Query Optimization for Ontology-Based Information Integration

Yingjie Li
Department of Computer Science and Engineering, Lehigh University
19 Memorial Dr. West
Bethlehem, PA 18015, U.S.A.
yil308@lehigh.edu

Jeff Heflin
Department of Computer Science and Engineering, Lehigh University
19 Memorial Dr. West
Bethlehem, PA 18015, U.S.A.
heflin@cse.lehigh.edu

ABSTRACT

In recent years, there has been an explosion of publicly available RDF and OWL data sources. In order to effectively and quickly answer queries in such an environment, we present an approach to identifying the potentially relevant Semantic Web data sources using query rewriting and a resource index. We demonstrate that such an approach must carefully handle query goals that lack constants; otherwise the algorithm may identify many sources that do not contribute to eventual answers. This is because the resource index only indicates if URIs are present in a document, and specific answers to a subgoal cannot be calculated until the source is physically accessed - an expensive operation given disk/network latency. We present an algorithm that, given a set of query rewritings that accounts for ontology heterogeneity, incrementally selects and processes sources in order to maintain selectivity. Once sources are selected, we use an OWL reasoner to answer queries over these sources and their corresponding ontologies. We present the results of experiments using both a synthetic data set and a subset of the real-world Billion Triple Challenge data.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems

General Terms

Theory

Keywords

Semantic Web, Information Integration

1. INTRODUCTION

In the Semantic Web, the definitions of resources and the relationship between resources are described by an ontology in order to automatically interpret the resources and retrieve useful information. The resources in the Web are independently generated and distributed in many locations. Due to the decentralized nature of the Semantic Web, it is inevitable that different communities of users or software developers will use their own ontologies to describe semantic data sources. Under such an environment, some Web applications want to integrate the ontologies and their data sources and access them without regard to the heterogeneity and the dispersion of the ontologies and the local systems. In order to support such need, Li et al. proposed an index-based mechanism for ontology-based information integration [9]. According to this method, each RDF and OWL data source in the Semantic Web can be treated as a bag of URIs and Literals. Therefore, a resource index can be created to integrate these sources. Based on this index, by reformulating and converting a conjunctive query into a set of Boolean subgoals, those potentially relevant data sources can be located and selected. However, because the resource index only indicates if URIs or Literals are present in a document, specific answers to a subgoal of the given query cannot be calculated until the sources are physically accessed - an expensive operation given disk/network latency. Furthermore, even if there was no disk/network latency problem, in real world, it is also quite normal that the number of sources related to a subgoal could be so large that it is impossible to load all of them into the reasoner to solve the queries. To solve these issues, in this paper, we present a query optimization algorithm for the query answering of the ontology-based information integration with resource index. Given a set of query rewritings that accounts for ontology heterogeneity, this algorithm incrementally selects and processes sources in order to maintain selectivity. Once sources are selected, we use an OWL reasoner to answer queries over these sources and their corresponding ontologies.

The contributions of this paper are as following:

- Query optimization for an efficient query processing for ontology-based information integration with resource index: we present a flat-structure query optimization algorithm that uses the selectivity of each triple pattern in the subgoals and the structure relation among different query rewritings as the heuristic information to schedule plans for the query processing.
- A number of experiments are also conducted to evaluate the characteristic of our proposed algorithm and demonstrate that the proposed algorithm outperforms the algorithm proposed in [9] on both synthetic data set and real world data set.

The remainder of the paper is organized as follows: In Section 2, we review related work. In Section 3, we describe the query optimization algorithm for ontology-based information integration using the resource index. Section 4 presents the experiments that we have conducted to evaluate the proposed algorithm. Finally, in Section 5, we conclude and discuss future work.

2. RELATED WORK

Currently, there are mainly three areas of work related with our paper: RDF query optimization, query answering over distributed ontologies and Database query optimization.

In the first area, RDF data can be serialized and stored in a database and a SPARQL query can be executed as an SQL join, hence recently a lot of database join query optimization techniques such as creating indexes have been applied to improve the performance of SPARQL queries. In the past couple of years, a lot of works have proposed ways of optimizing SPARQL join queries. The representative systems include C-Store [1], RDF-3X [11], MonetDB [15], Hexastore[19] and YARS2 [7].

Out of these systems, C-Store and MonetDB exploit the fact that typically RDF data has much less number of properties (predicates), thereby vertically partitioning the data for each unique predicate and sorting each partition (predicate table) on subject, object order (creating a subject-object index on each property table). Hexastore and RDF-3X make use of the fact that an RDF triple is a fixed 3-dimensional entity and hence they create all 6-way indexes (SPO, SOP, PSO, POS, OPS, OSP): one for each sorting order of subject, predicate and object. It has been demonstrated that this strategy results in good response time for conjunctive queries. The major disadvantages of both of these approaches are that they rely on centralized knowledge bases and that the indexes (or replication) are quite expensive in terms of space. This is because Hexastore shares common indexes within these 6 indexes, e.g., SPO and PSO share the "*O*" index, without any compression. RDF-3X goes one step further and compresses these indexes as described in their paper. RDF-3X also implements several other join optimization techniques like RDF specific Sideways-Information-Passing, selectivity estimation, merge-joins, and using bloom-filters for hash joins. YARS2 is another native RDF store and query answering system where index structures and query processing algorithms are designed from scratch and optimized for RDF processing. The novelty of the approach proposed by YARS2 lies in the use of multiple indexes to cover different access patterns. In YARS2, the widest coverage of access patterns is achieved for the management of quads in the form of <s, p, o, c>, where c stands for the context of a triple <s, p, o>. Six different indexes are proposed to cover 16 possible access patterns of quads, i.e., each of s, p, o and c can be either a constant or a variable. However, in this way, if more efficient query processing can be achieved, more disk spaces will be also required.

Along with these systems, there are also other systems being developed for RDF data storage and querying, such as, Jena-TDB [12] and Virtuoso [3]. Jena-TDB faces scalability issues while executing queries on very large datasets. Along with these, GRIN [18] focuses more on path-like queries on RDF data, typically which cannot be expressed using existing SPARQL syntax. GRIN is a novel index developed specifically for graph-matching queries in RDF. This index identifies selected central vertices and identifies the distance of other nodes from these vertices. However, it still is not clear how GRIN could be adapted for a distributed context.

