Abstract: In this paper, we propose an intelligent distributed query processing method considering the characteristics of a distributed ontology environment. We suggest more general models of the distributed ontology query and the semantic mapping among distributed ontologies compared with the previous works. Our approach rewrites a distributed ontology query into multiple distributed ontology queries using the semantic mapping, and we can obtain the integrated answer through the execution of these queries. Furthermore, we propose a distributed ontology query processing algorithm with several query optimization techniques: pruning rules to remove unnecessary queries, a cost model considering site load balancing and caching, and a heuristic strategy for scheduling plans to be executed at a local site. Finally, experimental results show that our optimization techniques are effective to reduce the response time.
An Intelligent Query Processing for Distributed Ontologies

Jihyun Lee¹, Jeong-Hoon Park¹, Myung-Jae Park¹, Chin-Wan Chung¹, and Jun-Ki Min²

¹Division of Computer Science
Department of Electronic Engineering and Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Republic of Korea, 305-701
{hyunlee, jhpark, jpark, chungcw}@islab.kaist.ac.kr

²School of Internet-Media Engineering
Korea University of Technology and Education
Byeongcheon-Myeon, Cheonan, Chungnam, Republic of Korea, 330-708
jkmsn@kut.ac.kr

Abstract

In this paper, we propose an intelligent distributed query processing method considering the characteristics of a distributed ontology environment. We suggest more general models of the distributed ontology query and the semantic mapping among distributed ontologies compared with the previous works. Our approach rewrites a distributed ontology query into multiple distributed ontology queries using the semantic mapping, and we can obtain the integrated answer through the execution of these queries. Furthermore, we propose a distributed ontology query processing algorithm with several query optimization techniques: pruning rules to remove unnecessary queries, a cost model considering site load balancing and caching, and a heuristic strategy for scheduling plans to be executed at a local site. Finally, experimental results show that our optimization techniques are effective to reduce the response time.

1. Introduction

In the Semantic Web, the definitions of resources and the relationship between resources are described by an ontology in order to automatically interpret the resources and retrieve useful information. The resources in the Web are independently generated in many locations. Thus, even if the ontologies describe resources in the same (similar) domain, they can use different representations (i.e., language and schema). Also, the ontologies are managed by various local ontology management systems which have different capabilities and strategies for storing and query processing. Under these environment, some Web applications want to access the ontologies without regard to the heterogeneity and the dispersion of the ontologies and the local systems. In order to support such
a request, an efficient query processing over the distributed ontologies is essential. Of course, existing distributed query processing techniques can be applied to query the distributed ontologies. However, they confront the limitations of the efficiency and the functionality since some important characteristics of a distributed ontology environment are not considered.

Figure 1 shows an example of the distributed ontology environment. There are three kinds of ontologies, UNIV, COLLEGE, and PUB which are managed in three different sites and two types of local systems (i.e., LS$_1$, LS$_2$). UNIV and COLLEGE describe the information of the university and the college, respectively, and PUB describes the publication information. For the simplicity, we describe only the schema and omit the instance part. These ontologies are independently generated but related to each other even if they have different schemas. For example, let us suppose the following conditions: first, the concept of Professor in UNIV is defined as the concept of Lecturer in COLLEGE. Second, the information of the authors in PUB can be found in UNIV and COLLEGE. In this distributed ontology environment, consider the following example queries:

**Example 1.** $Q_1$: *Find professors who teach ‘Algorithm’.*
**Example 2.** $Q_2$: *Find authors who wrote publications about ‘Semantic Web’ and also retrieve the name and the email addresses of the authors.*

In order to find the answer of query $Q_1$, we should retrieve professors and lecturers who teach ‘Algorithm’ from UNIV and COLLEGE, respectively. For query $Q_2$, UNIV and COLLEGE should be searched along with PUB to find the personal information of the authors who wrote papers about ‘Semantic Web’. For such a query, in order to efficiently find the answer dispersed in several ontologies and local sites, a distributed query processing method considering the heterogeneity of the ontologies is required.

The use of the semantic mapping is a representative approach to deal with the heterogeneity among different ontologies [1, 2, 3, 4]. In [1, 4], the semantic mapping is the semantic relationship (i.e., subsumption or equivalence) between
concepts (i.e., classes or properties) in two different ontologies and it has been extended to that between views (i.e., queries)[2, 3]. However, the previous works do not support more general semantic mapping and distributed query covering more than two ontologies. Besides, most of them have focused on only the rewriting of the query using the semantic mapping, and do not make an issue of the efficient distributed query evaluation (i.e., query rewriting, scheduling, and execution).

In this paper, we resolve issues of the distributed query processing over multiple heterogeneous ontologies. We extend the models of the distributed query and the semantic mapping to support more general distributed ontology query answering compared with previous works. Furthermore, we present a distributed ontology query processing algorithm with several query optimization techniques considering the characteristics of the distributed ontology environment.

The contributions of the paper are as follows:

**Extended models of the distributed ontology query and the semantic mapping:** We present a general distributed ontology query model to cover multiple different ontologies. We also present a general semantic mapping model in which more than two ontologies can be associated. The extension of query and semantic mapping models makes it possible to include relevant data which could not be accessed before in the query result. Also, our approach logically integrates independently grown distributed ontologies through the query rewriting based on the semantic mapping. As a result, we can efficiently extract an integrated answer of a distributed query over different ontologies.

**Optimization techniques for an efficient query processing on the distributed ontologies:** Multiple distributed queries are generated from an original distributed query to obtain results from dispersed ontologies. In order to remove unnecessary operations and to increase the parallelism among executions of the multiple queries, we suggest several optimization techniques. First, we present pruning rules to remove invalid and redundant queries. Second, we suggest a heuristic strategy for scheduling plans to be executed at a local site. Third, we propose a cost model considering site load balancing and caching for processing multiple distributed queries.

The remainder of the paper is organized as follows. In Section 2, we review related work. In Section 3, we present a distributed ontology query model and a semantic mapping model. Section 4 describes a distributed query processing technique with several query optimization techniques over distributed ontologies. Section 5 contains the results of experiments. Finally, in Section 6, we conclude this paper.

