Handling Cyclic Axioms in Dynamic, Web-Scale Knowledge Bases

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 recent years, there has been an explosion of publicly available Semantic Web data. In order to effectively integrate millions of small, distributed data sources and quickly answer queries, we previously proposed a tree structure query optimization algorithm that uses source selectivity of each query subgoal as the heuristic to plan the query execution and uses the most selective subgoals to provide constraints that make other subgoals selective. However, this constraint propagation is incomplete when the relevant ontologies contain cyclic axioms. Here, we propose an improvement to this algorithm that is complete for cyclic axioms, yet still able to scale to millions of data sources.

Keywords: Information Integration, Cyclic Axioms, Magic Sets, Equality Reasoning

1 Introduction

In recent years, there has been an explosion of publicly available distributed Semantic Web data. These data are often small (around 50 RDF triples per source), which is supported by the fact that many large data sets such as DBpedia, GeoNames and DBLP provide dereferenceable URIs for each of their instances; we treat each such URI as a lightweight source. In order to effectively integrate these Semantic Web data sources and answer queries, we proposed an inverted index mechanism and a complete non-structure query answering algorithm using this index [6]. Because this index only indicates whether URIs or literals are present in a document, the non-structure algorithm can not scale to the real Semantic Web. Therefore, we subsequently proposed a tree-structure query optimization algorithm that uses source selectivity of each query subgoal as the heuristic to plan the query execution by selecting a small subset of relevant sources from millions of possible sources [5]. This algorithm was designed for OWLII ontologies (the intersection of OWL with GAV and LAV rules) - a subset of OWL DL (Description Logic) [10]. However, it is incomplete when cyclic axioms are considered because it does not load all relevant sources that correspond to a query subgoal, but instead only loads those that contain the subgoal predicate and its available variable constraints. On the other hand, each iteration in the cyclic axiom could generate recursive variable constraints that

can be propagated into the following iterations. Consequently, without the fix point computation of cyclic axioms, the tree-structure algorithm will miss those sources collected by applying the recursive variable constraints in each iteration. Therefore, in order to guarantee completeness, we need special treatment for cyclic axioms. Furthermore, this process should be dynamic because data on the web is constantly in flux.

In DL, a cyclic axiom is one that references the same (or equivalent) classes (or properties) on both sides of the subsumption relation. Such an axiom may be explicit or inferred. For instance, $\exists P.C \sqsubseteq C$ is a cyclic axiom, where C is a class and P is a property. $P \circ P \sqsubseteq P$ is also a cyclic axiom, where P actually stands for a transitive property. Note, $rdfs{:}subClassOf$, $rdfs{:}subPropertyOf$, $owl{:}equivalentClass$ and $owl{:}equivalentProperty$ are not cyclic unless they are used to define a class/property to be a subclass/subproperty of itself. To the best of our knowledge, no one has calculated how many cyclic axioms are used in real world ontologies. However, Wang et al. [13] surveyed 1275 ontologies and found 39 (3%) that contained a transitive property. Instance coreference is a special case of cyclic axiom because $owl{:}sameAs$ is a ubiquitous transitive property as defined: $owl{:}sameAs \circ owl{:}sameAs \sqsubseteq owl{:}sameAs$. The Billion Triple Challenge 2010 data set has 6,932,678 URI resources connected by 8,711,398 unique $owl{:}sameAs$ statements [2]. The graph made of these $owl{:}sameAs$ statements consists of 2,890,027 weakly connected components. Most components are pairs of nodes joined by $owl{:}sameAs$ links. This observation implies that the typical $owl{:}sameAs$ network is small but not ignorable.

Although handling cyclic axioms is routine for typical inference algorithms, they present challenges when querying large distributed Knowledge Bases (KBs). Some related but different work has been proposed. Pan et al. described a fix point computation algorithm for cyclic axioms in DLDB3 [9]. Mei et al. discussed the fix point computation of cyclic axioms on ontology query answering over databases [8]. Urbani et al. proposed a MapReduce reasoner to deal with the fix point computation of $owl{:}sameAs$ triples [12]. Qasem et al. presented an extended GNS algorithm to handle instance coreference [11]. Their main drawback is that they all precompute (lacking flexibility) rather than dynamically compute the fix point of cyclic axioms for centralized KBs as opposed to large distributed KBs. In addition, Lam et al. [4] proposed an approach to blocking the expansion of cyclic axioms in order to guarantee termination of the computation during their cycle handling. However, this process aims to resolve the unsatisfied ontology but not for query answering over large distributed KBs. Another important approach of handling cyclic axioms is *Magic Sets* [1], which is a general algorithm for rewriting logical rules to compute the fix point of cyclic axioms. It applies the sideways information passing strategy (SIPS) and improves query answering efficiency by restricting the computation to facts that are related to the query. Since SIPS is basically similar to our constant constraint propagation, we incorporate the *Magic Sets* theory into our algorithm. More details will be given in Sections 2 and 3. Therefore, inspired by the traditional *Magic Sets* theory and based on the tree-structure algorithm [5], we propose a dynamic cyclic

axiom handling algorithm for query answering over large distributed KBs. Our main contributions are: 1) we develop a dynamic stack-based cyclic axiom handling algorithm, which dynamically computes the fix point of cyclic axioms and does instance equality reasoning (*owl*:*sameAs* inference) in a separate process, 2) we demonstrate that our algorithm can perform well on both a synthetic data set with 20 ontologies having significant heterogeneity and a real world data set with 73,889,151 triples distributed among 21,008,285 documents.

