2009 International Conference on Advanced Information Networking and Applications

Building Distributed Index for Semantic Web Data

Juan Li
Computer Science Department
North Dakota State University
j.li@ndsu.edu

Abstract— **With the rapid growth of Semantic Web, more and more Semantic Web data are generated and widely used. Efficient search and management of the Semantic Web data is becoming a very important issue. In this paper, we present Semantic Web data indexing scheme over a fully decentralized P2P network. Ontological knowledge is decomposed into atomic elements and then indexed with a Distributed Hash Table (DHT) overlay. Ontology reasoning, integration, and searching are all based on the index. A complex SPARQL query can be evaluated by performing relational operations such as select, project, and join on combinations of the atoms. A set of experiments have been made to show the effectiveness and efficiency of the proposed indexing scheme.**

Keywords-index; Semantic Web; overlay network; ontology

I. INTRODUCTION

The Semantic Web is an evolving extension of the World Wide Web (WWW) in which Web content can be expressed not only in natural language, but also in a format that can be read and used by software agents, thus permitting them to find, share and integrate information more easily [1,2]. Proposed by Tim Berners-Lee, inventor of the Web and HTML, the Semantic Web is his vision the future of the WWW [3]. The Semantic Web relies on ontologies that structure underlying data for the purpose of comprehensive and transportable machine understanding. Ontology is defined as "an explicit specification of a conceptualization" [4]. It is (meta)data schema, providing a controlled vocabulary of concepts, each with an explicitly defined and machine-processable semantics. By defining shared and common domain theories, ontologies help both people and machines to communicate concisely, supporting the exchange of semantics and not only syntax.

To publish and retrieve Semantic Web data, the World Wide Web Consortium (W3C) developed several recommendations: the Resource Description Framework (RDF [5,6]), the Web Ontology Language (OWL [7]), and the SPARQL query language [8]. RDF is a data model for information representation and exchange on Semantic Web. RDF makes statements about resources in the form of *subject-predicate-object* expressions, called *triples* in RDF terminology. The *subject* denotes the resource which has a Universal Resource Identifier (URI). The predicate denotes traits or aspects of the resource and *expresses* a relationship between the subject and the object. The *object* is the actual value, which can either be a resource or a literal.

OWL is a Web ontology language which is used to publish and share explicit and common descriptions of domain knowledge and provide support for efficient knowledge management. OWL has three increasingly-expressive sub-languages: OWL-Lite, OWL-DL, and OWL-Full, each geared toward fulfilling different application requirements. Description Logics (DL) is the logic foundation of the OWL. DLs are typically a decidable subset of First Order Logic, and are tailored towards Knowledge Representation (KR) [9]. They are suitable for representing structured information about concepts, concept hierarchies and relationships between concepts. All varieties of OWL use RDF for their syntax.

SPARQL is an RDF query language providing a graph pattern matching based paradigm for flexible RDF data graphs. Theoretically, graph pattern matching is more expensive than SQL and XQuery/XPath evaluation [10].

With the mature of Semantic Web technologies, more and more semantic web data are generated and widely used in Web applications and enterprise information systems. To fully utilize the large amount of semantic data, an effective indexing mechanism customized for semantic web data, especially for ontologies, is needed by human users as well as software agents and services. However, the above mentioned unique semantic features and the inherent distributed nature of semantic web data make its storage and retrieval highly challenging.

In this paper, we describe a distributed index structure, in which semantic web data indexes are distributed among a fully decentralized P2P overlay. This permits good scalability as storage and accessing load are distributed over all participants. The ontological framework is based on an efficient P2P indexing system that indexes the dispersed resource ontology knowledge with a decentralized DHT overlay. Ontological knowledge is decomposed into atomic elements and then indexed with DHTs. Ontology reasoning, integration, and searching are all based on the index. SPARQL queries can be evaluated by performing relational operations such as *select*, *project*, and *join* on combinations of the atoms. A key advantage of this ontological indexing scheme is its ability to index in different granularities, as we distinguish knowledge in different levels of abstraction.

The rest of this paper is organized as follows. In section 2, we describe how to publish the metadata information in different granularities on a DHT overlay. In Section 3, we illustrate how to solve complex SPARQL queries with examples. We report a performance and scalability evaluation in Section 4. In Section 5, we conclude this paper.

II. Distributed indexing scheme

The main purpose of an index is to reduce the number of direct accesses to the data while searching. Given a large-scale distributed semantic data repository, it is infeasible to do a thorough search. The indexing on the distributed repositories speeds up the searching process by only pushing down queries to information sources we can expect to contain an answer. For this purpose, we use a DHT-based P2P network which implements a distributed ontology repository for storing, indexing and querying semantic ontology data. The index corresponds to the queries used to retrieve the data. Since Semantic Web is represented with RDF syntax and complex structures can be easily encoded in a set of RDF triples, our indexing is based on RDF triples in the format of (*spo*), where *s* is the subject, *p* is the predicate and *o* is the object.