2. Related Work

Recently, the research on a query processing over distributed ontologies has been performed. [5] suggests a global data summary for locating data matching
query answers in different sources and the query optimization. However, [5] assumes that all distributed ontologies can be accessed in a uniform way like a global schema. In other words, the heterogeneity of schemas of the distributed ontologies is not considered. Besides, many tasks are concentrated on the mediator. As well as query scheduling, the merge (i.e., join) of all local query results is also executed in the mediator. Thus, when the mediator receives requests for many queries at the same time, the bottleneck on the mediator is inevitable.

The most of research on the query answering over distributed ontologies are based on the P2P architecture. Edutella[6] uses an unstructured P2P network which has no method to route a query to the relevant ontologies. Instead, the query is broadcasted in the entire network. Thus, a huge amount of unnecessary network traffic incurs. As a successor of Edutella, to provide better scalability, [7] presents a schema-based query routing strategy in a hierarchical topology using the super-peer concept. [7] also suggests a rule-based mediation between two different schemas in order to collect results from many peers using heterogeneous schemas. SomeRDFs[8] supports the semantic mapping between two atomic concepts and between the domain (or range) of a property and a class. Piazza[9] proposes a language (heavily relies on XQuery/XPath) to describe the semantic mapping between two different ontologies. In these works, for distributed query answering, a peer reformulates a query by using the semantic mapping and forwards the reformulated query to another peer related by the semantic mapping.

DRAGO[4] focuses on a distributed reasoning based on the P2P-like architecture. In DRAGO, every peer maintains a set of ontologies and the semantic mapping between its local ontologies and remote ontologies located in other peers. A reasoning service is performed by a local reasoner for the locally registered ontologies and the reasoning is propagated to the other peers when the local ontologies are semantically connected to the other remote ontologies. The semantic mapping supported in DRAGO is only the subsumption relationship between two atomic concepts. Besides, it does not support the Abox reasoning (i.e., instance retrieval).

KAONP2P[10] also suggests the P2P-like architecture for query answering over distributed ontologies. KAONP2P supports more extended semantic mapping which describes the correspondence between views of two different ontologies, where each view is represented by a conjunctive query. For the distributed query answering, it generates a virtual ontology including a target ontology to which the query is issued and the semantic mapping between the target and the other ontologies. Then, the query evaluation is performed against the virtual ontology.

The previous studies premise that a user makes a query based on a target ontology. Thus, they do not consider queries associated with multiple ontologies such as query $Q_2$ in Example 2. In addition, the recent several studies based on the P2P architecture consider the semantic mapping between two different ontologies. However, the scope of the semantic mapping is still not sufficient for the distributed ontology environment. They do not consider the semantic mapping associated with more than two ontologies like the following example:
This semantic mapping means if \( x \) wrote \( y \) about \( z \), \( x \) researches on \( z \), and the properties authorOf, isAbout and researchOn are defined in COLLEGE, PUB, and UNIV ontology, respectively. Besides, most of previous works have concentrated on only the query reformulation, but not on the efficient evaluation of distributed ontology queries.

3. Preliminary

The ontology describes the definitions of resources and the semantic relationships among the resources. An ontology consists of a schema and instances. The schema defines concepts (i.e., class and property) and relationships between the concepts. In the instance part, the type (i.e., class) of a resource and the relationship (i.e., properties) between resources are declared according to the schema. The ontology is expressed in triples describing the relationships between concepts, between instances, and between a class and an instance.

**Definition 1. (Triplet)** Triple \( t \) is \((p, s, o)\) where \( p \) is a property, \( s \) is the subject of \( p \) and \( o \) is the object of \( p \). \( p \) has the domain \( D(p) \) and the range \( R(p) \). \( s \) is a class or an instance of class \( D(p) \) and \( o \) is a class or an instance of class \( R(p) \). For a class \( c \) in a schema, \([c]\) denotes the set of instances belonging to \( c \).

For example, UNIV in Figure 1 includes the following triples: \((\text{subClassOf}, \text{Professor}, \text{Person})\) and \((\text{write}, \text{Person}, \text{Paper})\). Also, the instance part of UNIV can contain \((\text{type}, \text{prof}1, \text{Professor})\) and \((\text{write}, \text{prof}1, \text{pub}1)\).

**Definition 2. (Ontology Query)** Given ontology \( K \), query \( Q \) is a triple pattern like \( qt_1 \land ... \land qt_m \) where \( qt_i = (p_i, s_i, o_i) \) is a query triple over \( K \). The subject and the object of the query triple are a variable or a literal. \( qt_i \) contains at least one variable. And, for \( qt_i \), there is \( qt_j \) such that they share a variable.

For \( qt_i = (p_i, s_i, o_i) \) over ontology \( K \), \([qt_i]\) denotes the set of matches for \( qt_i \) in \( K \). \([p_i(s_i)]\) and \([p_i(o_i)]\) denote the set of distinct subjects and the set of distinct objects in \([qt_i]\), respectively. If \( p_i \) is type, \([qt_i] = [o_i] \) where the object \( o_i \) is a class declared in \( K \).

The query answering for \( Q \) over ontology \( K \) is to find matches for the triple pattern of \( Q \) from \( K \). Given ontology \( \text{UNIV}, (\text{type}, ?x, \text{Professor}) \land (\text{write}, ?x, ?p) \land (\text{isAbout}, ?p, \text{Semantic Web}) \) finds professors who wrote papers about ‘Semantic Web’ from \( \text{UNIV} \).

We assume that the schemas of multiple ontologies are provided to a user, and the user can generate a distributed query covering the multiple ontologies.