The remainder of the paper is organized as follows: Section 2 describes some preliminary work. In Section 3, we describe the cyclic axiom handling algorithms for large distributed KBs. Section 4 presents the experiments that we have conducted to evaluate the proposed algorithms. Finally, in Section 5, we conclude and discuss future work.

2 Preliminaries

In this section, we first introduce some background and the main drawback of our tree-structure algorithm [5]. Then, we will describe the traditional *Magic Sets* theory [1], especially the part related to our algorithm.

2.1 Tree-structure algorithm

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 are 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 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 $\mathrm{Q}\langle\overline{X}\rangle \leftarrow B_1\left(\overline{X}_1\right) \wedge \ldots \wedge B_n\left(\overline{X}_n\right)$ 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.

Our problem is, given a SWS, how do we *efficiently* answer a conjunctive query? The key point to this problem is how to prune sources that are clearly irrelevant and focus on those that might contain useful information for answering the query. We have shown that a term index could be an efficient mechanism for

locating the documents relevant to queries over distributed and heterogeneous semantic web resources [6]. Based on the term index, we proposed an effective tree-structure algorithm [5]. Given a rule-goal tree that aims to encapsulate all possible ways the required information could be represented in the sources [3] by using the axioms in the domain ontologies and the mapping ontologies, our tree-structure algorithm uses a bottom-up process to select sources and the selectivity of each goal node as a heuristic to greedily optimize and plan the query execution. The source selectivity of a selection procedure *sproc* for a query α is defined to be the number of sources not selected divided by the total number of sources available:

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

The tree-structure algorithm always starts with the most selective QTP, incrementally loads the relevant sources, and uses the data from the sources to further constrain related $QTPs$ in order to answer queries. It is only complete for acyclic ontologies expressed in OWLII defined below:

Definition 2. *The syntax of OWLII consists of DL axioms of the forms* $C \sqsubseteq D$, $A \equiv B$, $P \sqsubseteq D$, $P \equiv Q$, $P \equiv Q^-$, *where* C *is an* L_a *class,* D *is an* L_c *class,* A, B *are* L_{ac} *classes and* P, Q *are properties.* L_{ac}, L_a *and* L_c *are defined as:*

- L_{ac} *is a DL language where* A *is an atomic class and* i *is an individual. If* C *and* D *are classes and* R *is a property, then* $C \sqcap D$, $\exists R.C$ *and* $\exists R.\{i\}$ *are also classes.*
- L_a *includes all classes in* L_{ac}. *Also, if* C *and* D *are classes then* $C \sqcup D$ *is also a class.*
- L_c *includes all classes in* L_{ac}. *Also, if* C *and* D *are classes then* $\forall R.C$ *is also a class.*

In the presence of cyclic axioms, the tree-structure algorithm becomes incomplete because the cyclic axioms require that the goal node (e.g. the coreferenced instance for *owl*:*sameAs*) be iterated over to collect sources until a fix point is reached in order to obtain the complete answers. For instance, take the cyclic Datalog axiom $ancestor(?x, ?y)$:- $ancestor(?x, ?z) \wedge ancestor(?z, ?y)$ [1] and its query $ancestor(John, ?y)$. Assuming we have collected sources containing the substitutions $\{?z/Bob, ?y/Andy\}$ by using the subgoals $ancestor(John, ?z)$ and $ancestor(?z, ?y)$ respectively on the term index, the tree-structure algorithm then finishes processing this axiom because all of its subgoals have been handled and their corresponding sources have also been collected. However, those sources containing the recursive descendants of *Bob* and *Andy* are still relevant but will be missed because the given axiom is not recursively applied. Consequently, the tree-structure algorithm is clearly incomplete.

[1] The property composition axiom is actually beyond OWLII's expressivity and its use in the paper is for the purpose of giving an example. *owl*:*sameAs* is specially handled in Section 3.2.

2.2 Magic Sets

The *Magic Sets* method executes a top-down evaluation of a query by adding rules which narrow the computation to what is relevant for answering the query. As mentioned in section 1, it applies the SIPS strategy that describes how bindings passed to a rule's head by unification are used to evaluate the predicates in the rule's body. For instance, let V be an atom that has not yet been processed, and Q be the set of already considered atoms, then a SIPS specifies a propagation $V \rightarrow_X Q$, where X is the set of the variables bound by V, passing their values to Q.

The method is structured in four steps: rule adornment, rule generation, rule modification and query processing. They are illustrated as follows by considering the axiom $ancestor(X,Y)$:- $ancestor(X,Z)$, $ancestor(Z,Y)$ together with a query $ancestor(John,Y)$, where X, Y and Z are variables and $John$ is a given instance. Note, the given axiom is cyclic.

(1) Rule adornment: this phase is to materialize, by suitable adornments, binding information for predicates. These are strings of the letters b and f, denoting bound or free for each argument of a predicate. First, adornments are created for query predicates. The adorned query is $ancestor^{bf}(John,Y)$. In the given rule, $ancestor^{bf}(John,Y)$ passes its binding information to $ancestor(X,Z)$ by $ancestor^{bf}(X,Y) \rightarrow_X ancestor(X,Z)$. Then, $ancestor(X,Z)$ is adorned $ancestor^{bf}(X,Z)$. Now, we consider $ancestor(Z,Y)$, for which there is no binding information and we can still use the given axiom to expand it. Finally, we have two resulting adorned rules: $ancestor^{bf}(X,Y)$:- $ancestor^{bf}(X,Z)$, $ancestor^{ff}(Z,Y)$ and $ancestor^{ff}(Z,Y)$:- $ancestor^{ff}(Z,W)$, $ancestor^{ff}(W,Y)$, where W is a new introduced variable.

