Using Reformulation Trees to Optimize Queries over Distributed Heterogeneous Sources

Yingjie Li and Jeff Heflin

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

Abstract. In order to effectively and quickly answer queries in environments with distributed RDF/OWL, we present a query optimization algorithm to identify the potentially relevant Semantic Web data sources using structural query features and a term index. This algorithm is based on the observation that the join selectivity of a pair of query triple patterns is often higher than the overall selectivity of these two patterns treated independently. Given a rule goal tree that expresses the reformulation of a conjunctive query, our algorithm uses a bottom-up approach to estimate the selectivity of each node. It then prioritizes loading of selective nodes and uses the information from these sources to further constrain other nodes. Finally, we use an OWL reasoner to answer queries over the selected sources and their corresponding ontologies. We have evaluated our system using both a synthetic data set and a subset of the real-world Billion Triple Challenge data.

Keywords: information integration, query optimization, query reformulation, source selectivity.

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. Under such an environment, there is often the need to integrate the ontologies and their data sources and access them without regard to the heterogeneity and the dispersion of the ontologies. Although recent research has led to the development of knowledge bases (KBs) and/or triple stores to support this need, such systems have many disadvantages. First, centralized knowledge bases will become stale unless they are frequently reloaded with fresh data; this can be especially expensive if the knowledge-bases rely on forward-chaining. Second, they can require significant disk space, especially for triple stores that use multiple triple indices to optimize queries. For example, Hexastore [16] replicates each triple six times. Finally, there may be legal or policy issues that prevent one from copying data or storing it in a centralized place. For this reason, we believe it is important to investigate algorithms that allow data to reside in its original location, and that use summary indexes to

P.F. Patel-Schneider et al. (Eds.): ISWC 2010, Part I, LNCS 6496, pp. 502–517, 2010.

determine which locations contain data relevant to a particular query. In particular, we have proposed an inverted index-based mechanism that indicates which documents mention certain URIs and/or literal strings [5]. This simple mechanism is clearly space efficient, but also surprisingly selective for many queries. However, because this index only indicates if URIs or literal strings 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. To further complicate matters, ontology heterogeneity can lead to query answers being expressed in ontologies different from those used to express the query. To solve these issues, we further proposed a flat-structure query optimization algorithm that selects and processes sources given a set of conjunctive query rewritings [4]. For each rewriting, this 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. However, there are two key problems with this algorithm: 1) the number of rewrites can be exponential in the size of the query, especially when there are complex ontology axioms and 2) the selectivity of the algorithm is inhibited by its reliance on local information.

To solve the above issues, we present a novel algorithm for optimizing the selection of sources in ontology-based information integration systems. Like our prior work, this approach relies on an inverted index; however, the main contributions of this paper are:

- We present a tree-structure algorithm that performs optimizations that consider the structure of a reformulation tree. Using this tree and the term index, it estimates the most selective subgoals, incrementally loads the relevant sources, and uses the data from the sources to further constrain related subgoals.
- We demonstrate that this new algorithm outperforms the algorithms proposed in [4] and [5] 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: Section 2 reviews related work. In Section 3, we describe the tree-structure source selection algorithm for ontology-based information integration. Section 4 describes our experiments and in Section 5, we conclude and discuss future work.

2 Related Work

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

Query optimization has been extensively studied by traditional database researchers since the classic work by Selinger et al. [9]. Variations of these ideas are still common practice in relational optimizers: use statistics about the database

instance to estimate the cost of a query plan; consider plans with binary joins in which the inner relation is a base relation (left-deep plans); and postpone Cartesian product after joins with predicate. Following this, a number of optimization techniques for databases systems were proposed. The representatives include join-ordering strategies, and techniques that combine a bottom-up evaluation with top-down propagation of query variable bindings in the spirit of the Magic-Sets algorithm [8]. Join-ordering strategies may be heuristic-based or cost-based; some cost-based approaches depend on the estimation of the join selectivity; others rely on the fan-out of a literal [12]. All of these database query optimization techniques are designed for situations where data of different database relations are stored in the same file. However, in the Semantic Web, it is very common that data from the same relation is spread among many files. If the available indices do not completely specify the triples contained in a document, then high latency makes determining the extensions of the relations in these files very expensive. In such situations, query plans need to be developed incrementally.

In RDF query optimization, 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 recent years, many researchers have proposed ways of optimizing SPARQL join queries. MonetDB [11] exploits the fact that RDF data typically has many fewer predicates than triples, thereby vertically partitioning the data for each unique predicate and sorting each predicate table on subject, object order. RDF-3X [6] and Hexastore [16] attempt to achieve scalability by replicating each triple six times (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. YARS2 [3] 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. However, in this way, if more efficient query processing can be achieved, more disk space will be needed. GRIN [15] is a novel index developed specially for RDF graph-matching queries and focuses on path-like queries that cannot be expressed using existing SPARQL syntax. 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. [14] proposed Hermes, which translates a keyword query provided by the user into a federated query and then decomposes this into separate 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). Stuckenschmidt et al. [13] proposed a global data summary for locating data matching query answers in different sources and optimizing the query. However, this method does not consider the heterogeneity of schemas of the distributed ontologies.