In the second area, Stuckenschmidt et al. [16] suggests a global data summary for locating data matching query answers in different sources and the query optimization. However, Stuckenschmidt et al. assumes that all distributed ontologies can be accessed in a uniform way like a global schema. In other words, the heterogeneity of schemas of the distributed ontologies is not considered. Besides, many tasks are concentrated on the mediator. As well as query scheduling, the merge (i.e., join) of all local query results is also executed in the mediator. Thus, when the mediator receives requests for many queries at the same time, the bottleneck on the mediator is inevitable.

The most of research on the query answering over distributed ontologies are based on the P2P architecture. Edutella [10] uses an unstructured P2P network which has no method to route a query to the relevant ontologies. Instead, the query is broadcasted in the entire network. Thus, a huge amount of unnecessary network traffic incurs. As a successor of Edutella, to provide better scalability, Nejdl et al. presents a schema-based query routing strategy in a hierarchical topology using the superpeer concept. Nejdl et al. also suggests a rule-based mediation between two different schemas in order to collect results from many peers using heterogeneous schemas. SomeRDFs [2] supports the semantic mapping between two atomic concepts and between the domain (or range) of a property and a class. Piazza [5] proposes a language (heavily relies on XQuery/XPath) to describe the semantic mapping between two different ontologies. In these works, for distributed query answering, a peer reformulates a query by using the semantic mapping and forwards the reformulated query to another peer related by the semantic mapping. DRAGO [14] focuses on a distributed reasoning based on the P2P-like architecture. In DRAGO, every peer maintains a set of ontologies and the semantic mapping between its local ontologies and remote ontologies located in other peers. A reasoning service is performed by a local reasoner for the locally registered ontologies and the reasoning is propagated to the other peers when the local ontologies are semantically connected to the other remote ontologies. The semantic mapping supported in DRAGO is only the subsumption relationship between two atomic concepts. Besides, it does not support the ABox reasoning. KAONP2P [4] also suggests the P2P-like architecture for query answering over distributed ontologies. KAONP2P supports more extended semantic mapping which describes the correspondence between views of two different ontologies, where each view is represented by a conjunctive query. For the distributed query answering, it generates a virtual ontology including a target ontology to which the query is issued and the semantic mapping between the target and the other ontologies. Then, the query evaluation is performed against the virtual ontology. However, all of these P2P systems have a drawback in that each node must install system specific P2P software, presenting a barrier to adoption.

Another works related with our paper are Hermes [17] and Semplore [20]. Hermes employs Semplore to do query processing. It translates a keyword query provided by the user into a federated query and then decomposes this into sepa-

rate SPARQL queries that are issued to web data sources. A number of indexes are used, including a keyword index, mapping index, and structure index. The most significant drawback to the approach is that it does not account for rich schema heterogeneity (mappings are basically of the subclass/equivalent class variety).

In Database field, query optimization has been extensively studied since the classic work by Selinger et al. [13]. Ideas proposed in [13] are still common practice in relational optimizers: use statistics about the database instance to estimate the cost of a query evaluation plan; consider only plans with binary joins in which the inner relation is a base relation (left-deep plans); postpone Cartesian product after joins with predicate. Krishnamurthy et al. [8] proved that under some circumstance, if sorting is not required, a pipelined evaluation plan is the optimum solution. However, all of these Database query optimization techniques cannot be easily applied into the distributed ontology environment because they mainly deal with those relations stored in one or few physical Database files rather than a large number of physically distributed sources in distributed ontology situation.

3. QUERY OPTIMIZATION

As stated in the introduction, our query optimization algorithm is based on a resource index that is used to integrate the distributed and heterogeneous semantic web ontologies and data sources. Therefore, in this section, we first briefly introduce the resource index for ontology-based information integration, and then discuss our proposed query optimization algorithm in detail.

3.1 Resource Index for Ontology-Based Information Integration

The resource index is first proposed in [9] to integrate the distributed and heterogeneous semantic web resources. Due to limited space, we do not present its details here. Please see [9] for more. Basically, the resource index is an inverted index, where each term is either full URIs of subjects, predicates and objects or a free-text format literal value. The free-text format literal value is used to support queries that involve partial string matches. Formally, for a given document d, the terms contained in d can be expressed as following:

$terms(d) \equiv \{x| < s, p, o > \in d \wedge [x \equiv s \vee x \equiv p \vee (o \in U \wedge x \equiv o) \vee (o \in L \wedge x \in lit - terms(o))]\}$,

where $< s, p, o >$ stands for the triples contained in document d and lit-terms() is a function that extracts terms from literals, and may involve typical IR techniques such as stemming and stopwords. The dictionary of the bag-of-URIs is then $\bigcup_{d \in D} terms(d)$.

Based on the resource index, the potentially relevant semantic web data sources can be selected. The source relevance can be defined as following:

Definition 1. (Source Relevance)Given a conjunctive query Q, a term index I (this index is which terms appear in which data sources), a set of document identifiers U, and an ontology function o, a source u is potentially relevant to Q iff $\exists$ a source function s that conforms to I where $< U, o, s > \models Q\theta$ and $< U, o, s - u > \not\models Q\theta$.

Note, s conforms to I means that $\forall t \in T$, $u \in I(t)$ iff $t \in terms(s(u))$, where the function $terms$ is defined as $terms : ABox \rightarrow T$, and $s - u$ denotes a function f(x) as follows:

$$f(x) = \begin{cases} s(x) & if\ x \neq u \\ \emptyset & otherwise \end{cases}$$

In this definition, the term index is a function $I : T \rightarrow \mathcal{P}(U)$, where $T = C \sqcup P \sqcup R \sqcup L$. Here, we use C to refer to the set of all classes, P to refer to the set of all properties, R to refer to the set of all constant URIs, L to refer to the set of the terms in all literals, and $\mathcal{P}(U)$ to refer to the power set of U. Also, we use o to refer to an ontology function that maps U to a set of ontologies and s to refer to a source function that maps U to a set of data sources. The ontolgoies together with the data sources compose the Semantic Web resources.