(2) Rule generation: the adorned program is used to generate magic rules. For each adorned predicate p in the body of an adorned rule r_a, a magic rule r_m is generated such that (i) the head of r_m consists of $magic(p)$, and (ii) the body of r_m consists of the magic version of the head of r_a, followed by all of the predicates of r_a which can propagate the binding on p. In our example, two magic rules are $magic\_ancestor^{ff}(Z,Y)$:- $magic\_ancestor^{bf}(X,Y)$, $magic\_ancestor^{ff}(X,Z)$ and $magic\_ancestor^{ff}(Z,W)$:- $magic\_ancestor^{ff}(Z,Y)$, $ancestor^{ff}(W,Y)$.

(3) Rule modification: the adorned rules are modified by including magic atoms generated in Step (2) in the rule bodies. The resultant rules are called modified rules. For each adorned rule whose head is h, we extend the rule body by inserting $magic(h)$. In our example, $ancestor^{bf}(X,Y)$:- $magic\_ancestor^{bf}(X,Y)$, $ancestor^{bf}(X,Z)$, $ancestor^{ff}(Z,Y)$ and $ancestor^{ff}(Z,Y)$:- $magic\_ancestor^{ff}(Z,Y)$, $ancestor^{ff}(Z,W)$, $ancestor^{ff}(W,Y)$ are generated.

(4) Query processing: for each adorned predicate g^{α} of the query, (i) the magic seed $magic(g^{\alpha})$ is asserted, and (ii) a rule g :- g^{α} is produced. In our example, we generate $magic\_ancestor^{bf}(John,Y)$ and $ancestor(X,Y)$:- $ancestor^{bf}(X,Y)$.

The complete rewritten program consists of the magic, modified, and query rules. Given a non-disjunctive datalog program P, a query Q, and the rewritten program $P^{'}$, it is well known that P and $P^{'}$ are equivalent w.r.t. Q [1]. In

Magic Sets, the adornments of Step (1) aim to cover all possible bound/free information based on the given query and rules. Then, the generated magic rules in the following steps can easily avoid irrelevant facts while guaranteeing completeness during the fix point computation of the cyclic axioms. For our tree-algorithm, as shown by [5], the constant propagation mechanism is basically the same as the SIPS strategy, using the available binding of the rule's head to constrain the rule's body. In addition, because our purpose is to collect relevant sources by constructing boolean queries using the available constant constraint (bound value) and the predicate instead of the real computation of the fix point, which is actually accomplished by the **Reasoner**, it is sufficient for us to only incorporate the rule adornment step into our algorithm. Then, we can easily detect if two terms have the same predicate and adornment. If so, a cycle is formed and we can collect only those sources that are necessary for this cycle's fix point computation.

3 Cyclic Axiom Handling Algorithms

In this section, we first introduce the *Magic Sets*-inspired dynamic cyclic axiom handling algorithm without instance coreference. Then, we discuss the equality reasoning (instance coreference).

3.1 Cyclic Axiom Handling

To handle cyclic axioms, there are four key points we particularly need to take care of:

- How to represent and annotate cyclic axioms in the original rule-goal tree of the query reformulation?
- Within each iteration of one cyclic axiom, how to compute the new generated substitutions of the given cyclic axiom that will be passed into the next iteration? In this process, we call the set of new substitutions *Relevant Substitutions (RS)*.
- How to apply the *RS* into the selection of relevant sources by using the term index?
- In case of multiple cyclic axioms mutually nested in one query, how to identify their correct computation order?

For the first point, as the traditional *Magic Sets* theory does, we adorn the cyclic axioms by using their binding information. Then, we mark them in the rule-goal tree. In theory, if one goal node G is detected to be one that can be unified with its one ancestor goal node A on condition that G and A are the same predicate and have the same adornments, then we detect a cycle C starting with A and ending with G. However, in practice, we apply the heuristic that is if A and G also have the same bound value, then they are not a cycle because A and G collect the same sources by using the term index and there is no recursive

source collection. For instance, in Fig. 1, even though G_1 and G_2 compose a cycle, we can skip it because both of them only collect sources containing $John$ and $ancestor$. Formally, a cycle C is denoted as $C(A, G)$, where A is C's starting node and G is C's ending node. After the cycle is marked, the rule-goal tree is transformed into a rule-goal graph and each cyclic axiom will be converted into one or more rule-goal graph cycles correspondingly. Essentially, a rule-goal graph cycle means its corresponding axiom will be iteratively executed until a fix point is reached. For the second point, in the rule-goal graph, the RS of each iteration for one cyclic axiom essentially consists of the new generated substitutions of the cycle distinguished variables (CDVs) of this cyclic axiom's rule-goal graph cycles. We define each graph cycle's CDVs to be a set of the distinguished variables of the starting node of this cycle. In the previous example, C's CDVs contains all A's distinguished variables. At the end of each cycle iteration, we will compute the RS by asking the reasoner and apply it into the next iteration if the new RS is not empty. Otherwise, it means we have reached the fix point of the current cycle. Furthermore, if the RSs of all cycles in the rule-goal graph for one given cyclic axiom are empty, it means the fix point of this cyclic axiom has been reached. For the third point, we use the conjunction of each value in the RS and the goal predicate to query the term index. This helps us to significantly reduce the number of potentially relevant sources because of the constant constraints. For the fourth point, we will employ a cycle stack to plan the cyclic axiom handling sequence. Each cycle can be pushed onto the stack only if it is not already in the stack. Otherwise, we will postpone its processing.