Most of the research on query answering over distributed schemas or ontologies are based on the P2P architecture. Piazza [2] proposes a language (based on XQuery/XPath) to describe the semantic mapping between two different ontologies. In this work, a peer reformulates a query by using the semantic mapping and forwards the reformulated query to another peer related by the semantic mapping. DRAGO [10] 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. The semantic mapping supported in DRAGO is only the subsumption relationship between two atomic concepts and ABox reasoning is not supported. KAONP2P [1] 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. To support federated 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.

3 Query Optimization

In this section, we present some preliminary definitions regarding the distributed environment for our algorithm, the inverted term index, and the algorithms of our prior work. We then describe the novel tree-structure algorithm in detail.

3.1 Preliminaries

In the Semantic Web, there exist many ontologies, which can contain classes, properties and individuals. We assume that the assertions about the ontologies are spread across many data sources, and that mapping ontologies have been defined to align the classes and properties of the domain ontologies. For convenience of analysis, we separate ontologies (i.e. the class/property definitions and axioms that relate them) and data sources (assertions of class membership or property values). Formally, we treat an ontology as a set of axioms and a data source as a set of RDF triples. A collection of ontologies and data sources constitute what we call a *semantic web space*:

Definition 1. *(Semantic Web Space) A Semantic Web Space SWS is a tuple* $\langle D, o, s \rangle$*, where D refers to the set of document identifiers, o refers to an ontology function that maps D to a set of ontologies and s refers to a source function that maps D to a set of data sources.*

We have chosen to focus on *conjunctive queries*, which provide the logical foundation of many query languages (SQL, SPARQL, Datalog, etc.). A conjunctive query has the form $\langle \overline{X} \rangle \leftarrow B_1\left(\overline{X}_1\right) \wedge \ldots \wedge B_n\left(\overline{X}_n\right)$ where each variable appearing in $\langle \overline{X} \rangle$ is called a distinguished variable and each $B_i(\overline{X}_i)$ is a query triple pattern (QTP) $\langle s_i, p_i, o_i \rangle$, where s_i is a URI or variable, p_i is a predicate URI, and o_i is a literal, URI, or variable. Given a Semantic Web Space SWS, the answer set $ans(SWS, \alpha)$ for a conjunctive query α is the set of all substitutions θ for all distinguished variables in α such that: SWS $\models \alpha\theta$[1]. In this definition, the entailment relation $\models$ is defined in the usual way, albeit with respect to the conjunction of every ontology and data source in the Semantic Web Space.

Our problem of interest is given a Semantic Web Space, how do we *efficiently* answer a conjunctive query? Recall, we are assuming that we do not have a local repository for the full content of data sources and due to network latency, we need to minimize the number of sources that we will load to ascertain their actual content. Therefore we need to prune sources that are clearly irrelevant and focus on those that might contain useful information for answering the query. Here, we consider a system architecture where an **Indexer** is periodically run to create an index for all of the data sources and to collect the axioms from domain and mapping ontologies. Given a conjunctive query, the **Reformulator** uses the domain and mapping ontologies to produce a set of query rewritings. A **Selector** takes these rewritings and uses the index to identify which sources are potentially relevant to the query (note, since the index is an abstraction, we cannot be certain that a source is relevant until we load it). Then 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.

In our prior work [5], we showed that a term index could be an efficient mechanism for locating the documents relevant to queries over distributed and heterogeneous semantic web resources. 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. Formally, for a given document d, the terms contained in d can be expressed as following:

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

where $\langle s, p, o \rangle$ stands for a triple contained in document d, U is the set of URIs, L is the set of Literals and *lit-terms*() is a function that extracts terms from literals, and may involve typical IR techniques such as stemming and stopwords. The term index can then be defined as follows:

Definition 2. *(Term Index) Given a Semantic Web Space* $\langle D, o, s \rangle$*, the term index is a function* $I : T \rightarrow \mathcal{P}(D)$ *, where* $T = \bigcup_{d \in D} terms(s(d))$.

[1] $\alpha\theta$ is a shorthand for applying θ to the body of α, i.e., $B_1\theta \wedge B_2\theta \ldots \wedge B_n\theta$.

Using the term index we can define two functions that together determine how to select potentially relevant sources using the term index. Note that the sources for a QTP are basically those sources that contain each constant (URI or literal term) in the QTP.