In addition, $Q\theta$ is a shorthand for $B_1\theta \wedge B_2\theta \ldots \wedge B_n\theta$ for a given conjunctive query $Q\left(\overline{X}\right)$:- $B_1\left(\overline{X}_1\right), \ldots, B_n\left(\overline{X}_n\right)$ and its substitution θ. The Conjunctive Query Form is defined as following:

Definition 2. (Conjunctive Query Form)A conjunctive query has the form $Q\left(\overline{X}\right)$:- $B_1\left(\overline{X}_1\right), \ldots, B_n\left(\overline{X}_n\right)$ where $\overline{X}$ is a vector of variables and/or individuals and each B_i is a unary or a binary atom representing a concept or role term respectively.

Given a conjuctive query, the query answering algorithm proposed in [9] can find potentially relevant data sources by reformulating the given conjunctive query into a set of subgoal queries, converting them into a set of Boolean queries, and accessing the resource index. However, this algorithm suffers from the following three issues.

• During the query reformulation, each reformulation is independently counted as one subgoal for the original query. Therefore, this process loses the structure relations among the different query subgoals and it cannot scale very well into the real world with big number of query rewritings and big number of sources for each query rewriting.

• Because the resource index only indicates if URIs or Literals are present in a document, specific answers to a subgoal of the given query cannot be calculated until the sources are physically accessed - an expensive operation given disk/network latency.

• Furthermore, this algorithm also does not consier the selectivity of each triple pattern in the query during query evaluation.

Therefore, to deal with the above three issues, we propose a flat-structure query optimization algorithm with a bag of rewritings of the given query as its inputs. In our paper, these rewritings are generated from an *AND-OR* tree. Actually, our flat-structure query optimization algorithm can be generalized to any rewriting generation algorithm as long as the rewritings generated by these algorithms satisfy the format requirement of our proposed algorithm. For this point, we will discuss more in section 3.2.1. For each generated rewriting, we will apply the strategy that the Query Triple Patterns (QTPs) with highest selectivity is always first processed to plan the query execution. In the following discussion, we mainly focus on our query optimization algorithm and its all index accessing operations are over the resource index. To distinguish our proposed algorithm in this paper from the query answering algorithm in [9], we name the latter as non-structure query answering algorithm because it lacks the consideration of selectivity and structure information during the query execution plan.

```
SELECT *
WHERE {
        ?p rdf:type :Person
        ?p :hasLehighID ?n
        ?pap :has-author :jeff-heflin
        ?p :has-paper ?pap
}
```

Figure 1: An example of SPARQL query

3.2 Query Optimization Algorithm

According to Definition 2 in section 3.1, the Conjunctive Query Form basically corresponds to the most common SPARQL query, which is a query language for RDF and gaining importance as semantic data is increasingly becoming available in the RDF format. Basically, each SPARQL query consists of a bag of query triple patterns (QTPs) with join relations and therefore can be viewed a conjunctive triple pattern query. It resembles closely to an SQL query. A typical SPARQL query looks like the one shown in Figure 1. This query shows a join among four QTPs.

The QTPs contained in such queries can be broadly classified into four categories based on their selectivity.

• The first type is highly selective triple patterns with constant constraints. E.g. consider the QTP $<?pap, :has\text{-}author, :jeff\text{-}heflin>$. This QTP is highly selective since the given author ": $jeff\text{-}heflin$" is so specific that the number of sources satisfying it is so small.

• The second type consists of triple patterns that are selective despite having no constant constraints. This is because the predicate itself contained in the QTP is selective. E.g. consider the QTP $<?p, :hasLehighID, ?s>$. Since ": $hasLehighID$" is a specific attribute only for Lehigh people. Therefore, the number of sources satisfying this QTP is also small.

• The third type is not selective because it has neither constant constraints nor selective predicates. E.g. consider the QTP $<?p, :has\text{-}affiliation, ?a>$. The number of sources satisfying this QTP is so large that it is impossible to load all relevant sources to solve them in distributed environment.

• The fourth type is very hard to determine whether it is selective or not. E.g. consider the QTP $<?p, rdf:type, :Person>$. The selectivity of this QTP depends on the selectivity of the class ": $Person$". If it is rarely used in the dataset, then we can conclude that this QTP is highly selective. Otherwise, we will say that it is not selective.

The above classification of QTPs enlightens us one hint to plan the query executions. During this process, we can first evaluate the number of sources of each QTP for each rewriting of the original query by accessing the index, and then select the most selective one with the minimum number of sources to solve. After this step, we can apply the intermediate results generated by first step into other QTPs to determine the new number of sources satisfying each of them. With these results, the new most selective QTP will be then chosen and solved. This process is iteratively executed until the whole query is answered.

Basically, in the above process, the rewriting and selectivity are two factors to direct our query execution. This is why our proposed algorithm is called flat-structure query optimization algorithm.

3.2.1 Query Rewriting

In the query rewriting phase, a given conjunctive query will be reformulated into a bag of conjunctive subqueries. Our query rewriting algorithm is based on the PDMS algorithm [6], which takes as input a query, a set of GAV and LAV views describing the sources and the maps. The algorithm constructs a rule-goal tree: where goal nodes are labeled with atoms of the predicate, and rule nodes are labeled with ontology mappings. During the construction of the rule-goal tree, for each GAV mapping rule, an AND child goal node will be created, and for each LAV mapping rule, an OR child goal node will be created. Thus, the rule-goal tree is basically an AND-OR tree. From the generated AND-OR tree, a set of conjunctive subqueries for the original query can be obtained.

Figure 2 shows an example for a sample conjunctive query. We begin with the query Q, which asks for the persons together with their full-names who have publications affiliated by Lehigh University (lehigh-univ). First, the AND-OR tree for Q should be constructed by using the given mapping rules($r0$ and $r1$). Based on the generated AND-OR tree, four rewritings for Q will be generated. They are shown in the bottom of Figure 2. Each of these four rewritings would be counted as one individual conjunctive query and fed into our following flat-structure query optimization algorithm. Each QTP component of each rewriting query have logical AND relation with other QTPs in the same query.

Here, one key thing we need to highlight is that our flat-structure query optimization algorithm can be generalized to any query rewriting generation algorithm. In other words, as long as the query rewriting generation algorithm generates legal query rewritings for our proposed algorithm, the given conjunctive query can be solved.