We begin with the cyclic axiom $ancestor(?x, ?y)$:- $ancestor(?x, ?z) \wedge ancestor(?z, ?y)$ and its query $ancestor(John, ?y)$ to introduce our algorithm. Fig. 1 shows its rule-goal graph. The back arrow means a cycle is marked. Each goal node has associated adornments (bf or ff) and selectivity (the number of relevant sources).

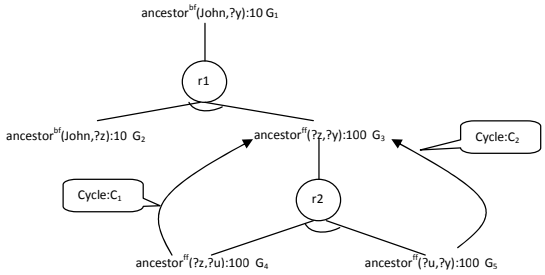

Fig. 1. An example cyclic axiom

At the beginning, using the term index, each goal node of the rule-goal graph is initialized with their respective selectivities and bindings. In this example, we have two cycles: $C_1(G_3, G_4)$ and $C_2(G_3, G_5)$. In our cycle detection, if a goal node is an ending node of some cycle, we will say a cycle is detected. We use

S to stand for our cycle stack. Initially, we start with the most selective node G_2 and use its substitutions to constrain its sibling G_3. In this process, we start with G_3's most selective node G_4 (Here, G_4 and G_5 have the same selectivity and we randomly select G_4), where C_1 is detected and pushed onto S. Then, we still start with G_4 in processing C_1. Now, C_1 is detected again and postponed because it is also already in S. We evaluate G_4 and apply its available constant substitutions into its sibling G_5, where C_2 is detected and pushed onto S. Now, S contains C_1 and C_2. We then start to process C_2 still beginning with G_4, where C_1 is detected and postponed again. Then, we evaluate G_4 and apply its substitutions to G_5, where C_2 is detected again and postponed. Now, we are at the end of one iteration of C_2, compute C_2's RS and apply it into the next iteration to select relevant sources. If the new RS is empty, it means C_2's fix point has been reached and C_2 is popped. Now, S only contains C_1. Then, we go back to the context of C_1. Obviously, in processing the next iteration of C_1, C_2 will be met again. The previous process of C_2 is repeated. Meanwhile, at the end of each C_1's iteration, we also compute C_1's RS and apply it into the next iteration to collect sources until the new RS is empty meaning C_1's fix point has been reached. Finally, we finish processing C_1 and C_2 and correspondingly collect all relevant sources of the given cyclic axiom.

Note, in the above process, $C_1(G_3, G_4)$ and $C_2(G_3, G_5)$ are actually redundant cycles because they collect the same data sources. Therefore, we need to avoid such repeated source collections. In our algorithm, we detect if two cycles are redundant, which means that each node in one cycle exactly has the same predicate and same adornments as a node in the other one and vice versa. These redundant cycles are categorized into different redundant cycle classes and stored into a structure called the Redundant Cycle Base (RCB). Each redundant cycle class is a set of cycles that are redundant to each other. Redundant cycles cause redundant source collection because they could generate the same recursive constants and then collect the same sources multiple times. Therefore, during the process of each redundant cycle, we need to check if the new recursive constant has been used by other cycles that are redundant with the current cycle. If not, we go ahead and start the next generation. Otherwise, we will skip this constant. Here, we cannot handle only one cycle instead of the whole set of redundant cycles because redundant cycles could appear in different positions of the rule-goal graph and they thus could have different recursive constants generated to collect different data sources. Then, even though $C_1(G_3, G_4)$ and $C_2(G_3, G_5)$ are both pushed onto our cycle stack in the given example, we can avoid the repeated source collection. In addition, for those instances that match query constants or that are used as join conditions, we will compute their equivalence ($owl{:}sameAs$) closure on the fly by calling our equality reasoning algorithm (Fig. 3). More details can be found in section 4.2.

The pseudo code for our cyclic axiom handling algorithms is shown in Fig. 2. The bold lines in Alg. 1 and Alg. 2 and the whole Alg. 3 are new results from this paper. In Alg. 1, RCB stands for Redundant Cycle Base defined in the previous paragraph. EKB is a structure that collects and organizes equivalence

