Designing an Information Integration and Interoperability System — First Steps

Dongfeng Chen Rada Chirkova Fereidoon Sadri

October 19, 2006
Revised: November 15, 2006

1 Introduction

The problem of processing queries in semantic interoperability has received significant attention because of its relevance to different kinds of data formats (e.g., XML or relational) and data-management problems. Our goal is to design an information-integration and interoperability system to impove the efficiency of query processing. We introduce the concept of "super peers" [3] into data integration and query processing in our system. Used in peer-to-peer (P2P) systems, super peer nodes act as a centralized resource for a small number of clients, while connecting to each other to form a pure P2P network. We discuss system architecture, query processing, and query optimization in this framework.

We begin by presenting our motivation. A well-known peer-based REVERE system [1] supports decentralized data sharing, where query processing is peer-based. Its goal is to provide mediation between schemas in a decentralized, incremental, bottom-up fashion that does not require global standardization. The strength of the REVERE system is that it discards the global or mediated schema. However, its disadvantage is that it doesn't deal with "inter-source" queries very well. Given a collection of data sources, inter-source queries refer to the class of queries that can not be answered completely by aswering the query on each source separately and then combining the answers. Rather, there is a need to answer the query on combinations of data from multiple sources at a time.

The coordinator-based system [2] is a two-tier architecture, where the top tier is a coordinator on which queries are posed. It supports inter-source query processing. Unfortunately, its main disadvantage is that it does not scale very well, and the coordinator may become the bottleneck in designing large-scale data-sharing systems.

Thus, we introduce "super peers" (or *super coordinators*) into the heterogeneous structure to replace a single coordinator. Figure 1 depicts the topology of a super coordinator system. The main advantage of this approach is to accommodate large-scale data sharing without losing semantic interoperability among data sources.

2 Super Coordinators

A super coordinator is in charge of management and coordination of query execution at its subordinate sources. While each data source is connected to only one super coordinator, a super coordinator may have multiple subordinate sources. (We call a super coordinator and its data sources a *cluster*). Each super coordinator has the semantics of data stored at the (subordinate) data sources in its cluster, including local source declarations and mappings to the system model. Super coordinators are also in charge of query preparation, translation, and optimization. We note

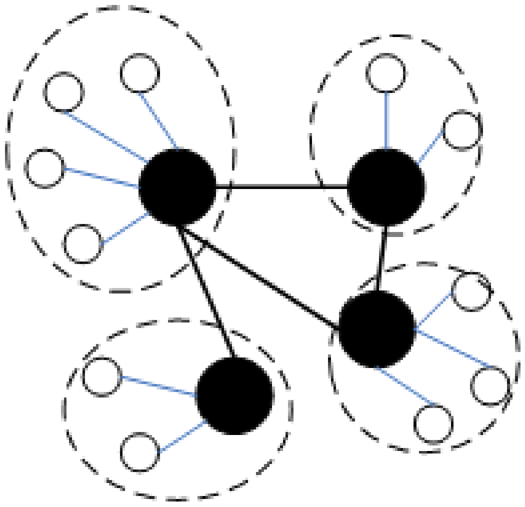

Figure 1: Proposed architecture of a super coordinator system.

that a super coordinator can become a single point of failure for its cluster. In this case, one of the data sources can be selected to assume the responsibilities of its super coordinator. The coordinator-based system can be viewed as a special case of a super-peer system with only one super coordinator (see Figure 2). Similarly, the peer-based system can be regarded as a super-peer system where each super coordinator has only one subordinate data source (see Figure 3).

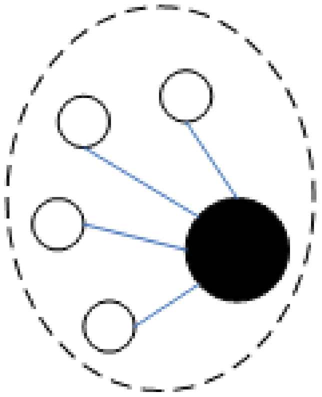

Figure 2: Architecture of a coordinator-based system.

On the other hand, super coordinators connect to each other to form a peer-based system. Super coordinators can serve as logical mediators or as query nodes. Schema mappings may be needed between small sets (of size, e.g., two) of super coordinators. Together with the mappings in their clusters, these schema mappings can be used by super coordinators to make use of relevant data anywhere in the system. As a result, query processing needs two mapping and data-translation steps in general: (a) mapping between a super coordinator and data sources in its cluster, and (b) mapping between a super coordinator and its peer super coordinators.