3.2.2 Flat-structure Query Optimization Algorithm

As shown before, for a given conjunctive SPARQL query, each QTP contained in this query could have different selectivity. Therefore, we can use this heuristic information to select and process sources, and to direct our query execution. We begin our discussion of this query optimization algorithm by introducing some preliminary notions: variable bindings (Definition 3), join conditions (Definition 4), join evaluation (Definition 5) and QTP selectivity (Observation 1). As for the SPARQL query expressivity, currently, our system only supports some basic functions of the SPARQL query language such as literal operations. For complex functions such as filter, order and so on, we have not dealt with them.

Definition 3. (Variable Bindings) A variable binding is a function called *bind* from the set of variables V to the set of URIs U or the set of Literals L. It can be formalized as $bind$: $V \rightarrow$U$\cup$L.

Definition 4. (Join Condition) Given multiple QTPs in a query, a join condition exists between two QTPs qtp_i and qtp_j iff there exists one variable $v \in V$, $v \in qtp\_i$ and $v \in qtp\_j$. Joins are commutative. Variable v is termed the join variable.

In our query processing system, a query may consist of either one QTP or multiple QTPs where each QTP satisfies the join condition with at least one other QTP. By using

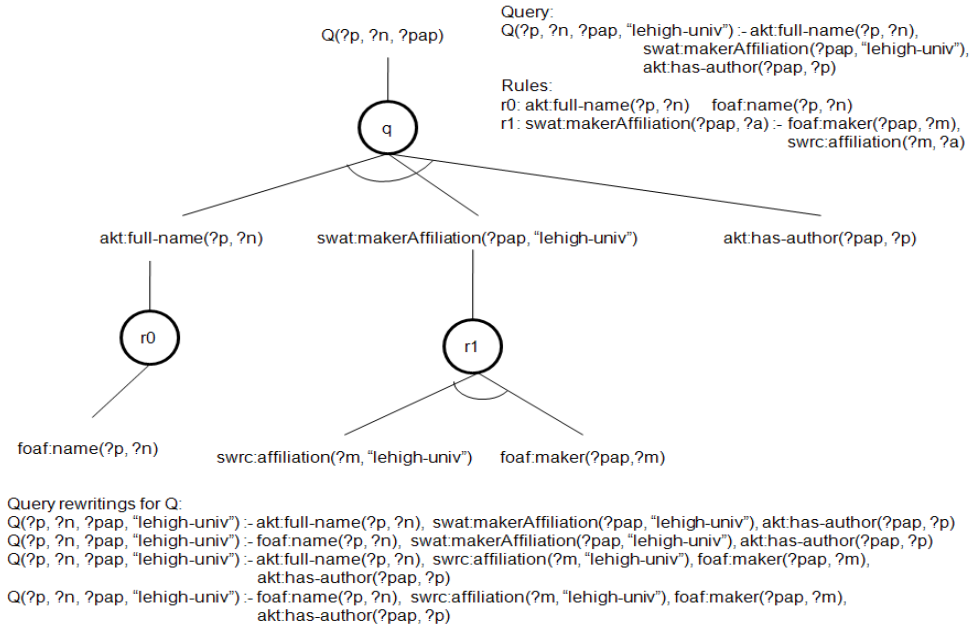

Figure 2: AND-OR tree for a sample conjunctive query

the join conditions, we can determine the selectivity of each QTP having join relation with the current QTP.

Definition 5. (Join Evaluation) Given a join condition between two QTPs qtp_i and qtp_j through a shared variable v, the join is evaluated by performing a resolution on one of the QTPs qtp_i or qtp_j, retrieving a binding for v and performing a computation of number of sources for the remaining QTP with the v binding inserted.

The above definition defines the process of determining the selectivity of other QTPs having join relations with the current processing QTP by using the substitutions for the shared variable provided by the current QTP.

Observation 1.(QTP selectivity) Based on the resource index I and given a conjunctive query Q, the selectivity of one QTP contained in Q can be reflected by the number of sources that can contribute to solving the current QTP by accessing I. Principally, given one QTP, the smaller the number of sources potentially solving this QTP is, the more selective this QTP is.

Given a conjunctive query, our algorithm first computes the selectivity of each QTP contained in this query. Based on the comparison of all QTPs' selectivity, we choose the most selective one as our query execution starting point and evaluate it by asking the reasoner. After this step, we get the set of substitutions for the variables in the chosen QTP. Then, these substitutions are applied into those QTPs satisfying the join condition with the chosen QTP. For each pair of them, the join evaluation process is executed and the selectivity for each QTP in the original query is then updated. With the new selectivity of each QTP, we then select the next most selective one to start the join evaluation and consequently update the selectivity. This process is iteratively executed until all QTPs have been evaluated. Finally, the answers and the sources determined by the given conjunctive query can be returned.

Figure 3 shows us the query optimization tree for one given sample query. In this tree, each tree node consists of three fields: the first one labeled by θ is to store the current available result set, the second one stores the QTP node information and the third one stores the list of selected sources by the current QTP node. From Figure 3, we can see the given query $q(?p, ?n, ?pap)$ consists of four QTPs: $akt{:}full\text{-}name(?p, ?n)$ as qtp_1, $swrc{:}affiliation(?p, {:}lehigh-univ)$ as qtp_2, $foaf{:}maker(?pap, ?p)$ as qtp_3 and $akt{:}has\text{-}title(?pap,$ "$semantic-web$") as qtp_4. By accessing the resource index, we obtain their individual selectivity respectively: 10 million sources for qtp_1, 80 sources for qtp_2, 4 million sources for qtp_3 and 75 sources for qtp_4. Therefore, at the beginning, qtp_4 is the most selective one and chosen to be the starting point. Then, we evaluate qtp_4 and get 5 substitutions for the variable ?pap in qtp_4. By checking the join condition of qtp_1, qtp_2, qtp_3 with qtp_4 respectively, we find qtp_3 having join relation with qtp_4. So, we apply the substitutions of qtp_4 into the join evaluation process for qtp_3 and get its new selectivity as 90. For qtp_1 and qtp_2, since they do not satisfy join condition with qtp_4, they will be directly inherited from the first level (represented by dashed lines). Then, we enter the second level. In this level, we have three QTPs (qtp_1, qtp_2 and qtp_3) with different selectivity. Now, qtp_2 is the most selective one and will be chosen. Then, the conjuncts of qtp_2 and qtp_4 will be evaluated by asking the reasoner. The new substitution set will be applied into the join evaluations for both qtp_1 and qtp_3 because both of them have join relation with qtp_2. After this step, we enter the level 3. In this level, we only have two QTPs (qtp_1 and qtp_3) for comparison and qtp_3 should be selected because of its higher selectivity. Then, the conjuncts of qtp_4, qtp_2 and qtp_3 are evaluated and the new