Definition 3. *(Term Evaluation) Given the set of possible query triple patterns Q and a set of constant terms T (that appear as subjects, predicates or objects of any $q \in Q$), the term evaluation function qterms: $Q \rightarrow \mathcal{P}(T)$ maps QTPs to the (non-variable) terms that appear in them.*

Definition 4. *(Source Evaluation) Given the set of possible query triple patterns Q and a set of document identifiers D, the source evaluation function is qsources: $Q \rightarrow \mathcal{P}(D)$. Given a QTP q and a term index I, $qsources(q) = \bigcap_{c \in qterms(q)} I(c)$.*

Subsequently, we proposed a *flat-structure* query optimization algorithm [4] where the **Selector** took a set conjunctive query rewrites as input and then locally optimized each of them. Since loading sources is the primary bottleneck of this type of system, we focused on optimizing the *source selectivity* – the total number of sources loaded. We define the source selectivity of a selection procedure *sproc* for a query α as the number of sources not selected divided by the total number of sources available:

$$Sel_{sproc}(\alpha) = \frac{|D| - |sproc(\alpha)|}{|D|} \tag{1}$$

The flat-structure algorithm is based on the simple observation that the join selectivity of a pair of QTPs is often higher than the overall selectivity of these two QTPs treated independently. Consider two QTPs q_1 and q_2 from the same conjunctive query that share a variable x, in database parlance this situation is called a *join condition* and x is the *join variable*. We note that the number of sources required to answer the query are often less than $qsources(q_1) \cup qsources(q_2)$. If we load the sources for q_1 first, we can find a set rs of variable bindings for the QTP from the triples contained in the sources. We can then apply each substitution $\theta \in rs$ to q_2 to generate a set of queries and get a set of sources for q_2 by doing index lookups for each: $\bigcup_{\theta \in rs} qsources(q_2\theta)$. It should be clear that by adding an additional constant to each QTP, this join approach often has a higher source selectivity than naively applying *qsources* to each QTP in the query, although note that the *join selectivity* depends on which QTP is processed first. The flat-structure algorithm iteratively loads the most selective QTP and uses its substitutions to calculate the join selectivity for all remaining QTPs.[2] The two main problems with this algorithm are:

[2] Here, we assume that all data sources are relatively small; the presence of very large data sources may lead to an issue where a QTP that has high source selectivity actually has low answer selectivity. Such problems could be addressed by keeping additional size statistics in the index.

- In order to avoid complications with inference impacting the number of sources for each QTP, it repeats the source selection procedure for each possible query rewrite. However, when there is significant heterogeneity in the ontologies, synonymous ontology expressions can lead to an explosion in the number of query rewrites. Processing a large number of rewrites can slow the system down, even if we cache the results of index lookups and are careful not to load the same source multiple times.
- The inability to use the full structure of query rewrites reduces the possible source selectivity of the query process. Since source selection is independently executed for each query rewriting, selectivity is based only on local information, and does not account for the possibility that a subgoal that initially appears selective actually is not selective once all of its rewrites are taken into consideration.

3.2 Tree Structure Query Optimization Algorithm

To address the issues discussed in the previous section, we propose to replace the **Selector** component of our previous architecture with a *tree-structure* query optimization algorithm that takes a rule-goal tree expressing the query reformulation as its input and performs a greedy, bottom-up analysis of which sources to load. A rule-goal tree is basically an AND-OR graph, where goal nodes are labeled with QTPs (or their rewritings), and rule nodes are labeled with ontology axioms [2]. The purpose of the rule-goal tree is to encapsulate all possible ways the required information could be represented in the sources. See Figure 1 for an example of a rule goal tree for query with three QTPs. In this example, a property composition axiom $r0$ from a mapping ontology has been used to rewrite *swat:makerAffiliation* as the conjunction of *swrc:affiliation* and *foaf:maker*. Qasem et al. [7] have shown how to produce such rule-goal trees when all ontologies are expressed in OWLII, a subset of OWL DL that is slightly more expressive than Description Horn Logic.

We begin with an example to provide the intuition for our algorithm, and then discuss its details subsequently. Consider the rule-goal tree in Fig. 1 for the query Q, which asks for the publications affiliated with Lehigh University (“*lehigh-univ*”), complete with the ids and names of their authors. In the diagram, each goal node has three associated costs: the *initial-cost* is the number of sources relevant to that goal if we do not consider any axioms, the *local-optimal-cost* is the number of relevant sources after applying available constant constraints and the *total-cost* is the number of sources after applying available constant constraints and collecting sources from the descendants. Additionally, the order in which we process goal nodes is indicated by the parenthesized numbers. The first step is to use the term index to initialize the tree with source selectivity information, represented by initial-costs next to each goal node. We start with the QTP leaf node that selects the fewest sources: *<?m,akt:has-affiliation,“lehigh-univ”>* (labeled with (1)). Since this is an OR node, we simply propagate its sources up to its parent goal. Thus, the total-cost for *<?m,swrc:affiliation,“lehigh-univ”>* is updated to 60 (40 sources from its child plus 20 sources of itself; for simplicity of

