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 rewritings and a term 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 term 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, Ontology

1. INTRODUCTION

In the Semantic Web, the definitions of resources and the relationship between resources are described by ontologies. The resources in the Web are independently generated and distributed in many locations. In such an environment, we often need to integrate the ontologies and their data sources and access them without regard to the heterogeneity and the dispersion of the ontologies. In order to support this requirement, we proposed an index-based mechanism for ontology-based information integration [2]. According to this method, each RDF and OWL data source in the Semantic Web can be treated as a bag of URIs and Literals. Then, a term index is created to integrate these sources. If we reformulate a conjunctive query into a set of Boolean subgoals, then we can use this index to only access those sources that might be relevant to the query. However, because the term 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. In addition, in the real world, the number of sources related to a subgoal could be so large that it is impossible to load all of them into a reasoner that can then answer the queries. To address these issues, we present a query optimization algorithm for ontology-based information integration using a term index. Given a set of query rewritings that accounts for ontology heterogeneity, this algorithm incrementally selects and processes sources. 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:

- We present a "flat-structure" query optimization algorithm that takes a set of query rewritings as input and uses the selectivity of each triple pattern in the rewritings as the heuristic to plan query execution.
- We conduct a number of experiments to evaluate the characteristics of our proposed algorithm. We demonstrate that the proposed algorithm outperforms our previous algorithm [2] on both a synthetic data set with 20 ontologies having significant heterogeneity and a real world data set with 73,889,151 triples distributed in 21,008,285 documents.

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 term 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 RDF query optimization, Hexastore [8] creates 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 are that it relies on centralized knowledge bases and that the indexes are quite expensive in terms of space. GRIN [7] developed a novel index for graph-matching queries in RDF. This index identifies selected central vertices and the distance of other nodes from these vertices. However, it is still not clear how GRIN could be adapted for a distributed context. In query answering over distributed ontologies, T. Tran et al. proposed Hermes [6], which translates a keyword query into a federated query and then decomposes this into separate SPARQL queries. A number of indexes are used, including a keyword index, mapping index, and structure index. The main drawback is that it does not account for schema heterogeneity. Stuckenschmidt et al. [5] suggested a global data summary for locating data matching query answers in different sources and the query optimization. However, this method does not consider the heterogeneity of the distributed ontologies. In the database field, Selinger et al. [4] proposed query optimization ideas including using statistics about the database instance to estimate the cost of a query evaluation plan, considering only plans with binary joins in which the inner relation is a base relation (left-deep plans) and postponing Cartesian product after joins with predicate. However, in the Semantic Web, it is very common that data from the same relation is spread among many files. In such situations, query plans need to be developed incrementally.

3. QUERY OPTIMIZATION ALGORITHM

As stated in the introduction, our flat-structure query optimization algorithm is based on a term index that is used to integrate the distributed and heterogeneous semantic web ontologies and data sources. Basically, the term index is an inverted index, where each term is either a full URI (taken from the subject, predicate or object of a triple) or a string literal value. Due to limited space, we do not present details of the term index. Please see our technical report [1] for details. Given a set of conjunctive query rewritings, our algorithm employs a source selection strategy that prioritizes selective subgoals of the query and uses the sources that are relevant to these subgoals to provide constraints that could make other subgoals more selective. This optimization algorithm can be combined with any query rewriting algorithm that produces a set of conjunctive queries. We begin by providing a brief overview of the architecture of our system and then discuss the details of our proposed query optimization algorithm.

In our system, the Indexer is periodically run to create the term index for all of the data sources, which are committing to different heterogeneous domain ontologies. These ontologies are integrated by mapping ontologies and the predicates defined in them are spread among many data sources. All ontologies are expressed in OWLII, a fragment of OWL [3]. The Indexer translates the axioms in these ontologies into