**Definition 3. (Distributed Ontology Query)** Given a set of ontologies \( S \), distributed ontology query \( DQ_S \) is a conjunctive query over ontologies in \( S \). \( DQ_S = Q_1 \land ... \land Q_n \) where \( Q_i \) is a local ontology query for an ontology in \( S \). And, for \( Q_i \), there is \( Q_j \) such that they share a variable.
Table 1: An example of Bridge Rules

<table>
<thead>
<tr>
<th>ID</th>
<th>BRIDGE RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1$</td>
<td>$C:(\text{type}, \ ?x, \ \text{Lecturer}) \rightarrow U:(\text{type}, \ ?x, \ \text{Professor})$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$U:(\text{type}, \ ?x, \ \text{Person}) \land U:(\text{write}, \ ?x, \ ?y) \rightarrow P:(\text{type}, \ ?x, \ \text{Author})$</td>
</tr>
<tr>
<td>$B_3$</td>
<td>$C:(\text{authorOf}, \ ?x, \ ?y) \land P:(\text{isAbout}, \ ?y, \ ?z) \rightarrow U:(\text{researchOn}, \ ?x, \ ?z)$</td>
</tr>
</tbody>
</table>

Query triples in a distributed query match triple instances of various ontologies. Thus, in order to distinguish the query triples according to the ontology, the identifier of the corresponding ontology is assigned to each query triple. A query triple $q_t$ over ontology $O_j$ is represented by $O_j: (p_i, s_i, o_i)$.

In the distributed ontology environment, there may be the semantic mapping among some views of different ontologies relevant to each other. The semantic mapping is expressed via the bridge rules.

**Definition 4. (Bridge Rule)** A bridge rule is represented in an assertion $DQ_S \rightarrow DQ_T$, where $DQ_X$ is a distributed ontology query. The assertion means that the answer of the left side query is contained by the answer of the right side query.

Table 1 shows an example of bridge rules over the distributed ontologies in Figure 1. We support the bridge rules related to more than two ontologies such as bridge rule $B_3$.

4. Distributed Ontology Query Processing

The schemas of multiple ontologies are provided to the user who wants to query over the ontologies, and the user makes distributed ontology queries covering the ontologies. We assume that the schemas are provided in the same representation format and the user is capable of understanding the content of each schema.

A distributed ontology query can be rewritten to multiple ontology queries according to bridge rules. The answer of the original user query will be covered by the distributed answers of the multiple queries. The queries search other ontologies which the user does not specify in the original query, even does not know about them. In conclusion, our approach does not physically integrate all local ontologies into a global ontology, but can find the integrated answer of the user query through the query rewriting using bridge rules and the evaluation of queries.

The optimization cost for processing a distributed query can be expensive. Especially, in order to find the optimal plan for the distributed query which can be processed in parallel at multiple sites, many parallel execution plans should be examined due to the various sources of parallelism. Therefore, a heuristic to reduce the search space and to provide an efficient query plan is
Input:  $DQ$(query), $BR$(bridge rules)
Output: $EQ$(a set of final queries)

begin
1. $EQ := \{DQ\}$; $tmpEQ := \{DQ\}$;
2. for each query $q_i$ in $tmpEQ$
3. $MR := \{r_j \in BR \mid$ bridge rule $r_j$ whose right side matches a part of $q_i\}$;
4. for each rule $r_j$ in $MR$
5. $JV := \{\text{variables of matched triples in } q_i \cap \text{variables of unmatched triples in } q_i\}$;
   /* $JV$ is the set of join variables in $q_i$ */
6. generate a new query, $EQ_n$, by replacing the matched triples in $q_i$ with the left side query of $r_j$ preserving the subjects and the objects;
7. if $JV$ is a subset of the set of variables in $EQ_n$
8. add $EQ_n$ to $tmpEQ$;
9. add $EQ_n$ to $EQ_n$;
10. end if
11. end for
12. remove $q_i$ from $tmpEQ$;
13. if the size of $tmpEQ = 0$ then return $EQ$;
14. end for
end

Figure 2: Query rewriting algorithm

required. In addition, the reformulated queries as well as the original query are also distributed ontology queries. If these queries are scheduled independently, the workload can be concentrated on a few local sites. Thus, a query plan which distributes the workload evenly among all local sites is required.

For the query rewriting and the query optimization, we assume that every local site in the distributed environment registers its ontologies in a meta-data registry. The registry maintains the schemas and the statistics of all ontologies for query rewriting and scheduling in the distributed environment. In addition, the bridge rules related to the ontologies are also stored in the registry. All local sites can remotely access the registry.

4.1. Query Rewriting

In the query rewriting phase, multiple distributed ontology queries are generated from a distributed ontology query according to bridge rules. Figure 2 describes the query rewriting algorithm. Given a query $q_i$, we find the set of all bridge rules $MR$ such that the right side of a rule in $MR$ matches a part of $q_i$ where ‘match’ means that each triple in one side has the corresponding triple with the same property on the other side and the join relationships among triples of both sides are the same (line 3). For each rule $r_j$ in $MR$, a new query $EQ_n$ is generated by replacing the part of $q_i$ matching the right side of $r_j$ with the left side of $r_j$ preserving the subjects and objects of the replaced part. Then, if the newly generated query does not lose the join relationships among triples in
Table 2: Reformulated queries

<table>
<thead>
<tr>
<th>ID</th>
<th>EQ</th>
<th>BR</th>
<th>FROM</th>
</tr>
</thead>
</table>
| DQ | U:
(type, ?x, Professor) ∧ U:
(researchOn, ?x, ‘semantic web’)                                                                                                                                  |     | DQ     |
| EQ1 | C:
(type, ?x, Lecturer) ∧ U:
(researchOn, ?x, ‘semantic web’)                                                                                                                                    | B1  | EQ1    |
| EQ2 | C:
(type, ?x, Lecturer) ∧ C:
(authorOf, ?x, ?y) ∧ P:
(isAbout, ?y, ‘semantic web’)                                                                                                                                    | B3  | EQ1    |

$q_i$, $EQ_n$ is accepted (lines 4-12). This process is applied to all newly generated queries as well as the original query $DQ$. If there is no more additional query, the process is terminated (line 13).