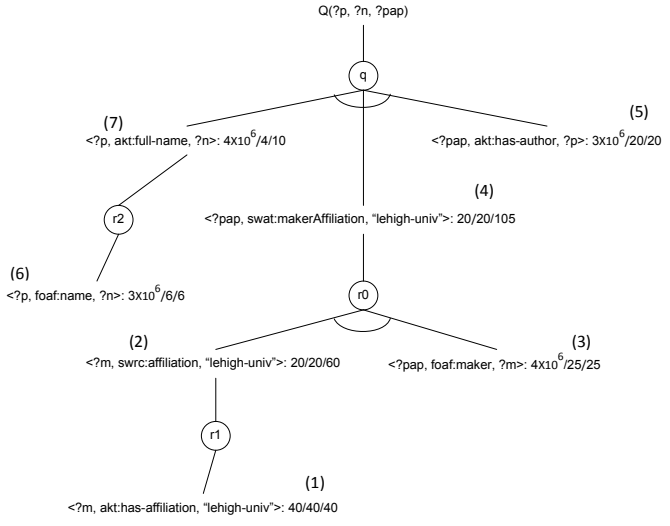

Fig. 1. Query resolution of one sample query with notations in form of initial-cost/local-optimal-cost/total-cost

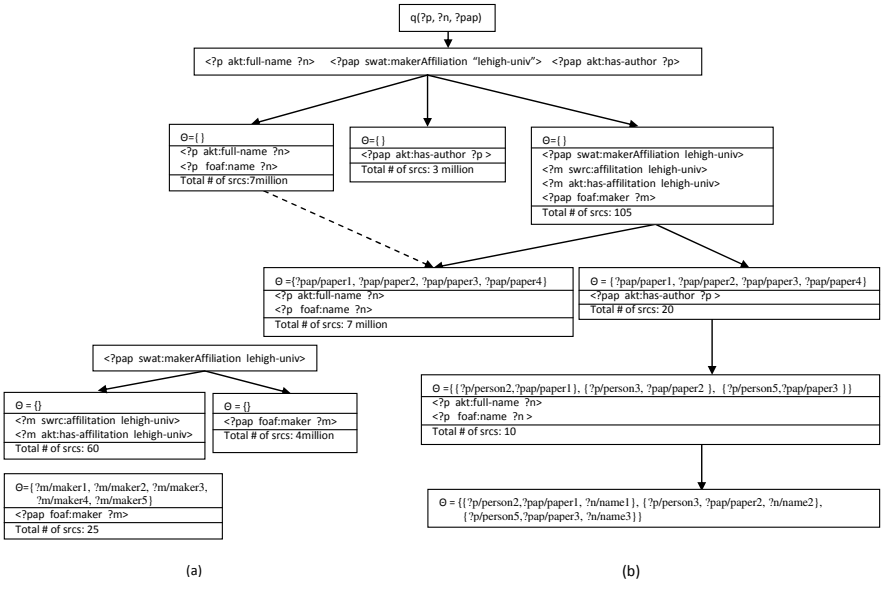

Fig. 2. AND-optimization. At each level of the tree a QTP is chosen greedily, its sources loaded and queried, and the answers applied to sibling QTPs.

exposition we are assuming that the sets of sources are disjoint, but this is not a requirement for the algorithm). Since all children of <*?m,swrc:affiliation, "lehigh-univ"*> have been processed, it joins the leaf nodes as a candidate for processing, and since it's total cost is 60, which is less than the initial costs of all other candidates, it is the next node to be processed. Since it is a child of $r0$, an AND rule node (indicated by the arc), we can use it to constrain its sibling *foaf:maker* as shown in Fig. 2(a). First, we load all sources associated with the goal node and issue the goal as a query for these sources. This query results in the substitutions for $?m$: $\{?m/maker1, ?m/maker2, ...\}$. Each of these substitutions is then applied to <*?pap,foaf:maker,?m*>, an index lookup is performed for each resulting QTP, and the total set of sources (in this case 25 of them) is used to update the total cost of this node in Figure 1, step (3). In step (4), the total cost of these nodes (60+25=85) is propagated to their parent *swat:makerAffiliation*, and is added to its initial cost (20), resulting in a total cost of 105. Since this node now has the best selectivity and is the child of an AND rule node (the original query), we need to perform another AND optimization as show in Fig. 2(b). As shown, once this node is selected, there are two siblings to choose from. However, before we can determine the cost of these nodes, we must repeat the tree process on the subtrees rooted at these nodes, thus the number of sources for <*?p,akt:full-name,?n*> is 7 million, the sum of its sources and the sources of its child <*?p,foaf:name,?n*>. We apply the substitutions from *swat:makerAffiliation* to each sibling, resulting in the number of sources of *akt:has-author* being reduced to 20 (updating its local-optimal-cost in Fig. 1), but not changing the sources of *akt:full-name*. In step (5) of Fig. 1, we select *akt:has-author*, load its sources, issue a combined query with the previous goal, and get a new set of substitutions. These substitutions are then applied to the subtree of *akt:full-name*, changing the local-optimal-costs of *foaf:name* and *akt:full-name* to 6 and 4, respectively, and changing the total-cost of *akt:full-name* to 10. As a result, the total number of collected sources for the given conjunctive query is $105 + 20 + 10 = 135$, compared to over 11 million if no optimization was done. Once all sources are loaded, we can ask the original query of the reasoner in order to get a final set of substitutions.