As mentioned, Description Logics (DL) is the logic foundation of the OWL.A DL knowledgebase includes a Terminology Box (T-Box) and an Assertion Box (A-Box). The T-Box is a finite set of terminological axioms, which includes all axioms for concept definition and descriptions of domain structure, for example a set of classes and properties. The A-Box is a finite set of assertional axioms, which includes a set of axioms for the descriptions of concrete data and relations, for example, the instances of the classes defined in the T-Box. T-Box statements tend to be more permanent within a knowledge repository. In contrast, A-Box statements are much more dynamic in nature. Generally speaking, there are many more A-Box instances than T-Box concepts.

One advantage of our ontological indexing is its ability to handle different granularities. We distinguish T-Box knowledge and A-Box knowledge in each peer's local repository as distinguishing between schema information and the data themselves. In this way, indices can be created based on these two types of knowledge. By the combination of these two indexing schemes an application on top can choose which scheme fits the needs of the system best. The system will be able to scale to hundreds of thousands of nodes and to large amounts of ontology data and queries.

A. A-Box indexing

The purpose of A-Box indexing is to index individual instance information so that the right instance can be efficiently located. We employ RDFPeer's indexing method presented by M. Cai *et al* [11]. The basic idea is to divide RDF description into triples and index the triples in a DHT overlay. We store each triple three times by applying a hash function to its *subject*, *predicate*, and *object*. In this way, a query providing partial information of a triple can be handled. The insertion operation of a triple *t* is performed as follows:

Insert(t) ≡Insert(SHA1Hash(t.subject), t),
Insert(SHA1Hash(t.predicate), t),
Insert(SHA1Hash(t.object), t)

For example, the statement
t: {< Billy>, <teaches>, < cs213 >} is first indexed by *subject*, and sends the following message to the overlay:

Insert {key, {("subject", <Billy>),
("predicate", <teaches >),
("object", < cs213>)}}
where key=SHA1Hash("< Billy> ")

In the message, the first attribute-value pair *("subject", < Billy>)* is the routing key pair, and key is the SHA1 hash value of the *subject* value. Similarly, the triple is indexed by *predicate* and *object* as well. The target DHT node stores the assertion and possibly generates new assertions by applying the entailment rules. These new assertions have to be sent out to other nodes. For example, transitive properties, such as *ancestorsOf*, will have a chaining effect. Thus, after finishing this process, the entire set of A-Box knowledge is accessible in a well-defined way over the community overlay. With this indexing scheme, triples can be retrieved from the DHT by fixing one part of the triple and using this part as a retrieval key.

A-Box indexing keeps each instance triple, thus queries can be accurately forwarded to the instance level. Applications with large storage requiring fast query responses would consider using A-Box indexing. The downside of indexing A-Box information is that the oversized indices of individual instances may cause large maintenance overhead, thus making the system hard to scale. Moreover, in many cases it would not even be applicable to index A-Box knowledge, e.g., when sources do not allow replication of their data (which is what instance indices essentially do). To solve this problem, we also provide another indexing scheme: T-Box indexing.

B. A-Box indexing

Similar to a database schema, a node's T-Box knowledge is more abstract, describing the node's high-level concepts and their relationships. Basically, the T-Box knowledge includes class elements and property elements. It also adheres to the triple (*spo*) format, while here the subject *s* is the class (or property) in question, *p* is the predefined OWL predicates describing the attribute of this class (property), and *o* is the value of the attribute of related class (property). Below is an example of a simple T-Box ontology describing a simple teaching relationship in the triple format.

@prefix univ: <http:/www.cs.ubc.ca/~juanli/univ#>
< univ:Teacher>,< rdf:type>,<owl: class>
< univ:Teacher>, <rdfs:subClassOf>, < univ:People>

< univ:Course>, < rdf:type>, <owl:class>

<univ:teach>,<rdf:type>,<owl:InverseFunctionalProperty>
< univ:teach>, <rdfs: domain>, < univ:Teacher">
< univ:teach>, <rdfs:range>,< univ:Course">
< univ:teach>,<owl:inverseOf>,< univ:isTaughtBy">

<univ:isTaughtBy>,<rdf:type>,<owl:InverseFunctionalProperty>
< univ:isTaughtBy>, <rdfs: domain>, < univ:Course>
< univ:isTaughtBy>, <rdfs:range>, < univ:Teacher>
< univ:isTaughtBy>, <owl:inverseOf>, < univ:teach>

The T-Box definition is indexed in the triple format as well. Classes and properties of the T-Box are indexed separately. The indexing process is the same as A-Box indexing – storing each triple three times by the *subject*, *predicate*, and *object* respectively. The three parts of the T-Box triple are uneven: a T-Box has only a limited number of predefined *predicates*, but many more *objects* and *subjects*. For example, many classes have a *subclass* property; each is encoded as a triple with *predicate rdf:subClass*. When indexing by the *predicate*, all these triples are mapped to the same key and therefore to the same peer in the network. This causes overloading of the peer in charge of the key. This problem can be solved by simply not indexing the overly popular keys; the query can be resolved by using other information of the triple.