Table 2 shows a part of the queries derived from $DQ$ by using the bridge rules in Table 1. Table 2 also presents the applied bridge rule (BR) and the previous query from which the query is derived (FROM). The query $EQ_1$ is generated from $DQ$ by using bridge rule $B_1$. $U:
(type, ?x, Professor)$ in $DQ$ is replaced by the left side of $B_1$, $C:
(type, ?x, Lecturer)$. $EQ_2$ is generated from $EQ_1$ through $B_3$.

Among the queries generated in the rewriting phase, there can be some invalid (i.e., there is no answer) or redundant (i.e., the answer is contained in the answer of another query) queries. If we prune such queries, we can effectively reduce the workload. As a result, we suggest two types of pruning rules to eliminate the unnecessary queries before the query scheduling.

Consider two query triples sharing a variable. If there is no overlap between the sets of instances matched to the variable in each ontology, we can conclude that the result of the query does not exist. Thus, such a condition can be used to prune the invalid queries. According to this condition, we define the following pruning rule:

**Rule 1.** Given a query $EQ$ containing $q_i = O_i:(p_x, s_x, o_x)$ and $q_j = O_j:(p_y, s_y, o_y)$, if $s_x = s_y$ and $[D(p_x)] \cap [D(p_y)] = \emptyset$ or if $s_x = o_y$ and $[D(p_x)] \cap [R(p_y)] = \emptyset$ (if $s_y = o_x$ is symmetric) or if $o_x = o_y$ and $[R(p_x)] \cap [R(p_y)] = \emptyset$ then $EQ$ has no result.

There can be superfluous queries whose answers are contained in the answer of another query. Consider the distributed ontologies in Figure 1 and the following queries:

$EQ_1: U:
(type, ?x, Professor) ∧ U:
(write, ?x, ?y)$

$EQ_2: C:
(type, ?x, Lecturer) ∧ U:
(write, ?x, ?y)$

Assume that there is a student who has some publications in $UNIV(U)$ and
Figure 3: The diagram representing the overlap of query answers: U and C denote UNIV ontology and COLLEGE ontology, respectively.

the student is also a lecturer in COLLEGE(C). In the diagram of Figure 3(a), the student is included in the a area. However, the striped area which is the answer of EQ1 can not cover the a area. Therefore, if we want to see all his/her publications, both EQ1 and EQ2 are needed. In contrast, consider the following queries:

\[EQ_3: U: (type, ?x, Person) \land U: (write, ?x, ?y)\]
\[EQ_4: C: (type, ?x, Member) \land U: (write, ?x, ?y)\]

Since the domain of the property write is U:Person, all instances matched to U: (write, ?x, ?y) (i.e., the gray area in Figure 3(b)) are retrieved by EQ3. The answer of EQ4 (i.e., the b area in Figure 3(b)) is included in the answer of EQ3. Therefore, EQ4 is a redundant query of EQ3. Consequently, we can define a new pruning rule as below:

**Rule 2.** Given a query EQ containing \(qt_i = O_i: (type, s_x, o_x)\) and \(qt_j = O_j: (p_y, s_y, o_y)\), if there is a query EQ′ that contains \(qt'_i = O_j: (type, s_x, o'_x)\) instead of \(qt_i\), where \(s_x = s_y\) and \(D(p_y) \subseteq o'_x\) or \(s_x = o_y\) and \(R(p_y) \subseteq s_x\), and other query triples in EQ′ are the same as those in EQ, then EQ is a redundant query since the result of EQ′ contains the result of EQ.

There can be an indirect semantic mapping between two concepts in different ontologies. Given the bridge rules X:A \(\rightarrow\) Y:B and Y:C \(\rightarrow\) Z:D, if the local subsumption relationship Y:B \(\rightarrow\) Y:C is declared, a semantic mapping X:A \(\rightarrow\) Z:D is implied. In order to derive such an indirect semantic mapping, our approach stores the local subsumption relationships between concepts as bridge rules. We assume that the local reasoning to find all subsumption relationships between concepts in an ontology is performed by a local ontology management system.

### 4.2 Query Optimization Technique

We assume that the ontologies relevant to a query are distributed at several local sites and all the local sites are capable of query processing (i.e., join).
Query $D_Q = Q_1 \land Q_2 \land Q_3$

$Q_1 = q_{t1}$, $Q_2 = q_{t2}$, $Q_3 = q_{t3} \land q_{t4}$

Assumptions
- User submits $D_Q$ to $S_1$
- $Q_2$ is a query over ontology $O_1$
- $Q_1$, $Q_3$ are queries over ontology $O_2$
- Ontology $O_i$ is stored at local site $S_i$
- $Q_i$ and $Q_{i+1}$ share a variable $p$

Thus, the local site where a user submits a query and all local sites where the ontologies relevant to the query are stored can be potentially involved in the processing of the query. We call these local sites ‘interesting sites’.

A query plan for processing a distributed ontology query has a binary tree structure. It describes a join order among the results of the local ontology queries, and locations where the joins should be performed. The result of the root plan is the final result of the distributed ontology query. Each leaf plan corresponds to a local ontology query. Figure 4 shows an example of the distributed ontology query $D_Q$ and its query plan. Table 3 presents notations used to describe a plan. The answer of query $D_Q$ is dispersed at local sites $S_1$ and $S_2$, and the user issues $D_Q$ to $S_1$. Thus, the interesting sites are $\{S_1, S_2\}$. $p_2$, $p_4$ and $p_5$ are plans for the local ontology queries $Q_1$, $Q_2$, and $Q_3$, respectively.

The execution site of $p_5$ is $S_2$ (i.e., $p_5.site = S_2$) and the result of $p_5$ should be sent to the site where the upper plan (i.e., $p_3$) is executed (i.e., $p_5.des = S_1$). $p_2$ and $p_3$ are the left and right sub-plans of $p_1$. The result of the root plan $p_1$ is the final result of $D_Q$.

4.2.1. Cost Model

In a distributed ontology environment, each local site has the capability to process distributed queries. Each local ontology query is processed in the local ontology management system which performs the disk based query processing. On the other hand, the distributed query processing which needs to join the results of the local ontology queries is processed in memory.