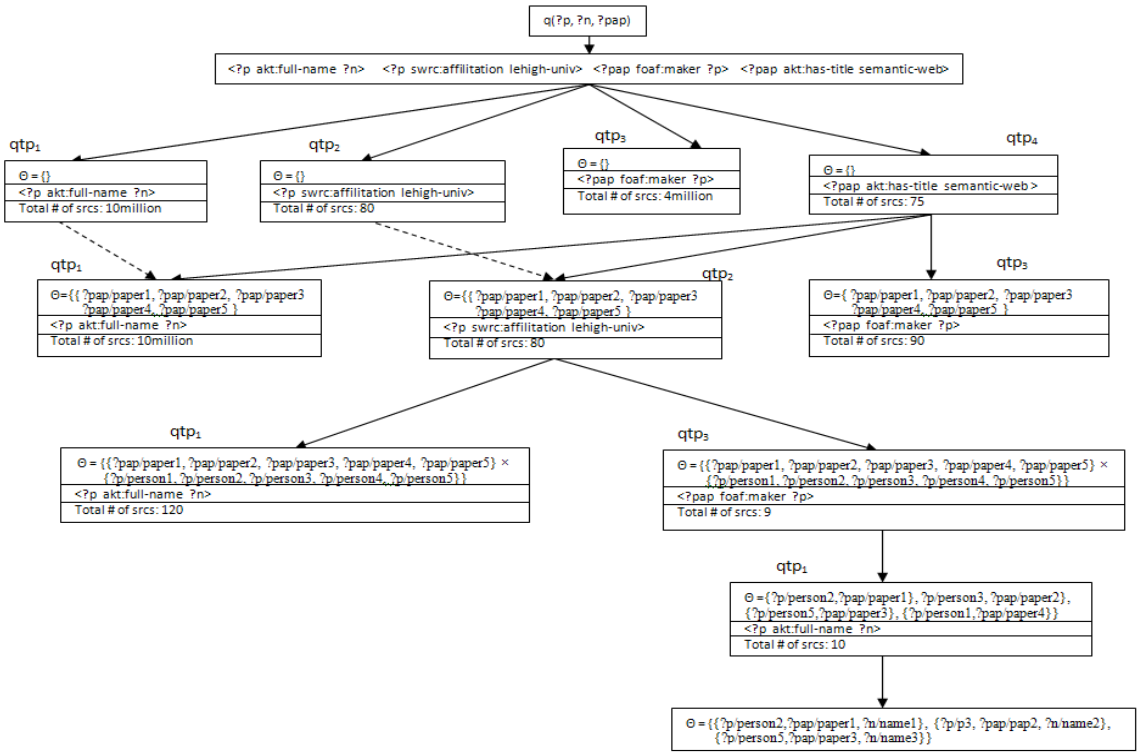

Figure 3: Query optmization tree

substitution set is applied into the join evaluation of qtp_1. After this step, we enter level 4, where there is only one QTP (qtp_1) and it is selected to be evaluated. After qtp_1 evaluation, till now, all four QTPs contained in the original query have been traversed. Hence, the new substitution set contains the final answers to the given query. Finally, the numbers of sources selected by each QTP are 75 for qtp_4, 80 for qtp_2, 9 for qtp_3 and 10 for qtp_1. As for the number of sources determined by the given conjunctive query, it should be equal to or less than (75+80+9+10 = 174). To make the following discussion easier, we denote this number as N. If the given query is one query rewriting for the original query, these selected sources will be collected and finally contribute to solving the original query.

Note, there exist two reasons making $N \leq 174$. The first one is that these 174 sources contain some duplicate sources, which means some sources can contribute to solving two or more than two QTPs in the same query. The second one is that some of these 174 sources probably have been loaded before. In this case, we only need to collect those sources that have not been loaded yet. Because of these two reasons, we can conclude that the exact number of sources determined by one query rewriting should be the number of those sources that have not been loaded before after removing the duplicates.

The algorithm pseudo codes are shown in Algorithm 1, Algorithm 2 and Algorithm 3.

In Algorithm 1, each query rewriting is fed into the source selection algorithm (Algorithm 2) to collect sources. After all rewritings have been checked, the collected sources are used to answer the original query. Because every final collected source has been actually loaded into the reasoner during the source collection (Line 4), in Algorithm 1, we do not need to load them again and can directly ask the reasoner to get the query answers (Line 5).

Algorithm 1 Flat-structure query optimization

function ResultSet queryResolved(Query *q*) **returns** a set of substitutions for *q*

inputs: *q*, a given query *q*

1: Let *srcslist* = {}
2: Let *rewritings* = getRewritings(*q*)
3: **for each** $rewrite \in rewritings$ **do**
4: getSourceList(*rewrite*)
5: ResultSet *rs* = askReasoner(*KB*, *q*, *reasoner*)
6: **return** *rs*

Figure 4: flat-structure query optimization algorithm

In Algorithm 2, the *topGoals* (line 5) contains those QTPs that have not been solved and the *completeGoals* (line 14) contains those QTPs that have been solved. During each iteration, one intermediate SPARQL query consisting of the QTPs contained in the *completeGoals* is generated to ask the reasoner for the substitutions (line 15 to line 17). When the *topGoals* becomes empty, our algorithm terminates because all QTPs have been evaluated and the collected sources are returned. In the while loop (line 8 to line 18), when every most selective QTP has been chosen (line 9), we will collect the sources determined by the current QTP and then load them into the reasoner (line 11 and line 12). So, this is the point to explain why in Algorithm 1 we do not need to load the collected sources again.

The process of join evaluation is shown in Algorithm 3.