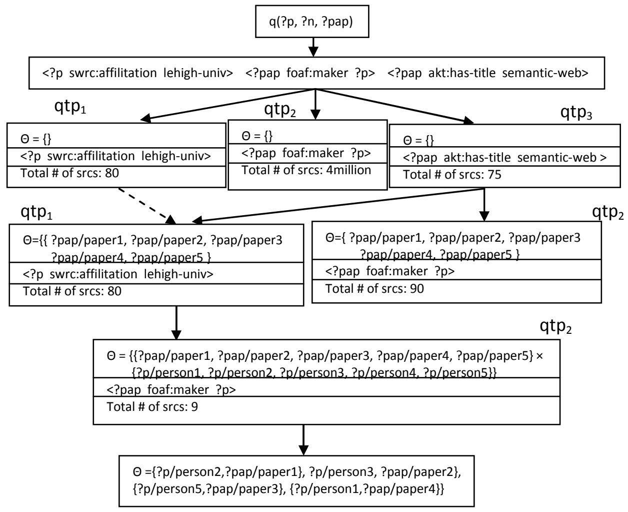
q(?p, ?n, ?pap)
<?p swrc:affilitation lehigh-univ> <?pap foaf:maker ?p> <?pap akt:has-title semantic-web>
qtp1
qtp2
qtp3
Θ = {}
<?p swrc:affilitation lehigh-univ>
Total # of srcs: 80
Θ = {}
<?pap foaf:maker ?p>
Total # of srcs: 4million
Θ = {}
<?pap akt:has-title semantic-web >
Total # of srcs: 75
qtp1
qtp2
Θ={{ ?pap/paper1, ?pap/paper2, ?pap/paper3 ?pap/paper4, ?pap/paper5 }
<?p swrc:affilitation lehigh-univ>
Total # of srcs: 80
Θ={ ?pap/paper1, ?pap/paper2, ?pap/paper3 ?pap/paper4, ?pap/paper5 }
<?pap foaf:maker ?p>
Total # of srcs: 90
qtp2
Θ = {{?pap/paper1, ?pap/paper2, ?pap/paper3, ?pap/paper4, ?pap/paper5} × {?p/person1, ?p/person2, ?p/person3, ?p/person4, ?p/person5}}
<?pap foaf:maker ?p>
Total # of srcs: 9
Θ ={?p/person2,?pap/paper1}, ?p/person3, ?pap/paper2}, {?p/person5,?pap/paper3}, {?p/person1,?pap/paper4}}

Figure 1: Query optimization tree

GAV/LAV rules. Given a conjunctive query, the Reformulator uses the domain and mapping ontologies to determine all possible conjunctive query rewritings. In our current implementation, we use the algorithm proposed by Qasem et al. [3] to generate query rewritings. For each rewriting, we employ our proposed optimization algorithm to incrementally collect relevant data sources. Once sources are selected, the Loader reads the selected sources together with their corresponding ontologies and inputs them into a sound and complete OWL reasoner, which is then queried to produce results. Since the selected sources are loaded in their entirety into a reasoner, any inferences due to a combination of these selected sources will also be computed by the reasoner.

As mentioned earlier, the flat-structure algorithm prioritizes selective query triple patterns (QTPs) to incrementally select relevant sources and solve queries. Thus, given a conjunctive query, our algorithm first computes the selectivity of each query triple pattern (QTP) contained in this query. We have found that typically the selectivity of a QTP is closely related to the number of sources that the term index determines are relevant to it. Therefore, in the following parts, when we mention a QTP's selectivity, we mean with respect to the number of relevant sources. Through the comparison of all QTPs' selectivities, we start with the most selective one and evaluate it by asking the reasoner. Then, the substitutions obtained from last step are applied to those QTPs having a shared variable (join condition in database terminology) with the chosen QTP and their respective selectivities are updated correspondingly. Then, we will start with the next most selective QTP and repeat the previous steps. This process is iteratively executed until all QTPs have been evaluated. Finally, the query answers and its relevant sources can be identified.

Figure 1 shows us an example. In this tree, each node consists of three fields: the available substitutions, the QTP node and the selected sources. This sample query includes three QTPs: $\langle$ $?p$, $swrc{:}affiliation$, $lehigh\text{-}univ$ $\rangle$ (qtp_1), $\langle$ $?pap$, $foaf{:}maker$, $?p$ $\rangle$ (qtp_2) and $\langle$ $?pap$, $akt{:}has\text{-}title$, "$semantic\text{-}web$" $\rangle$ (qtp_3). Using the term index, we might find that these QTPs' selectivities are 80, 4 million and 75 respectively. Since qtp_3 is the most selective, we load and

```
Algorithm 1 Query optimization
function OptimizeQuery(Query q) returns a list of sources
    inputs: q, a conjunctive query
1:  Let allsrcs = ∅, query = true, sibs = a set of qtps in q, rs = ∅
2:  srcs[] = array of sets of sources, indexed by qtps
3:  while ($sibs \neq \emptyset$)
4:      for each $qtp \in sibs$ do
5:          if ($rs = \emptyset$) then
6:              srcs[qtp] = index-lookup({qtp})
7:          else srcs[qtp] = $\cup_{\theta \in rs}$ index-lookup({$qtp\theta$})
8:      Let on = $min_{node \in sibs}$ ($|srcs[node]|$)
9:      $allsrcs = allsrcs \cup srcs[on]$
10:     load(srcs[on], KB)
11:     $sibs = sibs - \{on\}$
12:     Let query = query $\wedge$ on
13:     Let rs = askReasoner (KB, query)
14: return allsrcs
```