The sub-queries (i.e., local ontology query and intermediate distributed query) of a distributed ontology query can be executed in parallel. Thus, as the cost function, we use the response time which is the amount of the time it takes for a user to receive the query result. To estimate the response time of the distributed query processing, the cost of data transmission and the cost of join should be considered.

For a join operation, we consider two kinds of methods: Hash Join and Nested Loop Join. The join cost according to the join method can be estimated as follows:
Table 3: Notations for information of plan $p$

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p.l$</td>
<td>the left sub-plan of $p$</td>
</tr>
<tr>
<td>$p.r$</td>
<td>the right sub-plan of $p$</td>
</tr>
<tr>
<td>$p.site$</td>
<td>the site of the root of $p$</td>
</tr>
<tr>
<td>$p.des$</td>
<td>the site where the result of $p$</td>
</tr>
<tr>
<td></td>
<td>will be sent</td>
</tr>
</tbody>
</table>

**Join Cost.** The join cost of a plan $p$,

$$JC(p) = \begin{cases} 
HJC(p.l, p.r) = I \times |p.l| + R \times |p.r| & \text{for Hash Join} \\
NJC(p.l, p.r) = |p.l| \times |p.r| \times C & \text{for Nested Loop Join} 
\end{cases}$$

- $I$ : the cost of inserting an item in the hash table
- $R$ : the cost of retrieving a bucket from the hash table
- $|x|$ : the cardinality of the result of $x$
- $C$ : the comparison cost of two items

For each join operation, our optimizer chooses a cheaper method between these two join methods.

During the distributed query processing, data (i.e., query plans and query results) transmissions among local sites occur a lot. We only consider the transmission of the query result since the size of query plans is relatively very small. Thus, the transmission cost of the query result is estimated as follows:

**Transmission Cost.** The result transmission cost of a plan $p$,

$$TC(p) = \begin{cases} 
d \times \text{leng}(p) \times |p| & \text{if } p.des \neq p.site \\
0 & \text{if } p.des = p.site 
\end{cases}$$

- $\text{leng}(p)$ : the average length of tuples in the result of $p$
- $|p|$ : the cardinality of the result of $p$
- $d$ : the constant decided by the data transfer rate of the network (in inverse proportion to the data transfer)

Each leaf plan in a plan tree is carried out by the local ontology management system in its execution site. Thus, the cost of the leaf plan is dependent on the query optimization strategy of the local ontology management system. There are several approaches estimating the cost of the leaf plan: generic cost model approach, individual wrapper cost model approach, and learning curve approach[11]. The first approach uses a generic cost model with an adjustable parameter denoting the performance of each local ontology management system.
The second approach defines the separate cost model for each local system. For this approach, the query scheduling module should know the cost models of all the local systems. The last approach estimates the cost of the local query plan based on the statistics of the past plans. Our approach uses a generic cost model for estimating the cost for the local query plan since it is the simplest one and the cost models for all the local systems are generally not known. The cost of the plan \( p \) for a local ontology query \( LC(p) \) is estimated by the following generic cost model:

**Generic Cost.** The cost of a plan \( p \) for local ontology query,

\[
LC(p) = GC(p)
\]

\[
GC(p_j) = c \begin{cases} |p_j| & \text{if } p_j \text{ is a leaf plan in } p \\ JC(p_j) + GC(p_j,l) + GC(p_j,r) & \text{otherwise} \end{cases}
\]

- \( p_j \) : a sub-plan in \( p \)
- \( c \) : a constant denoting the performance of the local system

We use the same join cost model in both memory-based query processing and disk-based query processing. In order to reflect the difference between them in the cost estimation, we use different values of the parameters (e.g., the cost of inserting an item in the hash table, \( I \), and the cost of retrieving a bucket from the hash table for the hash join, \( R \)).

In the distributed query processing, the operands of a join operation can be generated from different sites. Thus, the transmission cost should be considered. However, each join operation is deferred until both operands (i.e., the results of left and right sub-plans) are prepared and other query plans to be executed before the plan at the same site finish their works. This is why the waiting time should be also considered. The cost of the distributed query plan is estimated as follows:

**Query Cost.** The cost for a distributed query plan \( p \),

\[
QC(p) = w + \begin{cases} LC(p) + TC(p) & \text{if } p \text{ is a leaf plan} \\ JC(p) + TC(p) & \text{otherwise} \end{cases}
\]

\[
w = \max\{QC(p.l), QC(p.r), QC(\text{LastPlan}(p.site))\}
\]

- \( \text{LastPlan}(p.site) \): the last plan which should be executed before \( p \) at \( p.site \).

The overall response time is the time taken to finish the processing for all distributed queries. The plans at a single site are executed sequentially, while the plans in different sites are executed in parallel. Consequently, the overall
Input: $DQ$(query), $BR$(bridge rules), $S$(sites)
Output: $PL$ (a set of plan lists for each site)

begin
1. generate multiple queries $EQ$ from $DQ$ according to $BR$; /*query rewriting*/
2. for each distributed ontology query $eq_i$ in $EQ$
3. generate local ontology queries, $LQ_i$, from $eq_i$;
4. for each local ontology query $lq_j$ in $LQ_i$
5. $p_j :=$ plan for join among query triples composing $lq_j$ /*by dynamic programming*/
6. add $p_j$ to $Ls$; /* $Ls$ is a set of local ontology query plans */
7. end for
8. $PL := PlanGeneration(Ls, S, PL)$;
9. end for
10. return $PL$;
end

Figure 5: Query scheduling algorithm

cost is decided by the cost of the query plan which is to be lastly finished among the plans processed at local sites.

Overall Cost. The overall cost of $EP$ which is a set of query plans for executing the distributed ontology queries,

$$OC(EP) = \max_{s_i \in S}\{QC(LastPlan(s_i))\}$$

- $S$ : the set of interesting sites
- $LastPlan(s_i)$ : the distributed ontology query plan which is lastly executed at site $s_i$.