Algorithm 2 Source selection for one given rewriting query (Part I)

```
function List getSourceList(Query q) returns a list of sources determined by q
    inputs: q, a given query q
1:  Let ResultSet rs = {{}}, List topGoals = {}, List completeGoals = {}, Tree tree = new Tree(),
    List srclist = {},
2:  for each qtp =< sub, pre, obj >∈ q do
3:      Let srcslist = askIndexer(INDEX, qtp)
4:      Let newNode = makeTreeNode(rs, qtp, srcslist)
5:      add(topGoals, newNode)
6:      addChild(root, newNode)
7:  addRoot(tree, root)
8:  while (topGoals ≠ ∅) do
9:      Let n = min_{n ∈ topGoals ∧ n ∈ tree}(n.numOfSrcs)
10:     add(srclist, n.srcslist)
11:     for each s ∈ n.srcslist do
12:         load(s, reasoner)
13:     Remove n from topGoals
14:     add(completeGoals, n)
15:     Let sub_query = ∧_{qtp_i ∈ completeGoals}(qtp_i)
16:     /*Get the set of substitutions for the sub_query by asking the Reasoner*/
17:     Let rs = askReasoner(KB, sub_query, reasoner)
18:     expandTree(n, rs, topGoals)
19: return srclist
```

Figure 5: Source selection for one given rewriting query (Part I)

Algorithm 3 Source selection for one given rewriting query (Part II)

```
function expandTree(Node n, ResultSet rs, List topGoals)
    input: n, a qtp tree node with the minimum number of sources
           rs, a list of substitutions
           topGoals, a set of qtps that have not been solved
1:  /*List join_candidlist is a list of qtps having join relation in subject or object with the n.qtp
    that is the qtp corresponding to the node n */
2:  Let join_candidlist = {qtp_i | (qtp_i ∈ topGoals) ∧ (qtp_i =< sub_i, pre_i, obj_i >) ∧
    (n.qtp =< sub_n, pre_n, obj_n >) ∧ (var(qtp_i)∧var(n.qtp) ≠ ∅)}
3:  Let non_join_candidlist = topGoals − join_candidlist
4:  Let srcslist = {}
5:  for each qtp =< sub, pre, obj >∈ join_candidlist do
6:      for each θ ∈ rs do
7:          Let list = askIndexer(INDEX, qtp, θ)
8:          add(srcslist, list)
9:      Let newNode = makeTreeNode(rs, qtp, srcslist)
10:     addChild(n, newNode)
11: for each qtp ∈ non_join_candidlist do
12:         if qtp ∉ n.getChildren() then
13:             Let srcslist = getSrcsList (qtp)
14:             Let newNode = makeTreeNode (rs, qtp, srcslist)
15:             addChild(n, newNode)
16: return
```

Figure 6: Source selection for one given rewriting query (Part II)

The *join_candidlist* (line 2) contains those QTPs that satisfy the join condition with the current checking QTP, while the *non_join_candidlist* (line 11) contains those QTPs that have no join relations with the current checking QTP. For each QTP in the *join_candidlist*, the join evaluation is executed to expand our optimization tree (line 5 to line 10). By this process, the selectivity of each QTP can be updated and used for the next iteration.

4. EVALUATION

To evaluate our query optimization algorithm, we have conducted two experiments based on the synthetic data set and the real world data set respectively. The first experiment is to compare with the non-structure query answering algorithm in the work of [9]. The second experiment is based on a subset of the real world BTC data set. Because the non-structure query optimization algorithm cannot very well scale into the real world data set because of the disk/network latency or the big number of sources for the predicates, in this experiment, we mainly give the performances of the flat-structure algorithm running in real world data set. These two groups of experiments are done on a workstation with Xeon 2.93G CPU and 6G memory running UNIX. In the indexer implementation, we currently use Lucene as our index builder. In all cases, we use KAON2 as our Reasoner.

Before go further into our evaluation, we need to first talk about the query generator for our experiments. This is because our flat-structure algorithm is selectivity directed, which can be exactly applied into those real queries with at least one constant in their QTPs.

4.1 Query Generator

In order to evaluate both flat-structure algorithm and non-structure algorithm, we need to construct some test queries that match the real world query features. Based on our observations, we think the real world queries have the following three characteristics.

- Each query have at least one answer. This requirement guarantees all QTPs of each query can be checked.
- The number of QTPs for each query varies. Then, we can show that the flat-structure algorithm can scale well with the number of unconstrained QTPs increasing. This is because more unconstrained QTPs means more sources should be selected to answer the query. Here, for those QTPs without constants, we call them unconstrained QTPs. This kind of QTPs is generally not selective.
- Each QTP of each query should satisfy the join condition with at least one other QTP in the same query. This way guarantees the selectivity of all QTPs in one query can be updated and query execution plan can be created.

With these three requirements, we developed a graph-based query generator. As we know, the RDF data format is by its very nature a graph. Therefore, each SPARQL query is basically one subgraph over the graph of the whole Knowledge Base (KB). Then, these different subgraphs form different query patterns. After query patterns determined, we need to replace some selected node values with query variables. In this process, if the junction node of the query pattern is replaced with one query variable, then this variable would be counted as the join variable in our final generated queries. To illustrate this process, we can give an example. Suppose the KB has the following knowledge:

<swat:jeff swat:first_name “jeff”>

<?x swat:first_name “jeff”>
<?x swat:last_name ?y>

In developing the query generator algorithm, the core function is how to expand graph nodes to generate the graph of the complete or part of the whole KB and also to construct the query subgraph pattern. In this process, for the expanded node, two main steps are executed.

• If there is one triple connecting the current expanded node to another new node, one edge with the predicate being the edge weight and starting from the checked node to the new node will be added. At the same time, the new node is also added.

• On the other hand, if there is one triple that connects the expanded node into another already expanded node, we just add the new edge.

During our experiment of query generation, there is one point about KAON2 reasoner we came across being worthwhile to be mentioned here. By using our query generator, we could generate queries having one QTP in form of $<constantURI, rdf:type, ?x>$. For this kind of QTP, the KAON2 reasoner does not support it. However, it seems reasonable that such QTP appears in the SPARQL queries, because sometimes we need to know the class type of one specific instance.