Figure 2: Flat-structure query optimization algorithm

evaluate its sources first. Then, we apply the obtained substitutions for ?*pap* into qtp_1 and qtp_2. After this step, their selectivities are updated to be 80 and 90 respectively. Then, we start to evaluate the next most selective QTP, qtp_1, and apply its substitutions for ?*p* into qtp_2. After this step, we only have qtp_2 left and evaluate it. Finally, the numbers of sources selected by each QTP are 75 for qtp_3, 80 for qtp_1 and 9 for qtp_2. Therefore, the total number of sources identified by the given query is 75+80+9 = 164. Note, in this process, we keep track of all sources that have been loaded, and do not repeat the loading of any source while answering a particular query.

The details of our approach are given in Algorithm 1. It takes a conjunctive query rewriting as its input. First, we initialize the selectivity of each QTP contained in *sibs* by executing a term index lookup (Lines 5-6). Then, we assign the most selective QTP to *on* and collect its relevant sources (Lines 8-10). Meanwhile, we remove *on* from *sibs* (Line 11) and evaluate *on* to get its substitutions (Lines 12-13). Each substitution θ is then applied to *on*'s siblings to constrain their individual selectivity (Line 7). Based on the new selectivity, the next most selective node is chosen and the above process is repeated until all QTPs have been processed (Line 3). Finally, the sources collected by *q* are returned (Line 14).

4. EVALUATION

In this section, we have conducted experiments on both synthetic and real world data sets to evaluate our query optimization algorithm. The evaluations are done on a workstation with Xeon 2.93G CPU and 6G memory running UNIX. For both experiments, we use the queries automatically generated by a graph-based synthetic query generator. These queries range from one to four triples, have at most four variables each, and each QTP of each query satisfies the join condition with at least one sibling QTP. For more details, see our technical report [1]. Our implementation of the Indexer uses Lucene to build the inverted index. In all cases, we use KAON2 as our Reasoner.

4.1 Source Selectivity Evaluation

Our first experiment compares the flat-structure algorithm to our original "non-structure" algorithm [2] with respect to

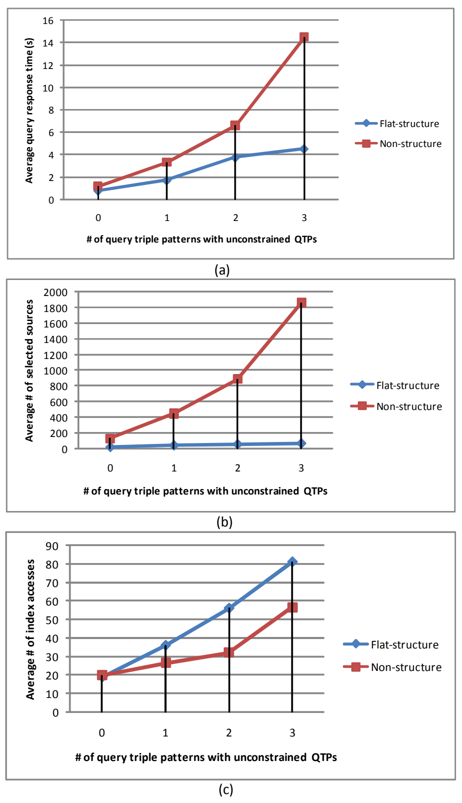

Figure 3: Average query response time with increasing number of unconstrained QTPs

query response time, source selectivity and number of index accesses. We conducted this experiment with 20 ontologies, 8000 data sources, and a diameter of 6, meaning that the longest sequence of mapping ontologies between any two domain ontologies is six. It took 21.5 seconds to build the 75.3MB index. We issue 100 random queries and for each query compute the response time, number of selected sources and number of index accesses. We then group queries based on the number of *unconstrained QTPs*, and compute average for each group. We define an unconstrained QTP as one with variables for both its subject or object, or with the *rdf:type* predicate and a constant object.