Algorithm 1 Source selection

function getSourceList(*rgraph*, rs, *q*)**returns** a list of sources
  inputs: *rgraph*, a given rule-goal graph (cyclic or non-cyclic)
    rs, a list of substitutions
    q, a list of query triple patterns
1: Let *frontier* = leaf nodes or cycle ending nodes, **static *EKB* = ϕ, static *RCB* = ϕ**
  srcs[] = array of sets of sources, indexed by goal nodes
2: **for each** goal node *n* in *rgraph* **do**
3: **if *n* has constant *C* and $C.equivalenceClass \notin EKB$ then**
4: **computeSameAs({*C*}, *EKB*)**
5: **for each** $\theta \in rs$ **do**
6: ***srcs[n]* = qsources($n\theta$, *EKB*)**
7: **do**
8: **Let** *n* = $min_{node \in frontier}$ (|$srcs[node]$|), *p* = *n*.parent
9: **if *n* is a cycle ending node AND $n.cycle \notin CycleStack$ then**
10: **update(*n.cycle*, *RCB*)**
11: **push(*CycleStack*, *n.cycle*)**
12: $srcs[n] = srcs[n] \cup$ **getCyclicSourceList**($n.cycle, rs, q$)
13: **pop(*CycleStack*)**
14: **if** *n* is a child of an AND rule node *r* **then**
15: $srcs[p] = srcs[p] \cup$ OptimizeANDNode($rgraph$, $n, siblings\ of\ n, srcs, q, EKB, RCB$)
16: **else**
17: $srcs[p] = srcs[p] \cup srcs[n]$
18: **if *n* is a child of *rgraph.root* and *rgraph* is a cycle then**
19: ***load*(*srcs[n]*, *KB*)**
20: **Let rsc = askReasoner (*KB*, *rgraph*)**
21: **Let *insts* = extractJoinInsts(*rsc*)**
22: **computeSameAs(*insts*, *EKB*)**
23: ***rgraph.RS* = computeRS(*rgraph.CDVs*, *RCB*)**
24: remove *n* and its siblings from *frontier*
25: **if** *p* has no descendants on *frontier* **then**
26: add *p* to *frontier*
27: **while** ($frontier \neq \{rgraph.root\}$)
28: **return** *srcs[rgraph.root]*

Algorithm 2 Node optimization

function OptimizeANDNode(*rgraph*,*on*,*sibs*,*srcs*,*q*,*EKB*,*RCB*)**return** a list of sources
  inputs: *rgraph*, a rule-goal graph; *on*, a goal node
    sibs, *on*'s sibling nodes; *srcs*, an array of sets of sources
    q, a list of query triple patterns; *EKB*, the EquivalenceKB;
    RCB, the redundant cycle base
1: Let $allsrcs = srcs[on]$, load(*srcs[on]*, *KB*)
2: **do**
3: q = $q \wedge on, rs$ = askReasoner (*KB*, *q*)
4: **Let *insts* = extractJoinInsts(*rs*)**
5: **computeSameAs(*insts*, *EKB*)**
6: **for each** $qtp \in sibs$ **do**
7: *srcs[qtp]* = getSourceList(subgraph rooted at *qtp*, *rs*, *q*)
8: Let *on* = $min_{t\ \in\ sibs\ that\ \ join\ with\ query}$ ($srcs[t]$)
9: Remove *on* from *sibs*
10: $allsrcs = allsrcs \cup srcs[on]$, load($srcs[on], KB$)
11: **if *on* is a child of *rgraph.root* AND *rgraph* is a cycle AND $sibs = \emptyset$ then**
12: **load(*srcs[on]*, *KB*)**
13: **Let rsc = askReasoner (*KB*, $rgraph$)**
14: **Let *insts* = extractJoinInsts(*rsc*)**
15: **computeSameAs(*insts*, *EKB*)**
16: ***rgraph.RS* = computeRS(*rgraph.CDVs*, *RCB*)**
17: **while** ($sibs \neq \emptyset$)
18: **return** *allsrcs*

Algorithm 3 Source selection for cyclic axioms

function getCyclicSourceList(*rgraph*, rs, *q*) **returns** a list of sources
  inputs: *rgraph*, a given rule-goal graph; rs, a list of substitutions
    q, a list of query triple patterns
1: Let $rsInc = rs$, *firstIt* = true, *allsrcs* = ∅
2: **while** ($rsInc \neq \emptyset$ **OR** *firstIt*)
3: $allsrcs = allsrcs \cup$ getSourceList($rgraph, rsInc, q$)
4: $rsInc$ = *rgraph.RS*
5: clear(*rgraph.RS*), *firstIt* = false
6: **return** *allsrcs*

Fig. 2. Pseudo code of handling cyclic axioms

information about instances, which will be elaborated in section 4.2. In the given rule-goal graph $rgraph$, each goal node has been adorned with its own binding information. In line 6 of Alg. 1, $qsources$ is a source evaluation function. Given a QTP q and a term index I, $qsources(q, EKB) = \bigcap_{c \in terms(q,EKB)} I(c)$, which is essentially a set of data sources that are relevant to q [5]. The EKB is used here to collect q's relevant sources by using both q's constants and their equivalent constants in EKB. In line 9, when the current most selective QTP (on) is a cycle ending node, it means that a cycle is detected and we need to use it to update our RCB and then push it into the cycle stack (lines 10 and 11). Note, each goal node in the rule-goal graph can only be involved in one cycle as an ending node because two cycles sharing one ending node is equivalent to one cycle starting and ending at these two cycle's root nodes that has been annotated before. Then, we enter Alg. 3 to compute the cycle's fix point (line 12). In Alg. 3, we repeatedly collect sources by executing Alg. 1 if the current cycle's RS is not empty (lines 2-5). Here, the RS are computed at the end of each cycle iteration in lines 18-23 of Alg. 1 and lines 11-16 of Alg. 2 by extracting the new substitutions of the current cycle's CDVs and then passed to Alg. 3 for the recursion use. In this process, the function $extractJoinInsts(rsc)$ extracts join instances from the given subsitution list rsc (line 21 in Alg. 1, and lines

4 and 14 in Alg. 2). Its results are passed to our instance coreference handling algorithm (Alg. 4) to compute the *owl*:*sameAs* closure (lines 4 and 22 in Alg. 1, and lines 5 and 15 in Alg. 2). The function $computeRS(rgraph.CDVs, RCB)$ is to compute the RS of the given rule-goal graph $rgraph$ (line 23 in Alg. 1 and line 16 in Alg. 2). Here, for each recursive constant, we check if its redundant cycles has used it before by using the RCB and rule out it from RS if it's been used. When the fix point is reached, we will return all collected sources (line 6) and go back to Alg. 1. Then, we continue to execute line 13 in Alg. 1 to pop the processed cyclic axiom.