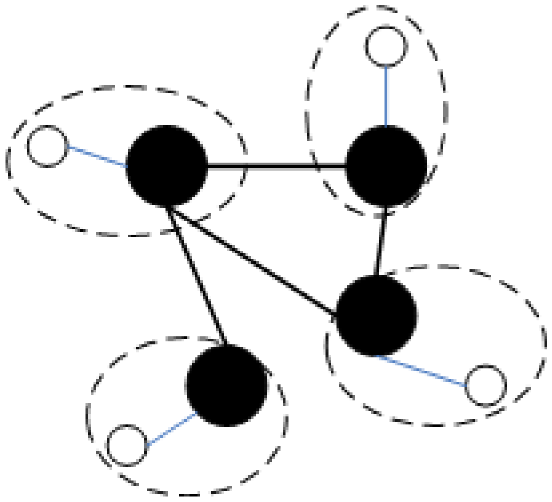

Figure 3: Architecture of a peer-based system.

3 System Architecture

As mentioned in Section 2, there exist two mapping stages: one is mapping between the schemas of a super coordinator and of its subordinate data sources (i.e., mapping within a super-coordinator cluster); the other is mapping which between peer super coordinators.

3.1 Mapping within a Super-Coordinator Cluster

This type of mapping can be set up using the process described in [2]. The database administrator at the super coordinator should determine the system model for the sources in the cluster under its coordination. The system model consists of binary predicates that are based on local source schema declarations, and provide a view of the sources data. The database administrator also provides mappings (or transformation rules) that map source data to the predicates in the system model of the super coordinator of the cluster.

3.2 Mapping between Peer Super Coordinators

In REVERE [1], each peer is a data source with its own schema. In the super-coordinator system, each peer is a coordinator, and the system model of the cluster plays the role of the peer schema. Mappings between the system models of super-coordinators should be provided. The ideal situation is when super-coordinators use the same set of binary predicates in their system model, in which case these mappings are trivial.

4 Query Processing

Systemwide user queries can be posed on any super coordinator, in a way that frees the user from having to learn the schemas or predicates of the remaining super coordinators. The super coordinator chosen as the user-query site (we call this super coordinator "originating super coordinator") is in charge of the query translation and optimization. A user query is translated into a set of local and inter-source subqueries. These subqueries are executed at the data sources. The originating super coordinator merges the answers to the subqueries posed on the data sources and returns the resulting final answer to the user.

4.1 Query Translation between Coordinators

In the first step of query processing, a user query should be translated (by the originating super coordinator) between super coordinators before it is executed at data sources. This translation depends on how we set up mappings between the super coordinators. We may do it in a way similar to our query-to-subquery translation process:

- Store mappings between coordinators into an XML document.
- Parse this XML document.
- Take a user query as an input, translate this query by combining mappings and other extra rules (if needed).

4.2 Query Translation in a Super Coordinator

After receiving the query Q' which is the result of translating the user query Q by combining mappings between coordinators, a super coordinator translates Q' into a collection of local and inter-source sub-queries for the data sources in its cluster, according to the concept of interoperability.

4.2.1 Deriving local subqueries

The process here is similar to that described in [2]. We now give a high-level description of the process:

- Define an XML Schema for mappings. Since mappings are composed of binary predicates, it is easy to build their XML Schema.
- Create an XML document for the mappings.
- Taking the query Q' and the XML document as inputs, expand the query by replacing each global predicate by the union of local fragments.

The pseudocode in Algorithm 1 gives the details of local sub-query generation for a data source.

Source mappings are stored in an XML file. Each mapping has a predicate, type, first argument, second argument, and glue variable. The global query is an XQuery stored in a file. The output sub-query is also an XQuery.

4.2.2 Deriving inter-source subqueries

The process of defining inter-source subqueries is similar to that for local subqueries, except that now we consider combinations of various data sources and their local fragments. In addition, we should take inter-source query optimization into account (See Section 5.1).