Storing the T-Box definition is only part of the indexing task. In a VO, many nodes may use the existing T-Box instead of defining their own. Therefore, another task of T-Box indexing is to link the T-Box triples with nodes using them. This is done by extracting the T-Box concepts from a node's ontology, and then using them as the key and the node's Id as the value to do indexing. Table 1 shows an example T-Box index table maintained by a peer.

TABLE I. AN EXAMPLE T-BOX INDEX TABLE STORED IN A NODE

Concept	Peers Involved	Related T-Box triple
OS	n_2,n_{14}, n_{31},n…	*<OS><superClasss><UNIX>*
		<OS><equivalentClass><OperatingSystem>
PC	n_7, …	*<PC><referentialClass><Computer>*
Unix	n_5, n_2	*<OS><superClasss><UNIX>*
CPU	n_{14}, n_5	*<Processor>< equivalentClass ><CPU>*
own	n_{11}, n_{53}	*<ownedBy><inverseProperty><own>*
run	n_2, n_{14}, n_5 n_{12}, n_{23}, …	*<run><equivalentProperty><execute>*
		<run><domain><Computer>
		<run><range><OS>

T-Box indexing only stores the schema information but ignores the individual instances. It has two functionalities: it helps answering knowledge schema queries; it also helps filtering the candidate result set for individual instance queries. Compared to the instance-level A-Box indexing, T-Box indexing does not require creating and maintaining oversized indices since there are far fewer concepts than instances. The down side of keeping only the schema information is that query answering without the index support at the instance level is much more computationally intensive. Obviously, there is a tradeoff between query overhead and indexing overhead. When the system has a high requirement for fast and efficient query answering, it has to pay more for the indexing. On the other hand, if the system does not index the detailed knowledge, it has to explore more nodes in searching for query results. An application should determine the right indexing granularity that can trade off the cost of maintaining the index against the benefit that the index offers for queries.

III. QUERY EVALUATION

Having introduced the metadata indexing scheme, we now turn our attention to how to utilize the index. Particularly this section describes how to combine lookup operations from different indexes to process queries. Our system supports SPARQL query language. The query evaluation process begins with the parsing of a user's query to SPARQL format. Then the query in terms of relations in the user's local ontology will be translated into sub-queries using the semantic mapping axioms indexed into the overlay. Then each of the sub-queries can be executed at different sources in parallel and the query engine can collect returned answers from the sources and combine them as the answer to the query. This process is illustrated in Fig. 1, in which we assume the underlining indexing is based on A-Box. Searching based on T-Box indexing is similar and is studied in Section 3.2.

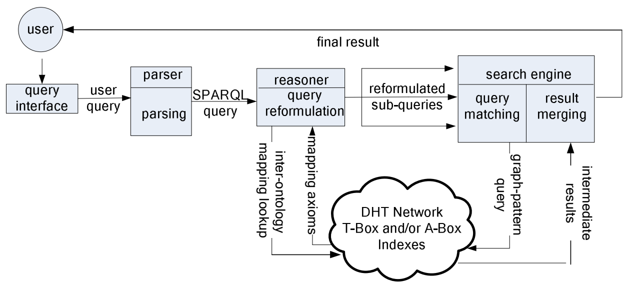

Figure 1. Query processing

A. *Processing SPARQL queries*

The building block for SPARQL queries is graph patterns which contain triple patterns. Triple patterns are RDF triples, but with the option of query variables in place of RDF terms in the *subject*, *predicate*, or *object*. A solution to a SPARQL graph pattern with respect to a source RDF graph *G* is a mapping from the variables in the query to RDF terms such that the substitution of variables in the graph pattern yields a sub-graph of *G* [32]. More complex SPARQL queries are constructed by using *projection* (SELECT operator), *left join* (OPTIONAL operator), *union* (UNION operator), and *constraints* (FILTER operator) [12]. The semantics for these operations are defined as algebraic operations over the solutions of graph patterns [13].