4.2.2. Query Scheduling Algorithm

Figure 5 shows the query scheduling algorithm. Given a distributed query $DQ$, the bridge rules $BR$, and the interesting sites $S$, the algorithm returns plan lists to be executed in each interesting site. As mentioned in Section 4.1, multiple queries $EQ$ is generated from $DQ$ based on the bridge rules $BR$ (line 1). The multiple queries are sequentially scheduled. For each distributed query, a query plan is generated and decomposed according to the execution site. A decomposed query plan is added to the plan list of its operating site (lines 2-9). We purpose to make the maximum use of the processing capacity of each local ontology management system. Thus, at the beginning of generating a plan for each distributed query, the query is partitioned into local ontology queries (line 3). Each local ontology query is scheduled by the dynamic programming based scheduling strategy (line 5). Then, a join plan among the local ontology queries of an distributed query is generated by $PlanGeneration$ algorithm in Figure 6.
Input: \( Ls \) (a set of local ontology query plans), \( S \) (sites),
\( PL \) (a set of plan lists for each site in \( S \))
Output: \( PL \)

begin
1. enumerate query plans \( EP \) by composing query plans in \( Ls \) according to the original query structure and interesting sites \( S \);
2. \( \text{minCost} := \text{MAX}; /* Initialize minCost */ \)
3. \( \text{minPL} := \text{null}; \)
4. for each query plan \( ep_i \) in \( EP \)
5. \( \text{tmpPL} := PL; \)
6. for each sub-plan \( sp_j \) in \( ep_i \)
7. add \( sp_j \) to \( \text{tmpPL}_s; /* \( s = sp_j.site \) */ \)
8. end for
9. \( ep_i.cost := 0; \)
10. for each \( \text{tmpPL}_s \) in \( \text{tmpPL} \)
11. schedule sub-plans of \( ep_i \) in \( \text{tmpPL}_s \) by \( \text{DPFO} \);
12. \( \text{cost} := \text{the cost of the last plan in } \text{tmpPL}_s; \)
13. if \( \text{cost} > ep_i.cost \) then \( ep_i.cost := \text{cost}; \)
14. end for
15. if \( \text{minCost} > ep_i.cost \) then
16. \( \text{minCost} := ep_i.cost; \)
17. \( \text{minPL} := \text{tmpPL}; \)
18. end if
19. end for
20. \( PL := \text{minPL}; \)
21. return \( PL; \)
end

Figure 6: PlanGeneration algorithm

(line 8). The PlanGeneration algorithm finds an efficient plan to achieve the effective parallel execution among plans.

In order to an efficient distributed ontology query processing, our query scheduling algorithm uses the following query optimization techniques:

**Deeper Plan First Order (DPFO)** Note that plans at a local site are sequentially executed. The execution order among the plans at a local site has an influence on the total response time. Figure 7 shows the response time (i.e., cost) of two different execution orders for the same query plan. There are two kinds of possible execution orders among plans \( p_2 \) and \( p_4 \) in \( S_2 \). If \( p_2 \) is executed before \( p_4 \), the execution of \( p_3 \) is delayed for the execution time of \( p_2 \) since \( p_3 \) needs the result of \( p_4 \). Besides, the delay is propagated to the total cost. On the other hand, if \( p_4 \) is executed first, it is possible to execute \( p_2 \) and \( p_3 \) in parallel, and ultimately the total cost can be reduced such that \( C_d < C_r \). In conclusion, we decide the execution order among plans at a single site where the deeper plan in the plan tree is executed first. We call this ordering method ‘Deeper Plan First Order (DPFO)’. The order of execution among plans with the same
Load balancing for distributed queries For a single distributed ontology query, multiple distributed ontology queries can be generated and processed for the query result. Therefore, during the scheduling of a distributed ontology query, if we do not consider the workload of each local site where the operations of the other queries are also assigned, many operations can be concentrated on several local sites, even one site. In order to deal with this problem, at the beginning of the scheduling for each query, the scheduler refers to the cost of the plan which is lastly executed among the plans belonging to the previously scheduled query for each local site, and decides the operating site for each join so as to minimize the overall response time.

Caching In the query rewriting phase, multiple distributed ontology queries are generated from a user query in order to cover the dispersed answer of the query. There can be common sub-queries among them. It means that the common sub-queries are repeatedly performed during the entire query processing. The caching is an effective way to reduce the overhead due to the repeated execution of the same query. Thus, during the execution of a distributed query, the result of the sub-query which will be accessed again by subsequent other distributed queries is stored in the cache at local sites. In addition, the caching can be considered in the cost estimation for a query plan. If the result of the query is cached, the cost of the plan is estimated as:

\[ QC(p) = w + |p| + TC(p) \] if the result of \( p \) is cached at \( p.site \)

5. Experimental Analysis

The previous studies on the query processing over distributed ontologies have focused on only the query rewriting for the distributed query processing, but not on the efficiency of the query evaluation. Thus, in order to show the efficiency of our query processing technique, we implemented AIDOS (An Intelligent Distributed Ontology query processing), and empirically compared the performance of the distributed query processing of two versions of AIDOS: AIDOS-I and AIDOS-II. AIDOS-I prunes unnecessary queries before the query

Figure 7: Two different execution orders of plans for DQ in Figure 4. \( d \) denotes the depth of a plan in the plan tree.
Table 4: An experimental environment

<table>
<thead>
<tr>
<th>Site</th>
<th>System</th>
<th>CPU</th>
<th>Memory</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>ONTOMS</td>
<td>2.8 GHz</td>
<td>2 GB</td>
<td>UNIV(U)</td>
</tr>
<tr>
<td>Site 2</td>
<td>ONTOMS</td>
<td>2.4 GHz</td>
<td>1GB</td>
<td>COLLEGE(C)</td>
</tr>
<tr>
<td>Site 3</td>
<td>JENA</td>
<td>2.4 GHz</td>
<td>768MB</td>
<td>PUB(P)</td>
</tr>
</tbody>
</table>