Algorithm 1: Local sub-query generation for a data source

```
input  : Global query $Q$, source mappings $m_i$ for source $i$
output: An output file containing local subquery $Q_i$ at source $i$ resulting from $Q$
1  Document xqueryDoc = getDoc($m_i$);
2  FileReader fr = new FileReader(Q);
3  BufferedReader br = new BufferedReader(fr);
4  while (strLine = br.readLine()) $\neq$ null do
5      if strLine contains a predicate then
6          retreive predicate, first argument, second argument, and glue variable from
           xqueryDoc;
7          replace "*-*/tuple" in strLine by the local predicate from xqueryDoc;
8          if strLine contains arguments then
9              replace "tuple/*" by the corresponding arguments in the predicate;
10             write the new strLine into the output file;
11             continue;
12         if strLine contains a variable $X$ then
13             replace $X$ by the glue variable in the corresponding predicate;
14     else if strLine merely contains a variable $X$ then
15         replace $X$ by the glue variable in the corresponding predicate;
16     else
17         don't change strLine;
18     write strLine into the output file;
19 close all opened files;
```

4.2.3 Query Processing

Local subqueries are executed locally at data sources. We can use an XQuery Engine, e.g., open source engines (such as SAXON), or commercial products (such as IBM DB2).

For inter-source subqueries, query optimization can achieve a number of goals. One goal (see Section 5.1) is to eliminate inter-source processing which typically has high costs. Another goal is to improve the efficiency of inter-source processing when we can not eliminate inter-source processing.

Consider query Q' defined using a join of predicates p and q, such that the processing of Q' involves inter-source processing of $p_i \bowtie q_j$, $i \neq j$.

1. One approach is to first materialize one of the predicates (say source i) locally and send it to a temporary directory of the other source (source j). Then only transform query variable declarations on the second predicate to its corresponding document, and execute the query (using the first predicate and the second document). After execution, this temporary directory is removed from source j.

2. Another approach is to use outer joins: Each source computes the full outer join of the fragments it has, then sends the result to its super coordinator. Super coordinators should take partial outer joins, then forward the results to the coordinator on which the user query is posed, using the transitive closure of the respective coordinator-to-coordinator mappings.

4.3 Merging Results of Subqueries

After query execution, super coordinators collect all the results of the local and inter-source subqueries. Super coordinators then merge these intermediate results into the final answer to the user query.

Here is what the *Merge Operation* does: Given k XML documents with the same schema, produce a single document that is the result of merging the k documents in a meaningful way. There exist many ways to merge multiple documents: One is *linear merge*, where we merge documents one by one; another is *binary merge*, where we merge several pairs of documents at a time; yet another is *parallel merge*, where we merge all documents at the same time.

Algorithm 2 shows one way to merge XML documents.

Algorithm 2: Merging XML Data

```
input  : XML documents and their schema
output: Single XML document $D$

$parseSchema(fileStr)$;
$D = mergeAll(dirStr)$;

Procedure parseSchema(schemaStr)
retrieve keys, unique nodes and other constraints from the schema;
treewalk all elements from root and classify elements into our pre-defined types;

Function mergeAll(dirStr)
foreach $XML$ in the directory $dirStr$ do
    $DocumentnextDoc = getDoc(XML)$;
    $rsDoc = mergeTwoDocs(rsDoc, nextDoc)$;
write $rsDoc$ into an XML document $D$;
return $D$

Function mergeTwoDocs(rsDoc, nextDoc)
get root element $root1$ from $rsDoc$;
get root element $root2$ from $nextDoc$;
foreach  element under $root1$ do
    $mergeElements(element, root2, rsDoc, nextDoc)$;
return $rsDoc$;
```

In Algorithm 2, all XML documents under the directory *dirStr* are merged into a single XML document D. Procedure parseSchema() is used to obtain keys and integrity constraints (e.g., logical identifires for elements, implicit and explicit constraints) from the XML schema. Procedure parseSchema() can also classify all elements in the XML documents into several types, which are specified in our merge *rules*. A number of methods can be used to merge multiple documents. For now, we merge two documents at a time. The temporary result is used as one of those two documents, which is used for the merge at the next step (see Algorithm 3).

5 Query Optimization

5.1 When can Inter-Source Queries be Avoided?

Terminology and Notations:

- Number of XML sources is n, number of predicates (binary relations) in the system model is m, and we are considering a user query involving the join of k predicates.
- Relations (Predicates) in the user query: $r_1, \dots, r_k$.
- Fragments of predicates: Each predicate r_i has (up to) n fragments, $r_i^1, \dots r_i^n$, where r_i^j is the fragment of r_i at source j.