1) Single triple pattern. The simplest query is to ask for resources matching one triple pattern. To illustrate how to perform this kind of simple SPARQL queries, imagine a query to discover the person who teaches course cs213. In SPARQL this query could be written as:

```
PREFIX sample:<http://www.cs.ubc.ca/~juanli/#sample >
SELECT ?person
WHERE {
   ?person sample:teach sample:cs213
}
```

In this query pattern, there is only one triple pattern and at least one part of the triple is a constant. Since we store each triple three times based on its hashed *subject*, *predicate*, and *object* values, we can resolve the query by routing it to the node responsible for storing that constant. Then the responsible node matches this triple against the patterns stored locally and returns results to the requesting node. In this example, there are two constants in the triple pattern; the query processor can use either of them as the DHT lookup key. For example, we can hash on the *object*: *sample:cs213*, then use it as the key to route the query. The node in charge of this key in the DHT overlay matches triples indexed locally using this pattern, and sends back the matched triples.

2) Conjunctive patterns. When the graph pattern is more complex containing multiple triples, or the query contains a group graph pattern, then each triple pattern will be evaluated by one or two different nodes. These nodes form a processing chain for the query. The first triple pattern is evaluated at the first node, the result is then sent to the next node for further processing. This process continues until the last triple pattern is processed. An alternative approach is to process patterns in parallel, and all results are sent to one node to do the final processing. A system should choose the appropriate approach according to its application. In our example, we use the sequential approach since sequentially joining intermediate results saves the traffic for transferring large amounts of unrelated data. The sequence to evaluate the triple patterns is crucial. Many database researchers have worked on it [14, 15]. Here, for simplicity, we assume that we evaluate the query with the original triple pattern order, in which adjacent triple patterns share at least one common variable.

For a query *q* that has *k* conjunctive triple patterns *(t_1, t_2, ...t_k)*, the query evaluation proceeds as follows: First, t_1 is evaluated using the single triple pattern processing method mentioned previously. The result is projected on the variables with values that are needed in the next query evaluation. Then the query together with the next triple sequence number and the intermediate result is sent to the node responsible for the next triple pattern. When a node n_i receives the query request, n_i evaluates the *i*-th triple pattern t_i of the query using its local triple index and the intermediate result from previous nodes. Then n_i computes the intermediate result and projects the result on columns that are needed in the rest of the query evaluation (i.e., variables appearing in the triple pattern t_{i+1} of *q*). This is a nested loop join on the common column for the inner relation. The process recursively repeats until the last triple pattern t_k of q is evaluated. Then, the last node n_k simply returns the result back to the querying node. We use an example to explain this process. The query to find authors who write papers in the field of P2P is listed below:

SELECT ?author
WHERE {
?author :create ?paper .
?paper :category ?cat .
?cat :label P2P
}

The query evaluation process is illustrated in Fig. 2. Each event in this figure represents an event in the network, i.e., the arrival of a new query request. The query request consists of three parts: (1) the original query, (2) the triple pattern to be processed in this node, represented with that triple's sequence number in the original query's triple lists, (3) the intermediate result from previous nodes. Initially, the intermediate result is empty (Ø).

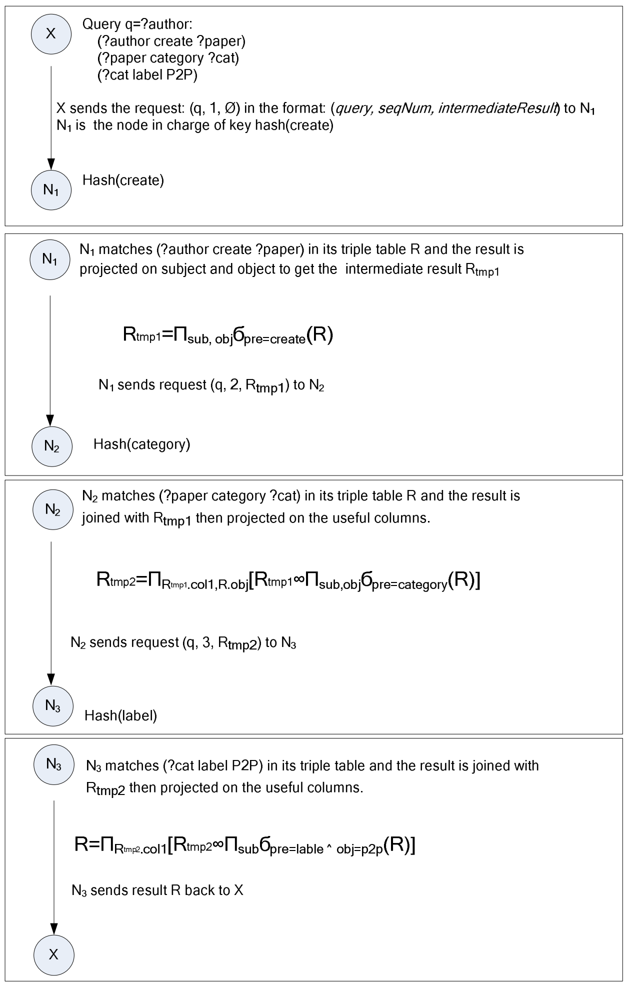
X
Query q=?author:
(?author create ?paper)
(?paper category ?cat)
(?cat label P2P)
X sends the request: (q, 1, Ø) in the format: (query, seqNum, intermediateResult) to N1
N1 is the node in charge of key hash(create)
N1
Hash(create)
N1
N1 matches (?author create ?paper) in its triple table R and the result is projected on subject and object to get the intermediate result Rtmp1
Rtmp1=Πsub, objбpre=create(R)
N1 sends request (q, 2, Rtmp1) to N2
N2
Hash(category)
N2
N2 matches (?paper category ?cat) in its triple table R and the result is joined with Rtmp1 then projected on the useful columns.
Rtmp2=ΠRtmp1.col1,R.obj[Rtmp1∞Πsub,objбpre=category(R)]
N2 sends request (q, 3, Rtmp2) to N3
N3
Hash(label)
N3
N3 matches (?cat label P2P) in its triple table and the result is joined with Rtmp2 then projected on the useful columns.
R=ΠRtmp2.col1[Rtmp2∞Πsubбpre=lable ^ obj=p2p(R)]
N3 sends result R back to X
X