The pseudo code for our algorithm is shown in Figure 3. Algorithm 1 processes a rule-goal tree, where the parameter rs, which provides a set of substitutions, is $\emptyset$ when first called, but instantiated in recursive calls. We use *frontier* to maintain a set of deepest, unprocessed goal nodes in the rule-goal tree; this is initialized to be the set of leaf nodes. In Lines 2-4, we use the term index to determine the initial selectivity of all goal nodes in the rule-goal tree. Then, the most selective node n is chosen from the *frontier* (Line 6). We check if n is a child of an AND rule, and if so Algorithm 2 is called to collect sources by using the greedy strategy (Lines 7-8). If the rule is an OR mapping, the sources from the rule children are directly broadcast upward to the rule parent goal node p (Lines 9-10). Since this completes the processing of n, we remove it from our *frontier* node set (Line 11) and if p currently has no descendants in $frontier$,

Algorithm 1 Source selection for structure-based query optimization

```
function getSourceList(rtree, rs) returns a list of sources
    inputs: rtree, a given rule goal tree rtree
            rs, a list of substitutions
1:  Let frontier = leaf nodes of rtree,
        srcs[] = array of sets of sources, indexed by goal nodes
2:  for each goal node n in rtree do
3:      for each θ ∈ rs do
4:          srcs[n] = qsources(nθ)
5:  do
6:      Let n = min_{node ∈ frontier} (|srcs[node]|), p = getParent(n)
7:      if n is a child of an AND rule node r then
8:          srcs[p] = srcs[p] ∪ OptimizeANDNode(n,
                        siblings of n, srcs)
9:      else
10:         srcs[p] = srcs[p] ∪ srcs[n]
11:     remove n from frontier
12:     if p has no descendants on frontier then
13:         add p to frontier
14: while (frontier ≠ {rtree.root})
15: return srcs[rtree.root]
```

Algorithm 2 Node optimization

```
function OptimizeANDNode(on, sibs, srcs) returns a list of sources
    inputs: on, a given goal node in the rule-goal tree
            sibs, a set of on's sibling nodes
            srcs, an array of sets of sources, indexed by goal nodes
1:  Let allsrcs = ∅, query = true
2:  allsrcs = allsrcs∪srcs[on]
3:  load(srcs[on], KB)
4:  do
5:      Let query = query ∧ on
6:      Let rs = askReasoner (KB, query)
7:      for each qtp ∈ sibs do
8:          srcs[qtp] = getSourceList(subtree rooted at qtp, rs)
9:      Let on = min_{t ∈ sibs that join with query} (srcs[t])
10:     Remove on from sibs
11:     allsrcs = allsrcs∪srcs[on]
12:     load(srcs[on], KB)
13: while (sibs ≠ ∅)
14: return allsrcs
```

Fig. 3. Pseudo code of tree-structure source selection algorithm

we add p to the *frontier* (Lines 12-13). When the *frontier* contains only the root of the given rule-goal tree, the *while* loop terminates and our source collection ends (Line 14). Finally, all collected sources are returned (Line 15).

In Algorithm 2 we optimize an AND node, given a most selective goal node *on*, its siblings *sibs*, and an array of the sources for each node in the tree (the latter is used as an output parameter to update the log of sources found for each node). We start by loading *on*'s sources into the knowledge base KB. Then, we evaluate *on* by asking the reasoner to get the substitutions of the variables contained in *on* (Lines 5-6). These substitutions are then applied to *on*'s siblings to enhance their individual selectivity (Lines 7-8). Note the recursive call to $getSourceList()$ in line 8; this ensures that any new constraints specified by rs are effectively applied to the subtree rooted at each sibling. Based on the new selectivity estimations, we choose the next most selective node that shares a join variable with the partial query to be the next *on* (Line 9). Then we remove *on* from *sibs*, add its sources to the sources retrieved so far, and load any newly selected sources (Lines 10-12). In the next iteration, *on* is conjuncted with the partial query *query*, the reasoner is queried, and the substitutions applied again to the siblings. This process is repeated until all sibling nodes of the initial given goal node are processed (Line 13). Finally, the sources collected by the current AND mapping rule are returned (Line 14). As an aside, the flat-structure algorithm essentially executes a variation of Algorithm 2 for every conjunctive query rewrite.