4.2 Source Selectivity Evaluation

Our first experiment attempts to demonstrate that the flat-structure algorithm is superior to the non-structure algorithm in source selectivity without much cost of index accessing. In our experiments, there are two main factors that affect the performance of both algorithms. One is the index accessing times. The more the index accessing times is, the more time-consuming the algorithm is. The other is the source loading time. After the potentially relevant sources selected, we will load them into the KAON2 reasoner to solve the query. These two factors can be reflected by the index accessing times and the number of selected sources. Therefore, in our experimental results, we will give the comparison of both algorithms on the metrics of average query response time, average number of selected sources and average index accesses.

In addition, in order to make the synthetic data set much closer to real world data, we modified the original data generator. First, we ensure that each generated file is a connected graph, which more accurately reflects most real-world RDF files. Second, we randomly sampled 200 semantic web documents and determined that the average number of triples in each result document is around 54.015 with the standard deviation 163.9148; in the interest of round numbers, we set the average number of triples in a generated document to be 50. In addition, we also noticed that in real world, there is only a small number of ontologies being committed to by a large number of semantic we data sources. Therefore, we conducted experiments with 20 ontologies, 8000 data source sources, and a diameter of 6, meaning that the longest sequence of mapping ontologies between any two domain ontologies was six. In this configuration, the average number of sources committing to each ontology is around four hundred.

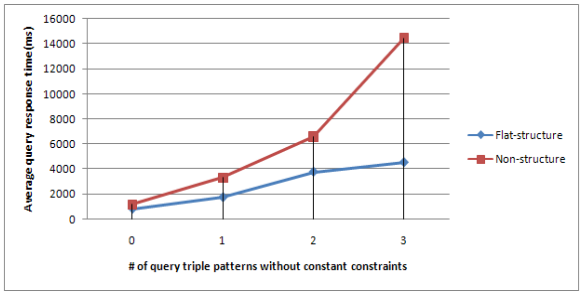

Figure 7: Average query response time with increasing number of unconstrained qtps

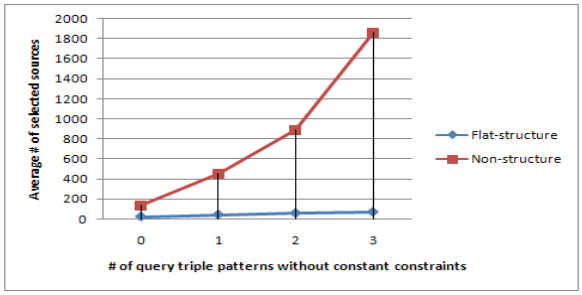

Figure 8: Average number of selected sources with increasing number of unconstrained qtps

In this experiment, our index size is 75.3MB. The time to construct the index is 21568ms. We issue 100 random queries and compute the average number of query response time, selected sources and index accessing times. All these metrics are calculated with the number of unconstrained QTPs increasing in the generated query set. The unconstrained QTPs are those QTPs without constants assigned to its either subject or object. The experimental results are shown in Figure 7, Figure 8 and Figure 9.

Figure 7 displays the comparison of the average query response time with the number of unconstraint QTPs increasing for both algorithms. From this result, we can see the flat-structure algorithm performs better than the non-structure algorithm in all cases. Furthermore, we can also conclude that the flat-structure algorithm scales better with the number of unconstrained QTPs increasing than the non-structure algorithm does. This is because with the number of unconstrained QTPs increasing, there would be more number of selected sources used to solve the query. At the same time, because our flat-structure algorithm has a better selectivity, which is shown in Figure 8, the query response time will not be increased sharply as the non-structure algorithm does.

Figure 8 displays the comparison of the average number of selected sources with the number of unconstrained QTPs increasing for both algorithms. From this result, we can see the selectivity of the flat-structure algorithm is roughly linear, while the non-structure algorithm is exponential with the number of unconstrained QTPs increasing. Therefore, we can say the flat-structure algorithm has gained a better source selectivity than the non-structure algorithm does.

Figure 9 displays the comparison of the average number of index accessing times with the number of unconstrained

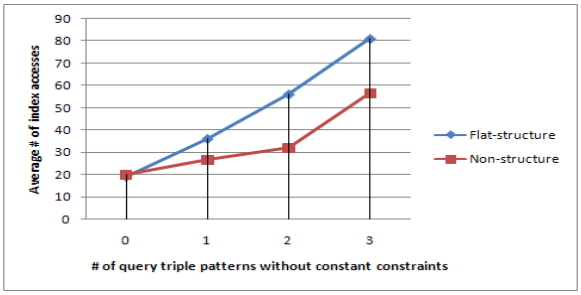

Figure 9: Average index accesses with increasing number of unconstrained qtps

Data sources	Ontology name-sapce	# of Triples
http://data.semanticweb.org/	swrc	174,816
http://sws.geonames.org/	geonames	14,866,924
http://dbpedia.org	dbpedia	48,694,372
http://dblp.rkbexplorer.com	akt	10,153,039
Total		73,889,151

Table 1: Data sources

QTPs increasing for both algorithms. From this result, we can see the flat-structure algorithm has more index accesses than the non-structure algorithm when the number of unconstrained QTPs increases. This is because our proposed flat-structure algorithm takes into account the structure information while solving each rewriting subquery for the original query. So, it has more index accesses.

Based on the experimental results shown in Figure 7, Figure 8 and Figure 9, we can conclude that the flat-structure algorithm gains better source selectivity by increasing its index accesses than the non-structure algorithm does. Furthermore, as we said before, the index accesses and source selectivity are two main factors to affect the query response time and for the flat-structure algorithm, the benefits of source selectivity gained has offset the cost of increasing the index accesses, therefore, we can conclude that the flat-structure query optimization algorithm performs better than the non-structure query answering algorithm even though it has higher cost of index accesses.

4.3 Scalability Evaluation

In this section, we will evaluate our system's scalability by using set of real world data sources. We choose a subset of the Billion Triple Challenge (BTC) 2009 data set, focusing on four collections as summarized in Table 1. The total number of triples in this dataset is 73,889,151. We created local N3 versions of the original files from the BTC by using the provenance information, result in 21,008,285 documents. The size of documents varies from roughly 5 to 50 triples each.