Figure 2. Processing a query with a conjunctive pattern (Results are represented as relational algebras. Π:Projection, б:Selection, ∞:join)

3) Value constraints. A constraint, expressed by the keyword FILTER, is a restriction on solutions over the graph pattern group in which the filter appears. In the simplest case, the value constraint refers only to variables that are bound in the current group and the constraint can be mapped into an equivalent relational expression. In this case the constraint may be applied simply by selecting on the appropriate column. For example, if we have *(?x :age ?y . FILTER(?y > 30))* we need only to select *?y* with value greater than 30.

Sometimes constraints are placed in optional patterns (explained in next section) with variables that do not appear in that block. In this case, since the bindings for that variable are not available at the time the intermediate result is selected, the constraint can be transferred to the results processing step. Alternatively, if available, the bindings for the variables in question can be joined to the intermediate results in which they appear.

4) Optional patterns. SPARQL's OPTIONAL operator is used to signify a subset of the query that should not cause the result to fail if it cannot be satisfied. It is roughly analogous to the left outer join of relational algebra. When processing queries with optional patterns, the intermediate results are produced for each pattern as before, but in the case of an optional pattern, columns that allow joining onto the required pattern must also be projected.

A query with an optional pattern is shown below. A node processes the first pattern, the required pattern, and gets the intermediate result R_{tmp}.(Equation 1). The query together with the intermediate result is sent to another node responsible for the optional pattern, where the optional pattern will be matched with the local triples and the result R_{opt} is outer joined with R_{tmp}.

```
SELECT ?name ?homepage
WHERE { ?person :name ?name .
          OPTIONAL { ?person :homepage ?homepage .}
          }
```

$$R_{tmp} = \pi_{sub}\sigma_{pre=name}(T_1) \qquad (1)$$

$$R_{opt} = \pi_{sub,obj}\sigma_{pre=\mathrm{homepage}}(T_2) \qquad (2)$$

$$R = \pi_{col1,col2}(R_{tmp}{}^{left}\infty R_{opt}) \qquad (3)$$

5) Disjunctive patterns. SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found. Pattern alternatives are syntactically specified with the UNION keyword. Obviously, this kind of disjunctive query could simply be resolved by evaluating each sub-query and then computing the union of the results. For example, for the query listed below, the two sub-queries are sent to different responsible nodes, which then calculate and return the intermediate results to the querying node, where the final result is merged.

```
SELECT ?name ?mbox
WHERE {
          ?person :name ?name .
          { ?person :mbox ?mbox } UNION
          { ?person :mbox_sha1sum ?mbox }
          }
```

B. Query optimization and relaxation

Query optimization should be performed in the query evaluation process to improve the performance. For example, according to the existing research [16], we can rely on algebraic equivalences (e.g., distribution of *joins* and *unions*) to order the evaluation sequence. We may want to separate the *unions* early to parallelize the execution of the *union* in several peers. Additionally, *selects* and *projects* should be pushed down to the lowest possible places, while *joins* should be evaluated closer to the intermediate peers to reduce the size of the result set as early as possible. Furthermore, statistics about the communication cost between peers and the size of expected intermediary query results can be used to decide which peer and in what order will undertake the execution of each query operator. There has been extensive work in query optimization [17, 14]; we can utilize their results in our system.

The matching manager has the task of finding candidate instances that match the specific query constraints, in particular to take into account the concepts, attributes and relationships. It is possible that the descriptions of different ontologies referring to the same real-world object can be significantly different. As a consequence, real-world objects that are meant to be an answer to a query are not returned because their description does not match the query due to insufficient mappings. If a query cannot get enough results because of this high heterogeneity, the matching manager can relax the query constraints by partially matching the query.