4 Evaluation

To evaluate our query optimization algorithm, we have conducted two experiments based on a synthetic data set and a real world data set respectively. The first experiment compares our tree-structure source selection algorithm to our previous non-structure [5] and flat-structure [4] source selection algorithms using a synthetic dataset with significant ontology heterogeneity. The second experiment tests the scalability and practicality of our algorithm using a subset of the real world Billion Triple Challenge (BTC) data set. For both experiments, we use a graph-based synthetic query generator to produce a set of queries that are guaranteed to have at least one answer each. These queries range from one to thirteen triples, have at most nine variables each, and each QTP of each query satisfies the join condition with at least one sibling QTP. All of our experiments are done on a workstation with a Xeon 2.93G CPU and 6G memory running UNIX. Our **Indexer** component is implemented using Lucene while our **Reasoner** is KAON2.

4.1 Heterogeneity Evaluation Using a Synthetic Data Set

Our first experiment compares the tree-structure algorithm to the non-structure and the flat-structure algorithms using a synthetic data generator that is designed to approximate realistic conditions. First, we ensure that each generated file is a connected graph, which is typical of most real-world RDF files. Based on a random sample of 200 semantic web documents, we set the average number of triples in a generated document to be 50. In order to achieve a very heterogeneous environment, 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 400. This configuration resulted in an index size of 75.3MB, which was built in 21.6 seconds. We issued 240 random queries, grouped by the number of unconstrained QTPs (from 0 to 10), where an unconstrained QTP is one with variables for both its subject and object or with an *rdf:type* predicate paired with a variable subject. For each group, we computed the average query response time, average number of selected sources and average number of index accesses. Due to the exponential increase in query response time, we only executed queries with up to 5 unconstrained QTPs for both the non-structure and flat-structure algorithms.

Fig. 4(a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs. From this result, we can see that the tree-structure algorithm and flat-structure algorithm are faster than the non-structure algorithm. The reason is that unconstrained QTPs are typically the least selective; thus, the more unconstrained QTPs there are, the more opportunities there are for the two optimization algorithms to use constraints to enhance the selectivity of goals. However, the benefits of the tree-structure algorithm become really noticeable for 5 or more unconstrained QTPs; in this situation the flat-structure algorithm begins to reveal exponential behavior while

the tree-structure algorithm remains linear. This is because complex mapping ontologies can lead to a number of conjunctive query rewrites that is exponential in the size of the query.

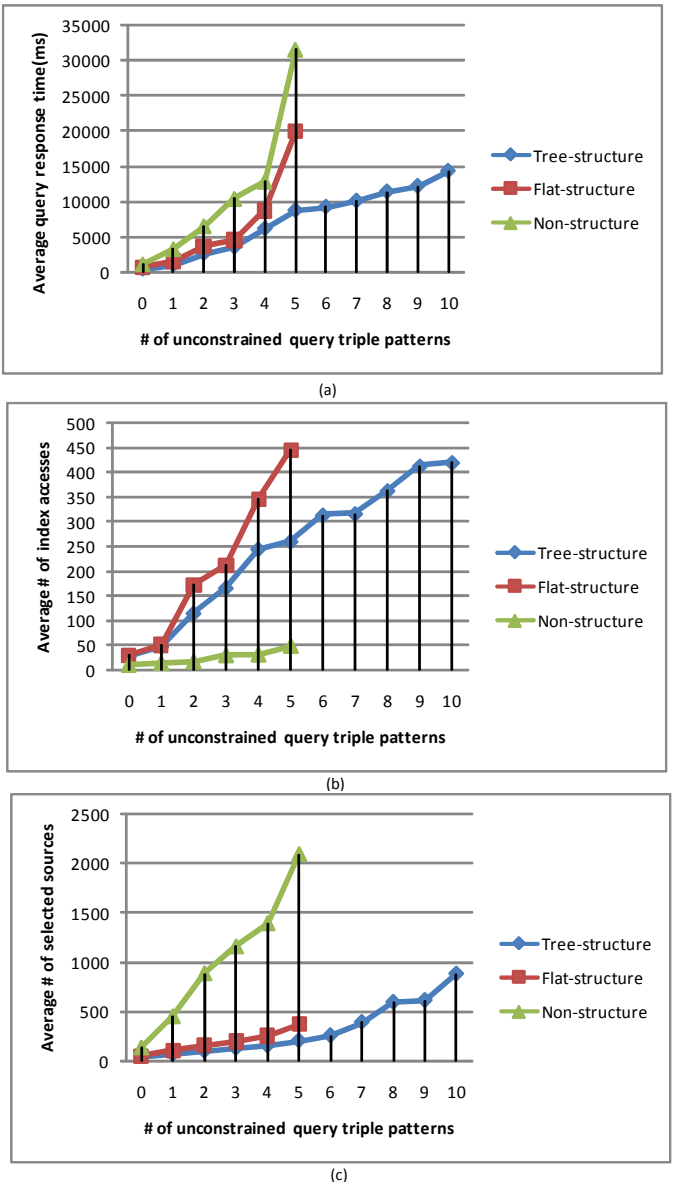

Fig. 4. Synthetic Semantic Web Space experimental results. Average query response time (a), index accesses (b) and number of selected sources (c) as the number of unconstrained QTPs varies.