Table 5: Bridge Rules

<table>
<thead>
<tr>
<th>ID</th>
<th>BRIDGE RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>C:(type,?x,Student)→U:(type,?x,Student)</td>
</tr>
<tr>
<td>B2</td>
<td>U:(type,?x,Person)∧U:(publicationAuthor,?y,?x)→P:(type,?x,Author)</td>
</tr>
<tr>
<td>B3</td>
<td>C:(type,?x,Member)∧U:(writtenBy,?y,?x)→P:(type,?x,Author)</td>
</tr>
<tr>
<td>B4</td>
<td>C:(name,?x,?y)→U:(name,?x,?y)</td>
</tr>
<tr>
<td>B5</td>
<td>C:(type,?x,Professor)→U:(type,?x,Professor)</td>
</tr>
<tr>
<td>B6</td>
<td>C:(type,?x,FullProfessor)→U:(type,?x,FullProfessor)</td>
</tr>
<tr>
<td>B7</td>
<td>U:(PublicationAuthor,?x,?y)→P:(writer,?x,?y)</td>
</tr>
<tr>
<td>B8</td>
<td>C:(writtenBy,?x,?y)→P:(writer,?x,?y)</td>
</tr>
<tr>
<td>B9</td>
<td>U:(type,?x,Publication)→P:(type,?x,Publication)</td>
</tr>
<tr>
<td>B10</td>
<td>C:(type,?x,Paper)→U:(type,?x,Professor)</td>
</tr>
<tr>
<td>B11</td>
<td>C:(teachesClass,?x,?y)→U:(teacherOf,?x,?y)</td>
</tr>
<tr>
<td>B12</td>
<td>C:(writtenBy,?x,?y)→U:(publicationAuthor,?x,?y)</td>
</tr>
<tr>
<td>B13</td>
<td>C:(writtenBy,?x,?y)∧P:(isAbout ?y ?t)→U:(researchOn,?x,?t)</td>
</tr>
</tbody>
</table>

5.1. Environment

We implemented AIDOS-I(II) using a relational database management system (RDBMS) and Java. The RDBMS is used for managing metadata information such as bridge rules and statistics. To construct the heterogeneous distributed experimental environment, ONTOMS[12] and JENA[13] were used as local ontology management systems. A detailed information on local systems is described in Table 4.

Scheduling, and uses DPFO heuristic in the scheduling of each query. AIDOS-I treats each distributed ontology query independently so that it does not consider the load balancing among sites and the caching for the repeatedly accessed data among the multiple distributed ontology queries. AIDOS-II supports all optimization techniques (pruning, DPFO, load balancing, and caching) mentioned in this paper. In addition, we present the effectiveness of pruning invalid and redundant queries.
### Table 6: Query Set

<table>
<thead>
<tr>
<th>ID</th>
<th>QUERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>U:(type, ?s, Student)</td>
</tr>
<tr>
<td>Q2</td>
<td>P:(type, ?a, Author)</td>
</tr>
<tr>
<td>Q3</td>
<td>U:(type, ?f, FullProfessor) ∧ U:(name, ?f, ?n)</td>
</tr>
<tr>
<td>Q5</td>
<td>U:(type, ?f, Professor) ∧ U:(publicationAuthor, ?s, ?f) ∧ U:(type, ?s, Student) ∧ U:(emailAddress, ?s, ?e) ∧ U:(advisor, ?s, ?f)</td>
</tr>
<tr>
<td>Q7</td>
<td>U:(type, ?p, Professor) ∧ U:(researchOn, ?p, ’semantic web’)</td>
</tr>
</tbody>
</table>

**Data set** We used Lehigh University Benchmark Data (LUBM)[1]. LUBM[14] is well-known OWL benchmark data containing the university ontology. In order to evaluate the performance of the distributed query processing, we produced three different types of ontologies (based on LUBM) but related to each other (i.e., UNIV(U), COLLEGE(C), PUB(P)), as briefly described in Figure 1. Several properties (e.g., researchOn) are newly inserted, and some instances are declared in different ontologies as different types. According to the three different ontology schemas, we generated variously sized OWL data for each schema: 10MB, 20MB and 50MB. Those generated OWL data were stored at three local sites along with the corresponding schemas. Also, we defined the set of bridge rules among the ontologies, as shown in Table 5.

**Query Set** We selected eight representative queries for the evaluation of the performance. Table 6 shows those selected queries. The criteria of selection are the complexity of the query and the bridge rule, the distribution degree of the answer, and the effect of pruning and caching.

The query processing time was measured by executing each query 5 times and averaging the time for 3 of the runs excluding the minimum and maximum times.

5.2. Experimental Results

**Pruning** Table 7 shows the effectiveness of our pruning technique in the query rewriting. EQ presents the number of queries generated by the rewriting algorithm in Figure 2. P.EQ and P.RULE show the number of queries after pruning and the applied pruning rule, respectively.

---

Table 7: The effectiveness of pruning

<table>
<thead>
<tr>
<th>QUERY</th>
<th>EQ</th>
<th>P.EQ</th>
<th>P.RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Q2</td>
<td>4</td>
<td>2</td>
<td>Rule 2</td>
</tr>
<tr>
<td>Q3</td>
<td>4</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>Q4</td>
<td>9</td>
<td>3</td>
<td>Rule 2</td>
</tr>
<tr>
<td>Q5</td>
<td>16</td>
<td>8</td>
<td>Rule 1</td>
</tr>
<tr>
<td>Q6</td>
<td>72</td>
<td>18</td>
<td>Rule 1,2</td>
</tr>
<tr>
<td>Q7</td>
<td>7</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>Q8</td>
<td>24</td>
<td>24</td>
<td>-</td>
</tr>
</tbody>
</table>

For Q4, the queries which are generated by replacing \( P: (\text{writer}, ?p, ?a) \) to \( U: (\text{publicationAuthor}, ?p, ?a) \) or \( C: (\text{writtenBy}, ?p, ?a) \) through the bridge rule \( B_7 \) and \( B_8 \) were removed by Rule 2, since, in \text{UNIV}, the domain of \( \text{publicationAuthor} \) is \( \text{Publication} \) and all writers of publications in \text{UNIV} are retrieved by the original query. Some queries derived from Q2 were also removed by Rule 2. In the data set used in our experiments, there is no intersection between \([C: \text{Student}] \) and \([U: \text{Person}] \). As a result, most queries including sub-queries such as \( C: (\text{type}, ?s, \text{Student}) \land U: (\text{advisor}, ?s, ?f) \) where the domain of \( \text{advisor} \) is \( U: \text{Person} \) have no answer. Thus, eight invalid queries for Q5 are eliminated by Rule 1. For Q6, both Rule 1 and Rule 2 are effectively used to remove the large number of invalid and redundant queries.