C. Query processing based on T-Box indexing

In our previous description of query evaluation, we assume the overlay maintains A-Box indexing. In that scenario, instance triple patterns are indexed in the overlay, and queries for instances can be accurately forwarded to the right peers in charge of the triples. If an application only maintains T-Box indices, the evaluation process is different.

For schema (T-Box) queries on T-Box indexing, the evaluation process is similar to the query evaluation process we just explained, because T-Box indexing is detailed enough to answer the schema query. For example, consider the query pattern:

```
SELECT ?class
WHERE {
          ?class rdfs:subClassOf ?someClass
          :teach rdfs:domain ?someClass}
```

It asks for the subclasses of a class which forms the domain of a property *teach*. The processing of this query is the same as the evaluation of conjunctive queries. The query will first be hashed on property *teach* to find its domain class, which will then be used to resolve the next sub-query.

T-Box indexing cannot be used directly to evaluate queries at the instance level, but it can restrict the query to a small set of nodes which are ontologically related to the query. These nodes have the T-Box knowledge to understand the query, thus are capable of answering the query. When a node issues an instance-level query, the T-Box concepts related to the query are extracted in the form of a keyword list, and these keywords are used as parameters to retrieve the relevant peers. We use an example to explain this process. The query is shown below:

```
SELECT ?author
WHERE {
   ?author rdf:type :Person .
   ?author :name "Juan Li" .}
```

First, the query processor uses the concepts *Person* and *name* as keys to locate all nodes related to these concepts. Then the query is sent to these nodes for further evaluation. This way, the search scope is limited to a number of nodes whose schemas are related to the query, although not all of them can answer the query.

IV. Experiment

Owing to the lack of access to the semantic environment with many nodes, our system performance evaluation falls back to simulations.

A. *Experimental setup*

As it is difficult to find representative real world ontology data, we have chosen to generate test data artificially. Our data does not claim to model real data, but shall rather provide reasonable approximation to evaluate the performance of the system. Ontology data can be characterized by many factors such as the number of classes, properties, and individuals; thus we have generated the test data in multiple steps. The algorithm starts with generating the ontology schema (T-Box). Each schema includes the definition of a number of classes and properties. The classes and properties may form a multilevel hierarchy. Then the classes are instantiated by creating a number of individuals of the classes. To generate an RDF instance triple t, we first randomly choose an instance of a class C among the classes to be the subject: *sub(t)*. A property p of C is chosen as the predicate *pre(t)*, and a value from the range of p to be the object: *obj(t)*. If the range of the selected property p are instances of a class C', then *obj(t)* is a resource; otherwise, it is a literal.

The queries are generated by randomly replacing parts of the created triples with variables. For our experiments, we use single-triple-queries and conjunctive-triple-queries. To create the conjunctive-queries, we randomly choose a property p_1 of class C_1. Property p_1 leads us to a class C_2 which is the range of p_1. Then we randomly choose a property p_2 of class C_2. This procedure is repeated until the range or the property is a literal value or we have created n ($n \leq 3$) triple patterns.

Our dataset uses the following parameters: The total number of distinguished ontologies is 100. We assume each node uses 1 to 3 ontologies. Each ontology includes at most 10 classes. The number of properties that each class has is at most k=3. The number of instances of each class at each peer is less than 10. Finally, the number of triple patterns in each query we create is either 1 or 3.

We implement a simulator of Pastry in Java on top of which we developed our indexing and routing algorithms. Each peer is assigned a 160-bit identifier, representing 80 digits (each digit uses 2 bits) with base b=2. After the network topology has been established, nodes publish their data on the overlay network. Then nodes are randomly picked to issue queries. Each experiment is run ten times with different random seeds, and the results are the average of these ten sets of results.

B. *Experimental results*

We are mainly interested in two different questions, related to three different aspects of the indexing and searching scheme. First, we want to verify the efficiency of answering typical lookup requests. Second, we need to compare the overhead of indexing T-box and A-box as well as the overhead of searching based on these two indexing schemes.

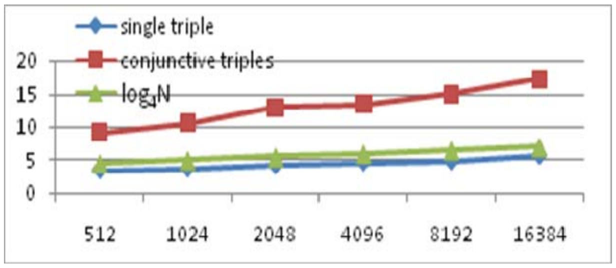

Figure 3. Query lookup efficiency (network size vs. query hops)

The first experiment answers the first question showing the number of routing hops as a function of the size of the Pastry network. We vary the number of Pastry nodes in the network from 2^9 to 2^{14}. We run two trials of experiments: one trial issues only single-triple-queries, while the other trial issues conjunctive-triple-queries. Fig. 3 shows the average number of routing hops taken as a function of the network size for both query patterns. $\log_2 bN$ is the expected maximum number of hops required to route a key in a network containing N nodes (In our experiment b=2), therefore, in the figure "$\log_4 N$" is included for comparison. The results show that the number of route hops scales with the size of the network as predicted: for the single triple query, the route length is below $\log_4 N$; for conjunctive queries, the number of routing hops is below $3\log_4 N$ as expected.