3.2 Equality Reasoning

Our equality reasoning is based on the heuristic that within the term index, the QTPs with constant constraints are often highly selective. For instance, given two QTPs: *owl*:*sameAs*(*rpi*:*james*, ?*y*) and *owl*:*sameAs*(?*x*, ?*y*), the first is much more selective than the second because of the specific constant *rpi*:*james*. Therefore, compared to the way of loading all sources containing the *owl*:*sameAs* predicate to compute the instance coreference closure, this way helps us significantly reduce the number of sources that are involved in the closure computation. Given a query, we call the set of all instances that are used for the instance coreference closure computation as *Relevant Instances (RI)*. Since we only compute the equivalence closure of the query constant instances and the join instances during the query solving, the cardinality of RI is often small. In our algorithm, we design an EquivalenceKB structure (EKB) that collects and organizes equivalence information about instances in RI. EKB essentially supports the disjoint set data structure operations on sets of equivalence classes of all known instances. An equivalence class in EKB is a set of instances that are equivalent to each other (explicitly or implicitly connected by *owl*:*sameAs*). Given an instance Ins in RI, we dynamically issue a boolean query "Ins" AND "*owl*:*sameAs*" to our index to find all relevant sources that contain Ins and its equivalent instances. Then, for each new discovered instance $newIns$, we further find $newIns$'s equivalent instances and merge the equivalence classes containing Ins and $newIns$. This process is repeated until no new instances are discovered. Since the cardinality of RI is often small as stated before, the computation will quickly and eventually terminate. Note, the equivalence class of each instance in RI is only computed once. The algorithm pseudo code is shown in Fig. 3.

First, we start with a set of seed instance URIs (Line 2), and use the term index to find all sources that contain each of these URIs concatenated with the "*owl*:*sameAs*" predicate (Lines 4 and 5). Note, the seed instances are not all coreferenced instances, but the instances in the RI of the given query and determined by Alg. 1. Then, we extract the new equivalent URIs (Line 7), merge the equivalence classes of the seed URIs and the new extracted URIs (Line 8), and collect the new URIs (Line 9). This process is iteratively repeated by using any new URIs discovered as seeds (Lines 10-11). Since there are a finite small number of seed instances as input, and the process will only continue as long as new URIs are discovered, the algorithm can quickly and eventually terminate.

Algorithm 4 Fix point computation for instance coreference

function computeSameAs(*insts*, *EKB*) **returns** a list of instances
1: **inputs**: *insts*, a list of seed URIs
2: **Let** $inslist = insts$, *oldinsts* = *insts*
3: **for each** uri ∈ insts **do**
4: **Let** *bquery* = *uri* + "AND" + "*owl:sameAs*"
5: **Let** *srcslist* = askIndexer(*bquery*)
6: **for each** $s \in srcslist$ **do**
7: Let *sameAsPairs* = {t | t = $< x, owl{:}\,sameAs, y > \in s, x = uri \vee$ y = uri}
8: updateEquivalenceKB(*uri*, *sameAsPairs*, *EKB*)
9: $inslist = inslist \cup$ all instances URIs from sameAsPairs
10: **Let** *newinsts* = *inslist* – *oldinsts*
11: $inslist = inslist \cup$ ComputeSameAs($newinsts, EKB$)
12: **return** *inslist*

Fig. 3. Pseudo code of handling instance coreference

4 Evaluation

To evaluate our algorithms, we have conducted two groups of experiments based on a synthetic data set and a real world data set respectively. The first group measures the cycle handling performance of our algorithm. The second group tests the scalability and practicality of our algorithm using a subset of the real world Billion Triple Challenge (BTC) data set. For both groups, 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 eight triples, have at most seven 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 6 GB memory running UNIX. Our indexer component uses a term index [5] implemented with Lucene. Our reasoner is KAON2.

4.1 Cyclic Axiom Evaluation Using a Synthetic Data Set

In this group of experiments, we conducted two separate experiments. The first aims to compare the tree-structure algorithm with the cycle handling algorithm without equality reasoning (tree-structure(cycle w/o sameAs)) to the tree-structure algorithm without the cycle handling (tree-structure(non-cycle)) and the non-structure algorithm using a synthetic data generator that is designed to approximate realistic conditions. The second aims to compare the tree-structure algorithm with the cycle handling including the equality reasoning (tree-structure(cycle)) to the tree-structure algorithm without the cycle handling (tree-structure(non-cycle)) and the non-structure algorithm. For both experiments, 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 sources, and a diameter of 2, meaning

that the longest sequence of mapping ontologies between any two domain ontologies was 2. 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.

