Designing an Information Integration and Interoperability System – First Steps

Dongfeng Chen Rada Chirkova Fereidoon Sadri
October 19, 2006

1 Introduction

The problem of processing queries in semantic interoperability has received significant attention because of its relevance to all kinds of data formats (e.g., text, XML) and data management problems. We try to design an information integration and interoperability system to impove the efficiency of query processing. In our system, we introduce "super peer" into data integration and query processing. "Super peer" [3] is orignally used in Peer-to-Peer systems. "Super-peer" nodes in P2P systems act as a centralized resource for a small number of clients, but these super peers connect to each other to form a pure P2P network.

The REVERE system [1] (a peer-based system) 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 schema or mediated schema. However, its disadvantage is that it doesn't deal with inter-source queries very well.

The coordinator-based system [2] is a two-tier architecture, where the top tier is a coordinator on which queries are posed. It has inter-source query processing. Unfortunately, its main disadvantage is that it doesn't scale very well, and the coordinator may become the bottleneck when people try to design large-scale data sharing systems.

Thus, we introduce "super peers" (now we call them super coordinators) into the heterogeneous structure by replacing the coordinators. Figure 1 illustrates what the topology of a super coordinator system looks like. The main advantage of this method that we propose is to design large-scale data sharing systems, without losing semantic interoperability among data sources.

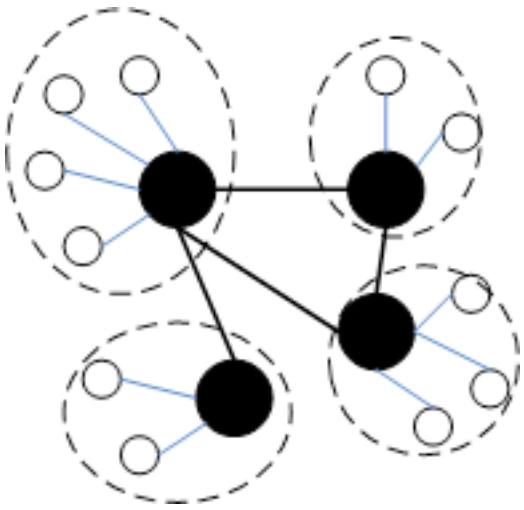


Figure 1: Illustration of a super coordinator system.

2 Super Coordinator

Super coordinators are in charge of coordination of sub-query execution at disparate sources, which don't just act as a coordinator. One data source is connected to only one super coordinator, but a super coordinator may have multiple sources. Each super coordinator has semantics of data stored at data sources, e.g., local source declarations, predicates, and mappings. Super coordinators are also in charge of query preparation, translation, and optimization. We note that a super coordinator can become a single point of failure for its cluster (we call a super coordinator and its data sources a **cluster**). In this case, one or more data sources are selected to form a single "virtual" super coordinator. For example, this selected data source has the same semantics of other data sources as its super coordinator does. The coordinator-based system is actually a super-peer system with only one super coordinator(see Figure 2).

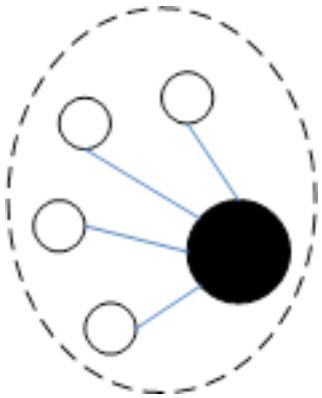


Figure 2: Illustration of a coordinator-based system.

On the other hand, super coordinators connect to each other to form the peer-based system. Super coordinators can serve as logical mediators, and/or query nodes. Schema mappings happen between two (or a small set of) super coordinators. Together with mappings in their clusters, super coordinators can make use of relevant data anywhere in the system by using schema mappings. As a result, query processing needs two mapping steps: one step happens between super coordinators via mapping schemas and data translating; the other one happens before super coordinators translate this query into local and/or inter-source subqueries. Actually the peer-based system can be regarded as a super-peer system where each super coordinator has only one data source(see Figure 3).

3 System Architecture

As mentioned in Section 2, there exist two mapping stages: one is schema mappings which happens between two super coordinators; the other is predicator mapping which happens among a super coordinator.

3.1 Mapping From Coordinator To Coordinator

In PDMS, each peer is a data source, which has its own schema. However, in the super-coordinator system, each peer is a coordinator without its own schema. Thus, we have to implement our mapping formalism.

1. Do as PDMS does. We can set up schemas for these super coordinators, so that we can use the same mapping program between super coordinators. In order to build a schema for a super coordinator,

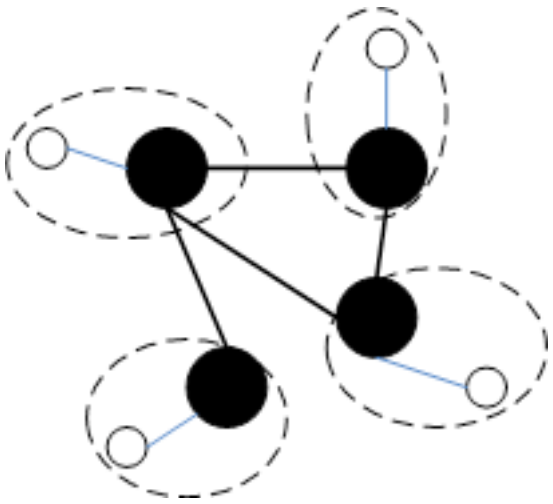


Figure 3: Illustration of a peer-based system.

we can merge all schemas of data sources under this coordinator into a new one. Alternatively, we can deduce a new schema from its local source declarations and binary predicates.

2. Modify the mapping program for binary predicates instead of for XML schemas. Here we will build mappings among binary predicates of varied coordinators.