The next experiment compares the performance of the T-Box and the A-Box indexing in terms of indexing overhead and query overhead. Each node may randomly choose n (n<3) ontologies from 100 distinguished ontologies, and instantiate each class with m (m<10) instances. For simplicity, each query uses the simple single triple pattern. With this configuration, we see from Fig. 4 that A-Box indexing incurs much more overhead than T-Box indexing, and the discrepancy increases as the network size increases, for example, A-Box indexing causes several orders of magnitude higher overhead than what TBox indexing creates when the network size is 4096. On the other hand, if the system can afford the cost of maintaining the large index, A-Box indexing can improve searching efficiency. Fig. 5 shows the query overhead in terms of cumulative query messages. It is obvious that with A-Box indexing, processing a query requires much less message forwarding overhead than that based only on T-Box indexing.

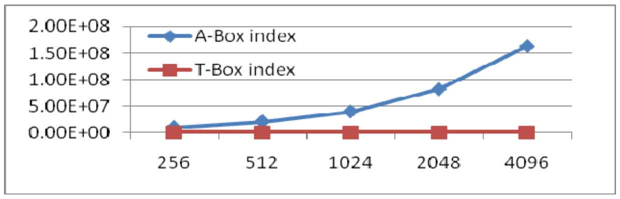

Figure 4. Indexing storage load of T-Box indexing and A-Box indexing(network size vs. cumulative index (bytes))

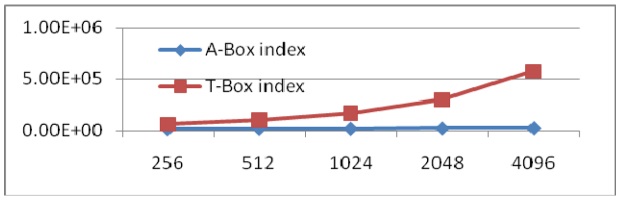

Figure 5. Indexing storage load of T-Box indexing and A-Box indexing (network size vs. cumulative query messages)

We have seen the differences between T-Box indexing and A-Box indexing. For an application, there are many factors to consider for choosing the right indexing scheme, for example, the storage capacity of the participating nodes, the nature of the major queries, and the organizations' policy, and the degree of heterogeneity of the system's ontology.

V. RELATED WORK

Most current research on searching or querying Semantic Web uses an Information Retrieval (IR)-based search engine (e.g. [18–22]). The IR-based work, such as Swoogle [19] and SWSE[20], indexes the Semantic Web by crawling and indexes the Semantic Web RDF documents found online and then offers a search interface over these documents. However, the IR-based semantic data search does not provide structured query capability.

Several groups [10, 23, 24, 25, 26, 27] have developed technology to store RDF nodes, arcs and labels into relational database systems, such as MySQL, Oracle, and DB2, so that Semantic Web data can be efficiently indexed and retrieved. They translate a SPARQL query into SQL statements which are evaluated on the triple store in relational databases. Our SPARQL query evaluation is based on their research result and converts the RDF graph pattern to relational algebra. These DB-based design strikes a careful balance between flexibility, scalability, query facilities, efficiency and optimization. However, taking centralized database servers to support complex queries on web-scale data is still a big challenge. Any server-centered architecture will not only create physical bottlenecks, but as communication relies on the use of ontologies will also create semantic bottlenecks [28].

To address the scalability issue, researchers have utilized P2P technologies to Semantic Web. For example, systems such as Edutella [29] and InfoQuilt [30] use broadcast or flooding to search RDF data, while many other projects, like RDFPeer [11] and OntoGrid [31] attempt applying DHT techniques to the retrieval of the ontology encoded knowledge. Our indexing scheme is based on DHT as well, but it has two unique advantages: (1) it distinguishes semantic data in different levels of abstractions, thus is more scalable and flexible; (2) it provides support for W3C recommended query language SPARQL, thus is easier to be propagated and used.

VI. CONCLUSION

With the growth of the Semantic Web data, there is a strong need to make the semantic information accessible to computer programs that search, filter, and convert information for the benefit of the users. In this paper, we introduced methods for distributed indexing and querying Semantic Web data over P2P network. A key advantage of this ontological indexing scheme is its ability to index in different granularities and support complex SPARQL query evaluation. Queries are processed in a distributed and transparent fashion, so that the fact that the information is distributed across different sources can be hidden from the users. A set of experiments have been made to show the effectiveness and efficiency of the proposed schemes.