Cyclic Axioms Without Equality Reasoning In this experiment, we issued 120 random queries to our synthetic data set to measure our cycle handling algorithm with the increasing cycle complexity, which is related to two factors: the average number of cycles per query and the average length per cycle. In addition, since the cycle complexity increases with the number of unconstrained QTPs, 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, we group our 120 test queries by the number of unconstrained QTPs (from 0 to 5). The reason for selecting 5 as our maximum number of unconstrained QTPs is that the non-structure algorithm can only effectively scale to queries with at most 5 unconstrained QTPs [5]. In the metrics, we computed the average query response time and the cycle complexity. The experimental results are shown in Fig. 4.

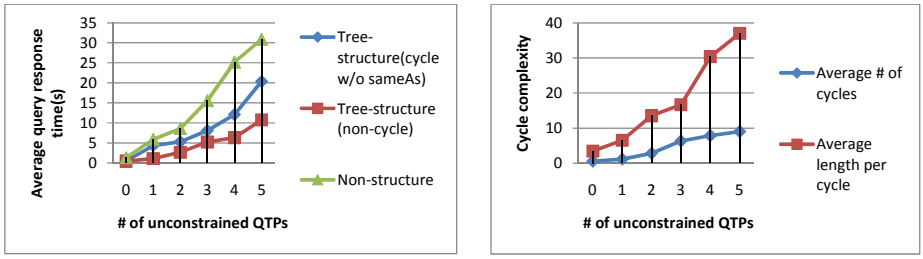

Fig. 4. Cyclic axiom handling algorithm w/o *owl*:*sameAs* experimental results. Average query response time (a) and cyclic axiom complexity (b) as the number of unconstrained QTPs varies.

Fig. 4 (a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs with cycle complexity increasing. From this result, we can see that the tree-structure (cycle w/o sameAs) algorithm is 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 tree-structure(cycle w/o sameAs) optimization algorithm to use constraints to enhance the selectivity of goals. Due to the additional cycle handling, the tree-structure (cycle w/o sameAs) algorithm is slower than the tree-structure algorithm (non-cycle), while the former can return more answers.

Fig. 4 (b) shows how the cyclic axiom complexity changes with the increasing number of unconstrained QTPs. As shown in this figure, our most complex test

queries have 5 unconstrained QTPs, 10 cyclic axioms per query and 35.5 nodes per cyclic axiom. To the best of our knowledge, this complexity is significantly greater than most queries issued to the semantic web. Therefore, we can conclude that our cyclic axiom handling algorithm can effectively scale to the real world.

Full Cyclic Axiom Evaluation In this experiment, we introduce the *owl*:*sameAs* triples in our synthetic data set based on the *owl*:*sameAs* statistics of the Billion Triple Challenge 2010 data set [7]. The ratio of sources containing *owl*:*sameAs* is 27.1%. The number of *owl*:*sameAs* triples is 2,765 of 45,673 total triples in 8000 sources. All *owl*:*sameAs* triples are categorized into 571 equivalence classes. The largest equivalence class contains 10 instances and the average equivalence class is 3.7. Like the last experiment, we still issued 120 random queries to our synthetic data set and group them by the number of unconstrained QTPs (from 0 to 5). In the metrics, we computed the average query response time and the query completeness. The experimental results are shown in Fig. 5.

Fig. 5 (a) shows how each algorithm's average query response time is affected by increasing the number of unconstrained QTPs with the increasing number of unconstrained QTPs. From this result, we can see that the tree-structure (cycle) algorithm is 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 tree-structure(cycle) optimization algorithm to use constraints to enhance the selectivity of goals. Due to the additional cycle and *owl*:*sameAs* handling, the tree-structure (cycle) algorithm is slower than the tree-structure algorithm (non-cycle).

Fig. 5 (b) shows the comparison of the completeness of the tree-structure (cycle) algorithm, the tree-structure (non-cycle) algorithm and the non-structure algorithm. Because the non-structure algorithm is complete [6], we take its results as ground truth. The percentage numbers in the graph are the completeness of the tree-structure (non-cycle) algorithm at each point. From this result, we can see that our tree-structure (cycle) algorithm is more complete than the tree-structure (non-cycle) algorithm. Furthermore, it returns the same number of answers as the non-structure algorithm, but has better query response time than the non-structure algorithm (as shown in Fig. 5 (a)).

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 including *owl*:*sameAs*. 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 documents. The size of documents varies from roughly 5 to 50 triples each. In order to integrate these heterogeneous documents, we manually created some mapping ontologies, primarily using *rdfs:subClassOf* and *rdfs:subPropertyOf* axioms (these schemas do not have any meaningful alignments that are more complex). In this experiment, our cyclic axioms are mainly

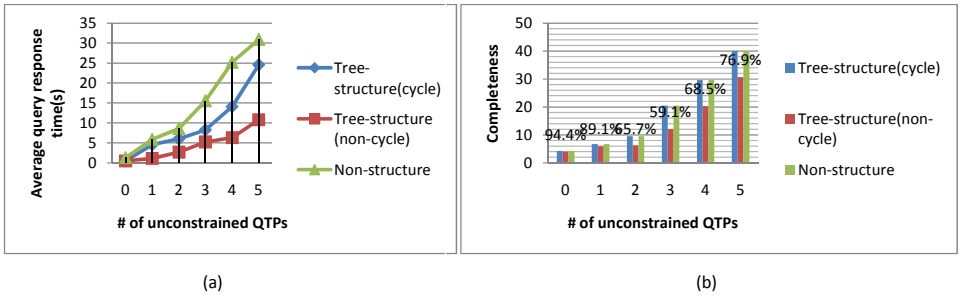

Fig. 5. Full cyclic axiom handling algorithm experimental results. Average query response time (a) and query completeness (b) as the number of unconstrained QTPs varies.

from mapping ontologies and *owl:sameAs* statements. The latter creates the most cycles. 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 Lucene configurations are 1500MB for RAMBufferSize and 1000 for MergeFactor, which are the best tradeoff between index building and searching for our experiment.