Fig. 4(b) shows how each algorithm's average number of index accesses is affected by the number of unconstrained QTPs. Note the index is stored on disk and is optimized for fast lookups, but a large number of accesses can have a noticeable impact on performance. From this result, we can see that the tree-structure and flat-structure algorithms require more index accesses than the non-structure algorithm: for 5 unconstrained QTPs they require 5.3x and 9.1x more accesses, respectively. This is because both algorithms take into account the query structure information while solving the original query and might need several index lookups for the same query subgoal but using different substitutions. However, the tree-structure algorithm has 58% fewer index accesses than the flat-structure algorithm. The reason is that when using the flat-structure algorithm, one QTP can appear in multiple query rewritings and receive constraints from different sets of siblings representing different rewrites, while in the tree-structure algorithm the constraints of a sibling already consider all possible rewrites of the sibling.

Fig. 4(c) shows how the number of unconstrained QTPs impact the average number of selected sources for each algorithm. From this result, we can see the selectivity of the tree-structure and the flat-structure algorithms are roughly linear, while the non-structure algorithm is exponential in the number of unconstrained QTPs. Furthermore, the tree-structure algorithm has a gentler slope for its source selectivity than the flat-structure algorithm. Note, loading sources is the primary bottleneck of the system, since it requires that triples be read from the disk or network. The similar trends in Fig. 4(a) and Fig. 4(c) reflect the importance of source selectivity to overall query response time.

4.2 Scalability Evaluation Using the BTC Data Set

In this section, we evaluate our algorithm's scalability by using a subset of the BTC 2009 data set (much of which comes from the Linking Open Data Project Cloud). We have chosen four collections, as summarized in Table 1, with a total of 73,889,151 triples. Using the provenance information in the BTC, we re-created local N3 versions of the original files from the BTC resulting in 21,008,285 data sources. The size of these data sources varies from roughly 5 to 50 triples each. In order to integrate the four heterogeneous collections, we manually created some mapping ontologies, primarily using *rdf:subClassOf* and *rdf:subPropertyOf* axioms (these schemas do not have any meaningful alignments that are more complex). 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. Our index construction time is around 58 hours and its size is around 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.

Table 1. Data sources selected from the BTC 2009 dataset

Data Source	Namespace	# of Sources	# of Triples
http://data.semanticweb.org/	swrc	41,974	174,816
http://sws.geonames.org/	geonames	2,324,253	14,866,924
http://dbpedia.org	dbpedia	10,615,260	48,694,372
http://dblp.rkbexplorer.com	akt	8,026,878	10,153,039
Total		21,008,285	73,889,151

Because the non-structure algorithm does not refine goals with constraint information from related goals, it cannot scale to the BTC data set. In fact, most of our synthetic queries cannot be solved by this algorithm. For example, consider the query Q:{$\langle <?x_0, swrc{:}affiliation, \text{“}lehigh-univ\text{”}\rangle.\langle ?x_2, akt{:}has-title, \text{“}Hawkeye\text{”}\rangle.\langle ?x_2, foaf{:}maker, ?x_0\rangle.\langle ?x_0, akt{:}full-name, ?x_1\rangle$}. 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 tree-structure and flat-structure algorithms can deal with it because the number of sources for the same QTP becomes 114 after variable constraints are applied. For this reason, we only compare the tree-structure and flat-structure algorithms here.

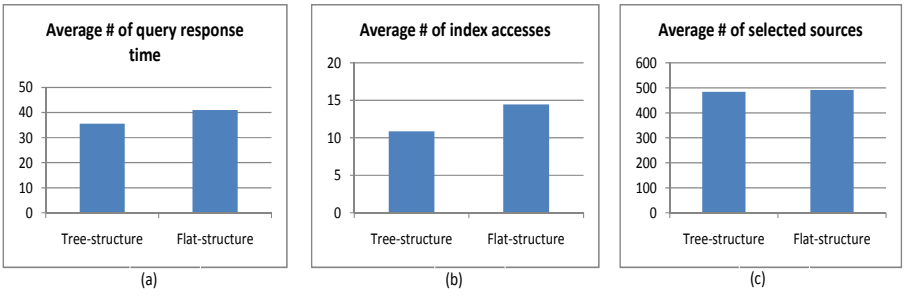

Fig. 5. BTC data set experimental results

We executed 150 synthetic queries with at most 10 QTPs and computed the same metrics as for the prior experiment. As shown in Fig. 5(a), the average query response time of the tree-structure algorithm is 35 seconds, which is a 13% improvement over the flat-structure algorithm. At the same time, it has 25% fewer index accesses as shown in Fig. 5(b). Fig. 5(c) shows that both algorithms select on average between 450 and 500 sources, and the tree-structure algorithm only shows a 1.6% improvement over the flat-structure algorithm here. We attribute this to the fact that the semantic mappings of the BTC experiment are not as complex as those for the synthetic data set, which leads to a small number of rewrites for each query. when there are potentially many rewrites for

a query. We posit that in real-world settings where more ontologies are involved, that the superiority of the tree-structure algorithm will be more pronounced.