Figure 3(a) displays the comparison of the average query response time for both algorithms. We can see that the flat-structure algorithm performs better than the non-structure algorithm in all cases. Furthermore, we can conclude that the flat-structure algorithm scales better with an increasing number of unconstrained QTPs than the non-structure algorithm does. This is because more unconstrained QTPs lead to more opportunities to optimize the query by intelligently selecting sources. At the same time, because our flat-structure algorithm has a better selectivity shown in Figure 3(b), the query response time does not increase sharply as the non-structure algorithm does. Figure 3(b) displays the comparison of the average number of selected sources for both algorithms. We can see the selectivity of the flat-structure algorithm is roughly linear, while the non-structure algorithm is exponential the number of unconstrained QTPs. Figure 3(c) displays the comparison of the average number of index accesses for both algorithms.

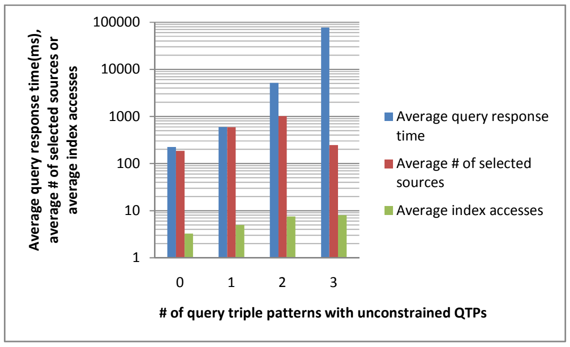

Figure 4: Experimental results of the flat-structure algorithm over the real world data set

It shows that the flat-structure algorithm has more index accesses than the non-structure algorithm when the number of unconstrained QTPs increases. The reason is that in case of the flat-structure algorithm, one QTP can appear in multiple query rewritings and consequently has a greater number of index lookups.

4.2 Scalability Evaluation

In this section, we evaluate our system's scalability by using a set of real world data sources. We chose a subset of the Billion Triple Challenge (BTC) 2009 data set, focusing on four collections: *http://data.semanticweb.org/*, *http://sws.geonames.org/*, *http://dbpedia.org* and *http://dblp.rkbexplorer.com*. The total number of triples in this dataset is 73,889,151, which are scattered in 21,008,285 documents. The size of documents varies from roughly 5 to 50 triples each. In order to enable integration of these heterogeneous documents, we manually created a set of mapping ontologies. Our index construction time is approximately 58 hours and the size of the resulting index is approximately 18GB. Each document takes around 10ms on average to be indexed. The Lucence configurations are 1500MB for RAMBufferSize and 1000 for MergeFactor, which are the best tradeoff between index building and searching for our experiment.

Because the non-structure algorithm does not propagate constant constraints when answering queries, it cannot scale to the BTC data set since most of our synthetic queries have at least one unconstrained QTP. For example, consider the query Q: $\{\langle\ ?x_0$, *swrc*:*affiliation*, "*lehigh-univ*" $\rangle$. $\langle\ ?x_2$, *akt*:*has-title*, "*Hawkeye*" $\rangle$. $\langle\ ?x_2$, *foaf*:*maker*, $?x_0\ \rangle$. $\langle\ ?x_0$, *akt*:*full-name*, $?x_1\ \rangle\}$. For the non-structure algorithm, the number of sources that can potentially contribute to solving $\langle ?x_2$, *foaf*:*maker*, $?x_0\rangle$ is 3,485,607, which is far too many to load into a memory-based reasoner. However, the flat-structure algorithm can easily handle this query because the number of sources for the same QTP becomes 114 after join constants are considered. Thus, we only give the experimental results of the flat-structure algorithm on BTC data set.

As shown in Figure 4, the flat-structure algorithm scales well in source selectivity and index accesses even though the query response time is exponential. According to the experimental results, for the whole query set, the average query response time is 35.5s, the average number of index accesses is around 4.8 and the average number of selected sources is around 511.3.

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

5. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a flat-structure query optimization algorithm for information integration of many sources committing to different ontologies. The experiments demonstrated that our new algorithm is better than our prior work [2] in that it has better query response time, because although it requires more index accesses, its source selectivity is less affected by the number of unconstrained QTPs. We have also shown the system scales to reasonable problem sizes, allowing randomly generated queries against 20 million heterogeneous data sources to complete in 30 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 will further improve query response time and better source selectivity, while also reducing the cost of calculating query rewritings. Second, our algorithm needs to be adapted to locate 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] Y. Li and J. Heflin. Query optimization for ontology-based information integration. Technical Report LU-CSE-10-002, Lehigh University, 2010.

[2] 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.

[3] A. Qasem, D. A. Dimitrov, and J. Heflin. Efficient selection and integration of data sources for answering semantic web queries. *International Conference on Semantic Computing*, pages 245–252, 2008.

[4] 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.

[5] 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.

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

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

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