Given a user query $Q = r_1 \bowtie \cdots \bowtie r_k$, it can be computed by k^n subqueries that arise from expanding Q as follows:

$$\begin{aligned} Q &= r_1 \bowtie \cdots \bowtie r_k \\ &= (r_1^1 \cup \cdots \cup r_1^n) \bowtie \cdots \bowtie (r_k^1 \cup \cdots \cup r_k^n) \\ &= (r_1^1 \bowtie r_2^1 \bowtie \cdots \bowtie r_k^1) \cup (r_1^1 \bowtie r_2^1 \bowtie \cdots \bowtie r_k^2) \cup \cdots \cup (r_1^n \bowtie r_2^n \bowtie \cdots \bowtie r_k^n) \end{aligned}$$

Among these k^n subqueries, only n are *local* subqueries. That is, each subquery $(r_1^i \bowtie r_2^i \bowtie \cdots \bowtie r_k^i)$, $i = 1, \dots, n$, uses only fragments from a single source i, and is called a *local* subquery. The remaning $k^n - n$ subqueries need fragments from multiple (at least two) sources, and are called *inter-source subqueries.*

We will henceforth use k-tuples to represent the subqueries. For example, $(x_1, \dots, x_k)$ represents the subquery $r_1^{x_1} \bowtie \cdots \bowtie r_k^{x_k}$, namely, the join fragments $r_1^{x_1}, \dots, r_k^{x_k}$.

The following result is from [2]:

Theorem 1 *Given $r_i(A,B)$ and $r_j(B,C)$, if B is the key of r_2 and a foreign key constraint holds from r_i to r_j on B, then $r_i^x \bowtie r_j^x \supseteq r_i^x \bowtie r_j^y$ for all x and y. We say $r_i^x \bowtie r_j^x$* subsumes *$r_i^x \bowtie r_j^y$, for all x and y. Note that $r_i^x \bowtie r_j^x$ is the join of fragments of r_i and r_j at the same source x. Hence, the* local *subqueries $r_i^x \bowtie r_j^x$, $x = 1, \dots, n$, subsume all* inter-source *subqueries $r_i^x \bowtie r_j^y$, $y \neq x$, in this case.* ■

Hence, if the conditions of the theorem hold, then $r_i \bowtie r_j$ can be computed by computing only the local subqueries. We would like to extend this result to general select-project-join queries with any number of predicates. Thus, the question we pose is what are the conditions under which all inter-source subqueries are subsumed by local ones. First we need some definitions and preliminary results.

DEFINITION 1 (Local Join Property) Given $r_i(A,B)$ and $r_j(B,C)$, if B is the key of r_j and a foreign key constraint holds from r_i to r_j on B, we say $r_i \bowtie r_j$ has the *local join* property, and denote it by $r_i \rightarrow r_j$. ■

Note that in the most general case, local join properties may exist between *fragments* of predicates at a source. That is, for a source h, we may have that B is the key for $r_j^h(B,C)$ and foreign key constraint holds from $r_i^h(A,B)$ to $r_j^h(B,C)$ on B. So, we are assuming that our notation $r_i \rightarrow r_j$ is a shorthand for: for all h, $r_i^h \rightarrow r_j^h$. In practice, a local join property is determined by the semantics of the predicates, and is expected to hold for the data at all sources. But a local XML schema at a given source does not necessarily *enforce* the constraint.

The Local Join Graph

DEFINITION 2 Let $r_1, \ldots, r_m$ be all the predicates in the system model. We can represent the local join property by a graph $G = (N, E)$, where the set of nodes N corresponds to the predicates $r_1, \ldots, r_m$, and $(r_i, r_j) \in E$ if r_i and r_j have the local join property. ■

Theorem 2 *Given the user query $Q = r_1 \bowtie \cdots \bowtie r_k$, the local subqueries subsume all inter-source subqueries if the local join graph restricted to $\{r_1, \ldots, r_k\}$ contains a directed spanning tree.* ■

We will use the following Lemma in the proof of Theorem 2.