3.2 Mapping in a super coordinator

This part is similar to what they did in the paper "Interoperability on XML Data". The database administrator or super coordinator should select binary predicates or properties based on local source declarations and their ontology. These administrator can also provide mappings or transformation rules that map source data to the predicates.

4 Query Processing

Queries can be posed over any super coordinator, without having to learn the schema or predicate of other super coordinators. Super coordinator is in charge of query translation and optimization. A user query is translated into a set of both local and inter-source subqueries. These subqueries are executed at data sources. Super coordinator merges the results of subqueries from data sources, then form the final answer for the user.

4.1 Query Translation between Coordinators

Firstly, a user query should be translated between coordinators before it is executed at data sources. This translation depends on how we set up mappings between super coordinators. We may do it as we have already done in the query-to-subquery translation.

- 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 translated from the user query Q by combining mappings between coordinators, super coordinator has to translate Q' into a collection of local and inter-source sub-queries according to the concept of Interoperability.

4.2.1 Local subquery

This part is well described in the paper "Interoperability on XML Data". The following shows the implementation:

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

The following Algorithm 1 shows more details about 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 Inter-source subquery

This translation is similar to that of local subqueries, except considering various data sources and their local fragments. At the mean time, we should take inter-source query optimization into account.

We propose extensions of theorems in [2] to eliminate inter-source query processing.

Observation 1 *Let a global query Q be defined as follows:* Q :- r(A,B), s(A,C), t(A, D). *Inter-source query processing in the process of evaluating Q can be eliminated if all these conditions are satisfied: r.A is the key of r, s.A is a foreign-key constraint to r.A, and t.A is a foreign-key constraint to s.A.*

Observation 2 *Let Q be defined as follows:* Q :- r(A,B), s(A,C), t(C, D). *Inter-source query processing in the process of evaluating Q can be eliminated if all these conditions are satisfied: r.A is the key of r, s.C is the key of s, s.A is a foreign-key constraint to r.A, and t.C is a foreign-key constraint to s.C.*

These two observations can be extended to multiple relations, as follows.

Observation 3 *Let Q be defined as follows:* $r_1(A, B_1) \bowtie r_2(A, B_2) \bowtie \ldots \bowtie r_n(A, B_n)$. *Inter-source query processing in the process of evaluating Q can be eliminated if all these conditions are satisfied:*
$r_1.A$ *is the key of* r_1*; and*
$r_i.A$ *is the key of* r_i*, and* $r_i.A$ *is a foreign-key constraint to* $r_{i-1}.A$*, if* $1 < i < n$*; and*
$r_n.A$ *is a foreign-key constraint to* $r_{n-1}.A$.

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

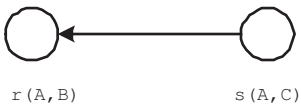
input : Global query Q, source mappings m_i for source i
output: An output file containing local subquery Q_i at source i resulting from Q

```
1  Document xqueryDoc = getDoc(m_i);
2  FileReader fr = new FileReader(Q);
3  BufferedReader br = new BufferedReader(fr);
4  while (strLine = br.readLine()) ≠ 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;
```

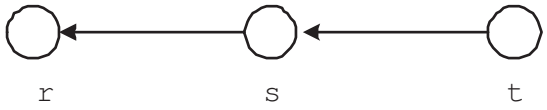
Observation 4 *Let Q be defined as follows: $r_1(A_1, B_1) \bowtie r_2(A_2, B_2) \bowtie \ldots \bowtie r_n(A_n, B_n)$. Inter-source query processing in the process of evaluating Q can be eliminated if all these conditions are satisfied:*
Only one of r_1's attributes is the key; and
Only one of r_i's attributes is the key, and one of r_i's attributes is a foreign-key constraint to r_{i-1}'s key, if $1 < i < n$; and
One of r_n's attributes is a foreign-key constraint to r_{n-1}'s key.

Before we propose our theorem, we define a graph for a predicate.

DEFINITION 1 *A* n*ode in the directed graph represents a predicate, and an* e*dge represents a foreign-key constraint. A simple graph is the following:*
r.A is the key of r, s.A is a foreign-key constraint to r.A,



So far, we can rewrite Observation 1 and Observation 2 into the following:



Theorem 1 *No inter-source queries on Set A of relations are needed, if and only if A has a Direct Spanning Tree with only one root based on foreign-key constraints.*

4.2.3 Query Optimization

Query optimization could happen at various cases. One case (see Section 4.2.2) is to eliminate inter-source processing which costs much. One case is to improve the efficiency of inter-source processing when we can't eliminate inter-source processing.

Lets consider Q' involving the join of predicates p and q, and the inter-source processing of $p_i \bowtie q_j$, $i \neq j$.

1. An approach is to first materialize one of the predicates locally (source i) and send it to a temperate 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 temporate directory is removed from source j.

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

4.3 Query Execution

When subqueries are executed locally at data sources, we use an XQuery Engine, e.g., open source engines (SAXON), or commercial products (the latest IBM DB2).

After query execution, super coordinator collect all of results for both local subqueries and inter-source subqueries. Super coordinator has to merge the intermediate results into the final answer for the user query.

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

Algorithm 2 shows one way to merge XML documents.

Algorithm 2: Merging XML Data

input : XML documents and their schema
output: One 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*. There are kinds of methods to merge multiple documents. Here we merge two documents one time. The temporary result is used as one of those two documents which will be merged next time (see Algorithm 3).

5 Summary

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 Y. Halevy, Oren Etzioni, AnHai Doan, Zachary G. Ives, Jayant Madhavan, Luke McDowell, and Igor Tatarinov. Crossing the structure chasm. In *CIDR*, 2003.

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

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