Consequently, the number of queries is in proportion to the complexity of query and in inverse proportion to the disjointness among sources related to the query.

**Load balancing** Table 8 shows the number of sub-plans (i.e., local query plans and intermediate distributed query plans) executed at each local site. S1, S2, and S3 present the number of plans assigned to each local site, and \( \text{TOTAL} \) represents the total number of the plans.

Generally, S1 processes the most plans among local sites S1, S2, and S3. The first reason is that S1 has the most powerful system. Second, for each query, most of reformulated queries require to access the ontology \text{UNIV} in S1.

The caching also has an influence on the load balancing among sites. From Q6 to Q8, AIDOS-I and AIDOS-II assign the different number of plans to each local site since AIDOS-II uses the caching, but AIDOS-I doesn’t. For example, in case of Q8, AIDOS-II expects that much time would be reduced by the caching in S2. Thus, AIDOS-II generates a plan where three more plans will be processed in S1, compared with AIDOS-I.

**Caching** Table 9 shows the number of sub-plans whose results are cached (\( \text{CACHED} \)) and the number of sub-plans which use the cached result (\( \text{U.CACHE} \)) for each query. From Q1 to Q4, there are no cached sub-plans since they have
Table 8: The number of plans assigned to each site (U:20MB, C:20MB, P:10MB)

<table>
<thead>
<tr>
<th>QUERY</th>
<th>TOTAL(T)</th>
<th>METHOD</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>2</td>
<td>AIDOS-I</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Q2</td>
<td>3</td>
<td>AIDOS-I</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q3</td>
<td>8</td>
<td>AIDOS-I</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Q4</td>
<td>3</td>
<td>AIDOS-I</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q5</td>
<td>28</td>
<td>AIDOS-I</td>
<td>18</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>18</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Q6</td>
<td>68</td>
<td>AIDOS-I</td>
<td>36</td>
<td>26</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>34</td>
<td>28</td>
<td>6</td>
</tr>
<tr>
<td>Q7</td>
<td>16</td>
<td>AIDOS-I</td>
<td>9</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>5</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Q8</td>
<td>40</td>
<td>AIDOS-I</td>
<td>21</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIDOS-II</td>
<td>24</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 9: The statistics of caching

<table>
<thead>
<tr>
<th>QUERY</th>
<th>CACHED(C)</th>
<th>C/T(%)</th>
<th>U,CACHE(U)</th>
<th>U/T(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q5</td>
<td>5</td>
<td>17.8</td>
<td>7</td>
<td>25</td>
</tr>
<tr>
<td>Q6</td>
<td>15</td>
<td>22.05</td>
<td>20</td>
<td>29.411</td>
</tr>
<tr>
<td>Q7</td>
<td>3</td>
<td>18.75</td>
<td>3</td>
<td>18.75</td>
</tr>
<tr>
<td>Q8</td>
<td>8</td>
<td>20</td>
<td>10</td>
<td>25</td>
</tr>
</tbody>
</table>

no common sub-queries among the queries. Thus, we only include the result of Q5 to Q8 in Table 9. From Q5 to Q8, the results of about 20% of sub-plans are cached and about 18-30% of sub-plans used the cached results without re-execution of the same query. As the query has the larger number of common sub-queries, the effectiveness of caching also increases.

Execution Time Figure 8 presents the response time for each query. Generally, AIDOS-II outperforms AIDOS-I by average 20.57% for Q5 to Q8 which used caching. During the query scheduling time, AIDOS-II considers the degree of workload for each local site and which plan needs to be cached. However, the scheduling time of AIDOS-II is almost the same with that of AIDOS-I. Also, this was tiny time enough to be ignored, compared with the query execution time. The major part which spends the most time is the processing each local ontology query in the local ontology management system. As a result, as the number of complex local query plans is larger, the response time also increases. In general, many local ontology queries are shared among queries. Consequently, AIDOS-II
effectively reduces the response time by caching the results of common queries and the load balancing according to the caching. In conclusion, the effectiveness of our optimization techniques are growing along with the increase of the data size and the number of queries which contain common sub-queries.

6. Conclusion

In this paper, we introduce an intelligent distributed ontology query processing method. We suggest more general models of the distributed ontology query and the semantic mapping among distributed ontologies than those of previous works so as to be applicable to more general environments and to retrieve the richer query answer. Also, through the query rewriting using the semantic mapping, we can obtain the integrated answer of a query over distributed ontologies without any global schema and the physical integration of different ontologies.

In addition, we propose several query optimization techniques. First, our approach eliminates invalid and redundant queries in the query rewriting phase using pruning rules. Second, Deeper Plan First Order (DPFO) scheduling method increases the parallelism of executing sub-plans belonging to a query but independent each other. Third, in order to distribute plans among queries, our approach considers the plans of the previously scheduled queries. Furthermore,
our approach uses the caching in order to remove the overhead due to the repeated execution of the same sub-queries among distributed queries. These load balancing and caching factors are reflected in the cost estimation for a query plan.

Finally, through the experimental results, we observed that our optimization techniques effectively reduced the response time. Especially, the effectiveness of the caching and the load balancing increases along with the growing of the data size and the number of distributed queries which access the common part of resources.

References