Lemma 1 *If $r_i \rightarrow r_j$, then $(r_1^{s_1} \bowtie \cdots \bowtie r_i^x \bowtie \cdots \bowtie r_j^x \bowtie \cdots \bowtie r_k^{s_k})$ subsumes $(r_1^{s_1} \bowtie \cdots \bowtie r_i^x \bowtie \cdots \bowtie r_j^y \bowtie \cdots \bowtie r_k^{s_k})$ for all s_i, x, and y.*
Proof: Since $r_i \rightarrow r_j$, then $r_i^x \bowtie r_j^x \supseteq r_i^x \bowtie r_j^y$ for all x and y by Theorem 1. The lemma follows. ■

Proof of Theorem 2:

Let G be the local join graph of the predicates. Without loss of generality, assume r_1 is the root, and $r_1, r_2, \ldots, r_k$ is the depth first search order of the directed spanning tree of the predicates in the user query. The user query can be written as $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_k$. We will show that each inter-source subquery $r_1^{s_1} \bowtie r_2^{s_2} \bowtie \cdots \bowtie r_k^{s_k}$ is subsumed by the local subquery $r_1^{s_1} \bowtie r_2^{s_1} \bowtie \cdots \bowtie r_k^{s_1}$. For simplicity, we will use our notation to represent the above subsumption as follows: $(s_1, s_1, \ldots, s_1) \supseteq (s_1, s_2, \ldots, s_k)$.

We will show, by (backward) induction, from $j = k$ to $j = 1$, that

$$(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_1, s_{j+1}, \ldots, s_k)$$

Basis $j = k$: Obviously, $(s_1, s_1, \ldots, s_1) \supseteq (s_1, s_1, \ldots, s_1)$.

Induction: Assume the inductive hypothesis holds for j, that is, $(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_1, s_{j+1}, \ldots, s_k)$. We want to show $(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_1, s_j, \ldots, s_k)$. Let parent of r_j in the directed spanning tree be r_i. Then, since $r_1, \ldots, r_k$ is the depth-first ordering of the tree, we must have $i < j$. By $r_i \rightarrow r_j$ and Lemma 1, we have $(r_1^{s_1} \bowtie \cdots \bowtie r_i^{s_1} \bowtie \cdots \bowtie r_j^{s_1} \bowtie \cdots \bowtie r_k^{s_k})$ subsumes $(r_1^{s_1} \bowtie \cdots \bowtie r_i^{s_1} \bowtie \cdots \bowtie r_j^{y} \bowtie \cdots \bowtie r_k^{s_k})$, for all y. Let $y = s_j$. Written in our notation: $(s_1, \ldots, s_1, s_{j+1}, \ldots, s_k) \supseteq (s_1, \ldots, s_j, s_{j+1}, \ldots, s_k)$. Combined with inductive hypothesis and by transitivity of subsumption: $(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_1, s_{j+1}, \ldots, s_k) \supseteq (s_1, \ldots, s_j, s_{j+1}, \ldots, s_k)$. This completes the proof of the induction.

Hence, the local subqueries $(i, \ldots, i)$, $i = 1, \ldots, n$ (n is the number of sources), subsume all inter-source subqueries. No inter-source subquery is needed in the computation of the user query. ■

Theorem 2 gives a sufficient condition, based on key and foreign key constraints, where no inter-source subquery is needed for the evaluation os a user query. The question naturally arises whether the condition of Theorem 2 is also necessary? In other words, if the restriction of the local join graph to query predicates does not have a directed spanning tree, does the evaluation of the query require evaluation of some inter-source subqueries? In the following sequence of results we answer this question in the positive.

Let G be the local join graph of all predicates in the system model. Consider a user query Q with k predicates. Let G' be the restriction of G to the predicates in Q. We will first study the case where G' is acyclic.

Theorem 3 *Let Q and G' be as above. If G' is acyclic and further does not have a directed spanning tree, then a database instance exists where (at least one) inter-source subquery is not subsumed by any local sunqueries.*

Proof: Let user query Q involve predicates $r_1, \ldots, r_k$ Since G' is not cyclic, it must have a node with indegree 0. Assume node r_i in G' has indegree 0. Further, there should be at least one node, r_j, that is not reachable from r_i. Otherwise, G' would have a directed spanning tree rooted at r_i. Further, r_i is also not reachable from r_j, since its indegree is zero.

Consider the following sets of nodes in G':

- N_1 is the set of nodes in G' that are reachable from r_i (including r_i itself). This set is non-empty.
- N_2 is the set of nodes from which r_j can be reached (including r_j itself). This set is also non-empty. Note that since r_j is not reachable from r_i, then N_1 and N_2 must be disjoint.
- N_3 is the set of remaining nodes, namely, $N_3 = N - (N_1 \cup N_2)$, where N is the set of nodes of G'.