5 Conclusions, Limitations and Future Work

We have proposed a tree-structure optimization algorithm for integrating millions of data sources that commit to different ontologies. Given a reformulation tree, this algorithm uses a bottom-up process to select sources and uses the selectivity of each goal node as a heuristic to optimize and plan the query execution. Our experiments have demonstrated that this new algorithm is better than both of our prior algorithms [4] [5] in that not only does it demonstrate query response time performance that is linear with respect to the number of unconstrained QTPs, it also has better source selectivity and requires fewer index accesses than the flat-structure algorithm. Meanwhile, we have also shown that our algorithm scales well, allowing many complex randomly generated queries against 20 million heterogeneous data sources to complete in 35 seconds.

Despite showing initial promise there are a number of limitations to the work in its present form. First, the algorithm focuses on conjunctive queries, and does not consider richer features of SPARQL such as OPTIONALs. In addition, in order to avoid the computational challenges of higher-order logics, it does not allow variables in the predicate position. Second, the implementation only works with OWLII, a subset of OWL DL, although any rewriting algorithm that produces an AND-OR reformulation tree could be used. Since finite reformulation trees cannot express rewrites of a query whose reformulation involves cyclic rules, completeness is only guaranteed for acyclic OWLII axioms. We note that this algorithm is designed for a setting where there are large numbers of small RDF files, and that it is not intended to issue queries to large SPARQL end points. Fortunately, due to Linked Data guidelines, we note that most large SPARQL end points expose an interface where a URL can be dereferenced to retrieve a small set of RDF triples describing each instance. The algorithm assumes that a correct set of mapping ontologies has been provided, and we note that any errors in these mappings can result in a loss of "semantic fidelity." Finally, the current algorithm is not guaranteed to find all relevant sources if there are *owl:sameAs* statements in the Semantic Web Space.

Our future work includes attempting to improve the selectivity of our algorithm even further and addressing many of its limitations. We believe it is possible to make better estimates about the selectivity of a node by maintaining upper and lower bounds and we will also look at storing additional statistics in our index. With respect to the limitations mentioned in the previous paragraph, we think that the most critical need is to adapt the algorithm to locate relevant *owl:sameAs* statements, which must necessarily be an iterative process in order to find their transitive closure. We believe that this paper provides a major step towards a pragmatic solution for querying a large, distributed, and ever changing Semantic Web.

References

1. Haase, P., Wang, Y.: A decentralized infrastructure for query answering over distributed ontologies. In: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC 2007, pp. 1351–1356. ACM, New York (2007)
2. Halevy, A.Y., Ives, Z.G., Madhavan, J., Mork, P., Suciu, D., Tatarinov, I.: The Piazza peer data management system. IEEE Trans. Knowl. Data Eng. 16(7), 787–798 (2004)
3. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A federated repository for querying graph structured data from the web. In: The Semantic Web, pp. 211–224 (2008)
4. Li, Y., Heflin, J.: Query optimization for ontology-based information integration. In: CIKM 2010. ACM, New York (2010)
5. Li, Y., Qasem, A., Heflin, J.: A scalable indexing mechanism for ontology-based information integration. In: IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (2010)
6. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: Proceedings of the 35th SIGMOD International Conference on Management of Data, SIGMOD 2009, pp. 627–640. ACM, New York (2009)
7. Qasem, A., Dimitrov, D.A., Heflin, J.: Efficient selection and integration of data sources for answering semantic web queries. In: International Conference on Semantic Computing, pp. 245–252 (2008)
8. Ramakrishnan, R., Ullman, J.D.: A survey of research on deductive database systems. Journal of Logic Programming 23, 125–149 (1993)
9. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, SIGMOD 1979, pp. 23–34. ACM, New York (1979)
10. Serafini, L., Tamilin, A.: Drago: Distributed reasoning architecture for the semantic web. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 361–376. Springer, Heidelberg (2005)
11. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-store support for rdf data management: not all swans are white. PVLDB 1(2), 1553–1563 (2008)
12. Staudt, M., Soiron, R., Quix, C., Jarke, M.: Query optimization for repository-based applications. In: Proceedings of the 1999 ACM Symposium on Applied Computing, SAC 1999, pp. 197–203. ACM, New York (1999)
13. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., Houben, G.: Towards distributed processing of RDF path queries. Int. J. Web Eng. Technol. 2(2/3), 207–230 (2005)
14. Tran, T., Wang, H., Haase, P.: Hermes: Data web search on a pay-as-you-go integration infrastructure. Web Semantics 7(3), 189–203 (2009)
15. Udrea, O., Pugliese, A., Subrahmanian, V.S.: Grin: A graph based RDF index. In: AAAI, pp. 1465–1470 (2007)
16. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB 1(1), 1008–1019 (2008)