REFERENCES

[1] Berners-Lee, Tim & James Hendler and Ora Lassila (May 17, 2001), "The Semantic Web", Scientific American Magazine.

[2] "W3C Semantic Web Frequently Asked Questions". W3C.

[3] Herman, Ivan (2008-03-07). "Semantic Web Activity Statement". W3C.

[4] T. R. Gruber, "Principles for the Design of Ontologies Used for Knowledge Sharing." International Journal Human-Computer Studies, 43(5-6):907-928, 1995.

[5] F. Manola and E. Miller. RDF primer. W3C recommendation, 2004.

[6] D. Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, 2004.

[7] M. K. Smith, C. Welty, and D. L. McGuinness. OWL web ontology language guide. W3C recommendation, 2004.

[8] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, 2008.

[9] B. Nebel, "Artificial intelligence: A computational perspective." Principles of Knowledge Representation, Stanford, 1996.

[10] Li Ma, Chen Wang, Jing Lu, Feng Cao, Yue Pan, Yong Yu, "Effective and Efficient Semantic Web Data Management over DB2.", SIGMOD'08, June , 2008.

[11] M. Cai, M. Frank, "RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network", in proc of WWW conference, NewYork, USA, May 2004.

[12] E. Sirin and B. Parsia. "SPARQL-DL: SPARQL Query for OWL-DL." In Proceedings of the 3rd OWL Experiences and Directions Workshop, 2007.

[13] J. Perez, M. Arenas, C. Gutierrez: "The semantics and complexity of SPARQL." In Proceedings of the 5th International Semantic Web Conference, 2006.

[14] M. JARKR, AND J. KOCH, "Query optimization in database systems." ACM Comput. Surv. 1984.

[15] T. Sellis, "Multiple-Query Optimization", ACM Transactions on Database Systems, 12(1), pp. 23-52, June 1990.

[16] G. Kokkinidis, L. Sidirourgos, and V. Christophides. "Query Processing in RDF/S-Based P2P Database Systems." In, Semantic Web and Peer-to-Peer. Springer-Verlag, 2006.

[17] J. KING. "QUIST: A system for semantic query optimization in relational databases." In Proceedings 7th International Conference on

Very Large Data Bases (Cannes, France). VLDB Endowment, 510–517. 1981.

[18] Zhang, L., Liu, Q., Zhang, J., Wang, H., Pan, Y., Yu, Y.: Semplore: An IR approach to scalable hybrid query of semantic web data. In: Proceedings of the 6th International Semantic Web Conference, 2007.

[19] Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R.S., Peng, Y., Reddivari, P., Doshi,V., Sachs, J.: Swoogle: a search and metadata engine for the semantic web. In: Proc. of the 13th ACM CIKM Conf. (2004)

[20] Hogan, A., Harth, A., Umbrich, J., and Decker, S. "Towards a scalable search and query engine for the web". In Proceedings of the WWW 2007.

[21] Guha, R., McCool, R., Miller, E.: Semantic search. In: Proc. of the 12th Intl. Conf. on World Wide Web. (2003)

[22] Rocha, C., Schwabe, D., Aragao, M.P.: A hybrid approach for searching in the semantic web. In: Proc. of the 13th Intl. Conf. on World Wide Web. (2004)

[23] Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF Schema. In: Proc. of the ISWC2002

[24] Pan, Z., Heflin, J.: DLDB: Extending relational databases to support semantic web queries. In: Workshop on Practical and Scalable Semantic Systems., 2003

[25] Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient SQL-based RDF querying scheme. In: Proc. of the VLDB 2005. (2005)

[26] Zhou, J., Ma, L., Liu, Q., Zhang, L., Yu, Y., Pan, Y.: Minerva: A scalable OWL ontology storage and inference system. In: Proc. of the ASWC2006.

[27] S. Harris. SPARQL query processing with conventional relational database systems. In International Workshop on Scalable Semantic Web Knowledge Base System, 2005.

[28] S Staab, H Stuckenschmidt Semantic Web and Peer-to-Peer, Springer, 2006

[29] W. Nejdl et al. "EDUTELLA: a P2P Networking Infrastructure Based on RDF". In Proc. of the WWW 2002

[30] M. Arumugam, A. Sheth, and I. B. Arpinar. "Towards peer-to-peer semantic web: A distribuited environment for sharing semantic knowledge on the web." In Proc. of the International World Wide Web Conference 2002 (WWW2002), Honolulu, Hawaii, USA, 2002.

[31] OntoGrid project: http://www.ontogrid.net/

[32] P. Hayes: RDF semantics. W3C Recommendation

[33] http://www.w3.org/TR/owl-semantics/ (2004)