We will construct an instance of a database, with two sources, where the inter-source subquery

$$\bowtie_{r_l \in N_1 \cup N_3} (r_l^1) \bowtie_{r_l \in N_2} (r_l^2)$$

is not subsumed by any local subqueries. The database instance is constructed as follows:

- Fragments at source 1: For each predicate $r_l(A, B) \in N_1 \cup N_2$, r_l^1 contains the tuple (a, b). That is, for each attribute, we use a unique value (such as a for A and b for B) and build one tuple for each fragment determined by the attributes of the fragment. Every other fragment of source 1, that is r_l^1 where $r_l \in N_3$, is empty.
- Fragments at source 2: r_i^2 is empty. All other fragments r_l^2 with the schema $r_l(A, B)$ contain one tuple (a, b) as explained above.

We can verify that

1. Fragments at sources 1 and 2 satisfy all the constraints (key and foreign key) specified by G'.

   Proof: First, all key constraints are satisfied since each fragment is either empty or has one tuple. Further, for each edge $r_l \rightarrow r_m$ in G', and each source, the database construction guarantees that either the fragment of r_l is empty, or the fragment of r_m is non-empty. That is, if, wlog, the schemas of the predicates are $r_l(A, B)$ and $r_m(B, C)$, and fragment of r_l is not empty (hence it has one tuple (a, b)), then the fragment of r_m also has one tuple (b, c). Hence, foreign key constraints are also enforced.

2. The local subqueries $r_1^1 \bowtie \cdots \bowtie r_k^1$ and $r_1^2 \bowtie \cdots \bowtie r_k^2$ are both empty.

   Proof: Since r_j^1 and r_i^2 are empty.

3. The inter-source subquery $\bowtie_{r_l \in N_1 \cup N_3}(r_l^1) \bowtie_{r_l \in N_2}(r_l^2)$ contains one tuple, namely the tuple $(a, b, \ldots)$ consisting of the values associated with attributes that appear in the schemas of $r_1, \ldots, r_k$.

   Proof: All fragments in the join are non-empty, and contain the tuple of values associated with their attributes. ■

Theorem 3 settles the question whether having a direceted spanning tree is sufficient *and* necessary for the case of acyclic local join graphs. In practice, there is one type of cycles that occur frequently in local join graphs of system models. Consider, for example, an entity with key A and properties (attributes) B and C that are required and single-valued. The system model will contain predicates $r_1(A, B)$ and $r_2(A, C)$, and the local join graph has edges $r_1 \rightarrow r_2$ and $r_2 \rightarrow r_1$. In fact, an entity with key A and single-valued, required properties $B_1, \ldots, B_m$ gives rise to nodes corresponding to $r_1(A, B_1), \ldots, r_m(A, B_m)$ in the local join graph, with edges between every such r_i and r_j pairs. The same phenomenon is also observed when an n-ary relationship, $n > 2$, is represented by a set of binary predicates. In the following, we extend our results to the important case of local join graphs that contain this type of cycles.

Let G be the local join graph of the system model, Q be a user query, and G' be the restriction of G to the predicates of Q. We write $r_i \leftrightarrow r_j$ if $r_i \rightarrow r_j$ and $r_j \rightarrow r_i$. Obviously, $\leftrightarrow$ is an equivalence relation for the above cases. For the graph G', we define a *reduced* graph G'' by collapsing all nodes in each equivalence class of $\leftrightarrow$ into a single node. Obviously, G'' does not have any cycles of length 2. But, in general, it can still be cyclic. We can extend Theorem 3 to the case where G'' is acyclic.

Theorem 4 *Let Q and G'' be as above. If G'' is acyclic and further does not have a directed spanning tree, then a database instance exists where (at least one) inter-source subquery is not subsumed by any local sunqueries.*

Proof: Proof of Theorem 3 can be applied to G'' as follows: We define the node sets N_1, N_2, N_3 as before for G'' (instead of G'). The same database instance construction is used in this case, except, we observe that a node in G'' may represent an equivalence class of nodes in G', and hence it represents a set of fragments. The instance construction is applied to all predicate fragments represented by a node in G''. Hence, as dictated by the construction, either all these fragments are empty, or, for each $r_l(A, B)$ in the equivalence class represented by the node, the corresponding fragment has the tuple (a, b) in it.