In the real world data set, because the non-structure algorithm does not use the constant constraints to plan the query execution, it cannot scale into the real world data set. In our generated test query set, most of them cannot be solved by the non-structure algorithm. Take the SPARQL query in Figure 10 for example.

For this query, the number of sources that can potentially

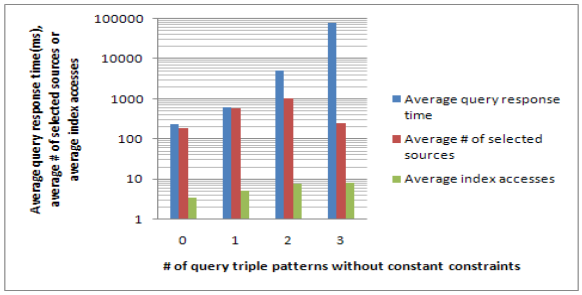

Figure 11: Experimental results of the flat-structure algorithm over real world data set

contribute to solving the QTP $<?x_2, foaf:maker, ?x_0>$ is 3,485,607, which is so big a number that the non-structure algorithm cannot load them into the reasoner to solve it. Therefore, for this kind of queries, the non-structure algorithm cannot deal with them. However, the flat-structure algorithm can deal with it very well because the number of sources for the same QTP become 114 after constant considered. Therefore, in this part, we only give the experimental results of the flat-structure algorithm on the real world data set.

As shown in Figure 11, the flat-structure algorithm can scale well into the real world data set. According to the experimental results, for the whole query set, the average query response time is 35.490s, the average number of index accessing times is around 4.822916 and the average number of selected sources is around 511.3020.

In this experiment, our index construction time is aound 58 hours and its size is around 18GB. Each document takes around 10ms on average. The Lucence configurations are 1500MB for RAMBufferSize and 1000 for MergeFactor, which are the best tradeoff between index building and searching for our scalability.

Since our algorithm does not yet select all relevant sources with sameAs information, we assume an environment where any relevant sameAs information is already supplied to the reasoner. We do this by initializing the KB with the necessary sameAs statements.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a flat-structure query optimization algorithm for the query answering of the ontology-based information integration. According to this method, during the query evaluation, we first evaluate the selectivity of each QTP for each of its rewriting subqueries by accessing the index, and choose the most selective one with the minimum number of sources to solve. Then, we can apply the intermediate results into other QTPs to determine the new number of sources that contribute to solving each of them. With these results, the new most selective QTP will be chosen and solved. This process is iteratively executed until the whole query is answered. In other words, given a set of query rewritings that accounts for ontology heterogeneity, our proposed algorithm incrementally selects and processes sources in order to maintain selectivity. Once sources are selected, we use an OWL reasoner to answer queries over these sources and their corresponding ontologies. We have demonstrated that our new algorithm is better than the work in [9]

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x_0 ?x_1 ?x_2 WHERE {
?x_0 <http://swrc.ontoware.org/ontology#affiliation> <http://data.semanticweb.org/organization/lehigh-university> .
?x_2 <http://www.aktors.org/ontology/portal#has-title> "Hawkeye: A Practical Large Scale Demonstration of Semantic Web Integration" .
?x_2 <http://xmlns.com/foaf/0.1/maker> ?x_0 .
?x_0 <http://www.aktors.org/ontology/portal#full-name> ?x_1 .
}
```

Figure 10: A real world SPARQL sample query

in that not only does it have better query response time, but it also can gain higher source selectivity with the number of unconstraint QTPs increasing even though the index accessing times is increased. We have also shown that the system scales well, allowing randomly generated queries against 20 million heterogeneous data sources to complete in seconds.

However, there is still significant room for improvement. First, it is relatively more expensive to use our current algorithm to compute the set of rewritings for the given conjunctive query. We intend to develop a better query optimization algorithm that cannot only gain quicker query response speed and better source selectivity but we can also save the cost of calculating the query rewritings. Second, real semantic web sources often use the different identifiers for the same entity. Our algorithm needs to be adapted to locate relevant owl:sameAs statements; this must necessarily be an iterative process in order to find the transitive closure of relevant owl:sameAs statements. We believe that solving such problems will lead to a pragmatic solution for querying a large, distributed, and ever changing Semantic Web.

6. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] P. Adjiman, F. Goasdoué, and M. c. Rousset. Somerdfs in the semantic web.

[3] O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In S. Auer, C. Bizer, C. Müller, and A. V. Zhdanova, editors, *Conference on Social Semantic Web*, volume 113 of *LNI*, pages 59–68. GI, 2007.

[4] P. Haase and Y. Wang. A decentralized infrastructure for query answering over distributed ontologies. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1351–1356, New York, NY, USA, 2007. ACM.

[5] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. pages 556–567, 2003.

[6] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. *Data Engineering, International Conference on*, page 505, 2003.

[7] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. pages 211–224, 2008.

[8] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *In VLDB*, pages 128–137, 1986.

[9] Y. Li, A. Qasem, and J. Heflin. A scalable indexing mechanism for ontology-based information integration. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, 2010.

[10] 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. 2001.

[11] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 627–640, New York, NY, USA, 2009. ACM.

[12] A. Owens. Clustered tdb: A clustered triple store for jena.

[13] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, I. A. Lorie, and T. G. Price. Access path selection in a relational database management system. pages 23–34, 1979.

[14] L. Serafini and A. Tamilin. Drago: Distributed reasoning architecture for the semantic web. In *In ESWC*, pages 361–376. Springer, 2005.

[15] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, 2008.

[16] H. Stuckenschmidt, R. Vdovjak, J. Broekstra, and G. Houben. Towards distributed processing of rdf path queries. *Int. J. Web Eng. Technol.*, 2(2/3):207–230, 2005.

[17] T. Tran, H. Wang, and P. Haase. Hermes: Data web search on a pay-as-you-go integration infrastructure. *Web Semant.*, 7(3):189–203, 2009.

[18] O. Udrea, A. Pugliese, and V. S. Subrahmanian. Grin: A graph based rdf index. In *AAAI*, pages 1465–1470, 2007.

[19] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, pages 1008–1019, 2008.

[20] L. Zhang, Q. Liu, J. Zhang, H. Wang, Y. Pan, and Y. Yu. Semplore: An ir approach to scalable hybrid query of semantic web data.