the lookup message is routed through the overlay network to the node responsible for the key. Different DHT systems construct different overlay networks and employ different routing algorithms. They can guarantee to finish lookup in $O(\log N)$ or $O(dN^{1/d})$ hops and each node only maintains the information of $O(\log N)$ or d neighbors for a N nodes network where d is the dimension of the hypercube organization of the network. Therefore, they provide very good scalability as well as failure resilience. However, these DHT systems only provide single key based lookup and do not efficiently support multi-attribute and range queries, nor do they provide RDFPeers' triple storage load balancing.

9. FUTURE WORK AND CONCLUSION

We would like to perform further experiments measuring cost in terms of message sizes rather than just in the routing hops that are the customary P2P performance metric, and to implement the RDQL-to-RDFPeers-Native-Queries translator that we have only sketched in this paper. We would like to be able to bound the size of initital query results, so that e.g. only the first one-hundred matches are returned with the note "there are roughly 4,000 more matches, would you like to retrieve them?". We have not yet implemented triple deletions in RDFPeers, only triple insertions. We would also like to improve load balancing using a background virtual node migration scheme, and to add (binary and other non-meta-) data storage support on top of RDFPeers.

In conclusion, RDFPeers advances the state of the art of P2P RDF systems by guaranteeing that query results will be found if they exist, by not requiring up-front schema definition, by not relying on super-peers, and by balancing the triple-storing load between the most and least loaded nodes. Its storage cost in neighborhood connections is logarithmic to the number of nodes in the network, and so is its processing cost in routing hops for all insertion and most query operations, thus enabling distributed RDF repositories of truly large numbers of participants.

10. ACKNOWLEDGEMENTS

The successor probing technique of Section 7.3 was inspired by discussions with Shahram Ghandeharizadeh and Antonios Daskos about load balancing techniques. We gratefully acknowledge feedback from the anonymous reviewers, Stefan Decker, and Geoff Pike, and AFOSR funding under grant F49620-01-1-0341.

11. REFERENCES

[1] http://www.w3.org/RDF. World-Wide Web Consortium: Resource Description Framework.

[2] http://www.w3.org/TR/rdf-schema. World-Wide Web Consortium: RDF Schema.

[3] RDFStore. http://rdfstore.sourceforge.net.

[4] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A multi-attribute addressable network for grid information services. In *4th Int'l Workshop on Grid Computing*, 2003.

[5] S. Ghandeharizadeh, A. Daskos, and X. An. PePeR: A distributed range addressing space for P2P systems. In *Int'l Workshop on Databases, Information Systems, and Peer-to-Peer Computing (at VLDB)*, 2003.

[6] F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal hash table. In *2nd Int'l Workshop on P2P Systems*, Feb. 2003.

[7] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *11th World Wide Web Conference*, 2002.

[8] B. McBride. Jena: Implementing the RDF Model and Syntax specification. In *2nd Int'l Semantic Web Workshop*, 2001.

[9] L. Miller. Inkling: RDF query using SquishQL. http://swordfish.rdfweb.org/rdfquery.

[10] L. Miller, A. Seaborne, and A. Reggiori. Three implementations of SquishQL, a simple RDF query language. In *First Int'l Semantic Web Conference*, 2002.

[11] National Institute of Standards and Technology. Publication 180-1: Secure hash standard, 1995.

[12] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. S. A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P networking infrastructure based on RDF. In *11th World Wide Web Conference*, 2002.

[13] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Lser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *12th World Wide Web Conference*, May 2003.

[14] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured P2P systems. In *2nd Int'l Workshop on P2P Systems*, 2003.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *ACM SIGCOMM*, 2001.

[16] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. In *2nd Int'l Workshop on P2P Systems*, Feb. 2003.

[17] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[19] R.V.Guha. rdfDB : An RDF database. http://guha.com/rdfdb.

[20] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, Jan. 2002.

[21] S. Sen and J. Wong. Analyzing peer-to-peer traffic across large networks. In *ACM SIGCOMM Workshop on Internet Measurement*, Nov. 2002.

[22] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[23] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-01-1141, UC Berkeley, 2001.