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.rkb-explorer.com/	akt	8,026,878	10,153,039
Total		21,008,285	73,889,151

Table 1. Data sources selected from the BTC 2009 dataset.

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. Even though some reasoners can load this amount of data as long as the system has 3GB of memory, load times are typically in the 7 hours range, which is clearly unsuitable for real-time queries. However, the tree-structure algorithms (cycle and non-cycle) can solve this problem 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 family algorithms.

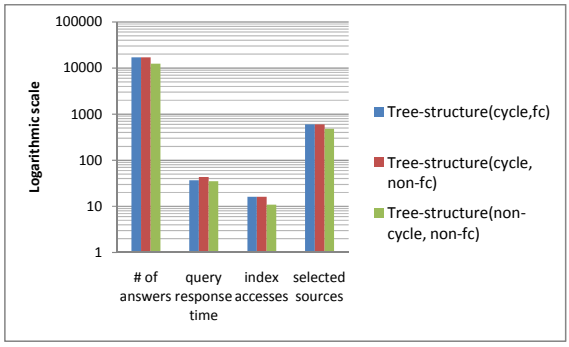

Fig. 6. BTC data set experimental results.

We executed 150 synthetic queries with at most 10 QTPs. In the metrics, we computed the average number of answers, average query response time, average number of selected sources and average index accesses of three algorithms: the tree-structure (cycle, fc), the tree-structure (cycle, non-fc) and the tree-structure(non-cycle, non-fc) algorithm. Here, "fc" stands for front-coding, which is an optimization technique we applied in order to improve the query response time of our algorithms. This is because of the fact that many URIs in the BTC data set have the same server name, and within each such set, there are many with the same namespace. The "fc" technique replaces each common server name with a number. As a result, our boolean query lengths are greatly compressed. The results are shown in Fig. 6 using a logarithmic scale. According to the results, we can see that the tree-structure (cycle, fc) and the tree-structure (cycle, non-fc) algorithms returned 36.8% more answers than the tree-structure (non-cycle, non-fc) algorithm even though they have small increases in the other three metrics because of the additional cycle processing. However, with the front-coding optimization, the query response time of the tree-structure (cycle, fc) algorithm has gained around 20% improvement over the tree-structure (cycle, non-fc) algorithm and is only around 5% more than the tree-structure (non-cycle, non-fc) algorithm.

5 Conclusions and Future Work

In this paper, we proposed a stack-based fix point computation algorithm to dynamically handle cyclic axioms including instance coreference for query answering over large distributed KBs. Using this algorithm, our system can deal with cyclic axioms on the fly and scale to queries with 8 QTPs (5 unconstrained QTPs), 10 cyclic axioms per query and 35.5 nodes per cyclic axiom on average. Meanwhile, it can return the same number of answers as the complete non-structure algorithm. In addition, we have also shown that our algorithm scales well on a real world data set, allowing randomly generated queries against 20 million heterogeneous data sources to complete in 30 seconds.

Despite showing initial promise, there is still significant room for improvement. First, the algorithm only focuses on conjunctive queries in SPARQL without FILTER and 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. In the future, we will explore how to extend our algorithms to support richer SPARQL queries and more expressive ontologies such as OWL 2, and also consider how to theoretically prove the correctness of our approach. We believe that this paper provides a major step towards a pragmatic solution for dynamic cyclic axiom handling in querying a large, distributed, and ever changing Semantic Web.

References

1. F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1986.
2. L. Ding, J. Shinavier, Z. Shangguan, and D. L. McGuinness. Sameas networks and beyond: Analyzing deployment status and implications of OWL: sameas in linked data. In *International Semantic Web Conference*, pages 145–160, 2010.
3. A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.
4. J. S. C. Lam, D. H. Sleeman, J. Z. Pan, and W. W. Vasconcelos. A fine-grained approach to resolving unsatisfiable ontologies. *J. Data Semantics*, 10:62–95, 2008.
5. Y. Li and J. Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *International Semantic Web Conference*, pages 502–517, 2010.
6. 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.
7. Y. Li, Y. Yu, and J. Heflin. A multi-ontology synthetic benchmark for the semantic web. In *In Proc. of the 1st International Workshop on Evaluation of Semantic Technologies*, 2010.
8. J. Mei, L. Ma, and Y. Pan. Ontology query answering on databases. In *International Semantic Web Conference*, pages 445–458, 2006.
9. Z. Pan, Y. Li, and J. Heflin. A semantic web knowledge base system that supports large scale data integration. In *The Workshop on Scalable Semantic Web Knowledge Base Systems, ISWC*, 2009.
10. 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.
11. A. Qasem, D. A. Dimitrov, and J. Heflin. Towards scalable information integration with instance coreferences, 2009.
12. J. Urbani, S. Kotoulas, J. Maassen, N. Drost, F. Seinstra, F. V. Harmelen, and H. Bal. WebPIE: A web-scale parallel inference engine. In *In: Third IEEE International Scalable Computing Challenge (SCALE2010), held in conjunction with the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2010.
13. T. D. Wang, B. Parsia, and J. A. Hendler. A survey of the web ontology landscape. In *International Semantic Web Conference*, pages 682–694, 2006.