We can verify that the database instance constructed in this way satisfies all key and foreign key constraints. The constrains for all fragments within one equivalence class is satisfied since either all are empty or all contain one tuple as determined by their schema. Foreign key constraints between fragments belonging to different nodes in G'' are also satisfied by the construction of N_1, N_2, and N_3 as before.

Further, similar to the previous case, the database instance will have at least one non-empty inter-source subquery, while the local subqueries are empty. ■

Observation 1 *If G'' has a directed spanning tree, then so does G'. Hence, by Theorem 2, all inter-source subqueries are subsumed by local subqueries in this case.* ■

Finally, we can extend our results to general local join graphs by collapsing all cycles. The following definition formalizes the approach.

Let G be the local join graph of the system model, Q be a user query, and G' be the restriction of G to the predicates of Q. We say r_j is *reachable* from r_i if there is a path from r_i to r_j in G'. We write $r_i \Leftrightarrow r_j$ when r_j is reachable from r_i and r_i is reachable from r_j. Obviously, $\Leftrightarrow$ is an equivalence relation. For the graph G', we define a *totally-reduced* graph G''' by collapsing all nodes in each equivalence class of $\Leftrightarrow$ into a single node. Obviously, G''' is acyclic. Note that we do not consider self loops (of the form $r_i \rightarrow r_i$) that may appear in G'''. Self loops represent foreign key constraints within predicates in an equivalence class, and are not relevant to the database instance construction utilized in the proofs of our theorems.

Theorem 5 *Let Q and G''' be as above. No inter-source subquery is needed in the evaluaiotpn of Q if and only if G''' has a directed spanning tree.*

Proof: Proofs of Theorems 3 and 4 can be applied to G''' as well. Further, Observation 1 applies to G''' as well. Theorem follows. ■

6 Summary and Conclusion

What we have done is the following:

- We proposed our prototype of super-coordinator systems.
- We presented mapping formalisms among data sources.
- We implemented translating a query into local subqueries.
- We proposed theorems about eliminating inter-source query processing and query optimization.
- We implemented merge operation.

What we are doing and going to do may be:

- To complete the design of our system.
- To enhance interoperability between super coordinators.
- To build up the mapping program between super coordinators.
- To implement inter-source query execution.
- To implement the full outer join for XML data.
- Other relevant issues.

References

[1] Alon Halevy, Oren Etzioni, AnHai Doan, Zachary Ives, Jayant Madhavan, Luke McDowell, and Igor Tatarinov. Crossing the structure chasm. In *CIDR*, pages 117–128, 2003.

[2] Laks V. S. Lakshmanan and Fereidoon Sadri. Interoperability on XML data. In *International Semantic Web Conference (ISWC)*, pages 146–163, 2003.

[3] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *ICDE*, pages 49–60, 2003.

Algorithm 3: Merging XML Data (continued)

```
Procedure mergeElements(e1, eleInNextDoc, rsDoc, nextDoc)
get this $e1$'s type $typeVal$;
switch typeVal do
    case Single Requried Leaf
        check existance of the corresponding element $e2$ in $nextDoc$;
        compare $e1$'s value with $e2$'s value;
        if neither $e2$ exists, nor its value is equal to $e1$'s, print ERROR;
        break;
    case Single Optional Leaf
        if $e2$ exists, compare $e1$'s value with $e2$'s value;
        break;
    case Single Requried NonLeaf
        check existance and size of $eleInNextDoc$ in $nextDoc$;
        $ele2 = eleInNextDoc$'s child;
        foreach element under $e1$ do
            $mergeElements(element, ele2, rsDoc, nextDoc)$;
        break;
    case Single Optional NonLeaf
        if the size of $eleInNextDoc$ is 1, CALL $mergeElements()$ recusively;
        break;
    case Multi Unique Leaf
        merge $eleInNextDoc$'s children;
        remove duplicates which have the same logical identifiers;
        break;
    case Multi optional Leaf
        merge $eleInNextDoc$'s children, if they exist;
        remove duplicates which have the same logical identifiers;
        break;
    case Multi Unique NonLeaf
        check the unique node in the result of $parseSchema()$;
        CALL $mergeElements()$ recusively;
        break;
    case Multi Optional NonLeaf
    case Multi Reduplicate Leaf
    case Multi Reduplicate NonLeaf
        merely attach $eleInNextDoc$'s children;
        break;
    case Set Node
        CALL $mergeElements()$ recusively;
        break;
    otherwise
        print ERROR
        break;
```