Efficient Query Reformulation in Peer Data Management Systems

Igor Tatarinov, Alon Halevy
{igor, alon}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195, USA

ABSTRACT

Peer data management systems (PDMS) offer a flexible architecture for decentralized data sharing. In a PDMS, every peer is associated with a schema that represents the peer's domain of interest, and semantic relationships between peers are provided locally between pairs (or small sets) of peers. By traversing *semantic paths* of mappings, a query over one peer can obtain relevant data from any reachable peer in the network. Semantic paths are traversed by reformulating queries at a peer into queries on its neighbors.

Naively following semantic paths is highly inefficient in practice. We describe several techniques for optimizing the reformulation process in a PDMS and validate their effectiveness using real-life data sets. In particular, we develop techniques for pruning paths in the reformulation process and for minimizing the reformulated queries as they are created. In addition, we consider the effect of the strategy we use to search through the space of reformulations. Finally, we show that pre-computing semantic paths in a PDMS can greatly improve the efficiency of the reformulation process. Together, all of these techniques form a basis for scalable query reformulation in PDMS.

To enable our optimizations, we developed practical algorithms, of independent interest, for checking containment and minimization of XML queries, and for composing XML mappings.

1. INTRODUCTION

Sharing data among multiple sources is crucial in a wide range of applications, including enterprise data management, large-scale scientific projects, sharing data between government agencies and data sharing on the World-Wide Web. Data integration systems offer an architecture for data sharing in which data is queried through a central *mediated schema*, but the data itself stays at the sources in their local schemas. Recent data integration products have been successful at enabling data sharing, but on a relatively small scale.

The main limitation of data integration system is the need for the mediated schema. In many applications, owners of data want to be able to share data without any central authority (even at the logical level). In some cases, the data is so diverse that a mediated schema would be almost impossible to build or to agree upon, and very hard to maintain over time.

Peer data management systems (PDMS) have been proposed as an architecture for decentralized data sharing [20, 1, 27, 5, 7]. A PDMS (see Figure 1) consists of a set of (physical) peers. Each peer has an associated schema that represents its domain of interest. Some peers store actual data, and describe which data they store relative to their schema. The stored data does not necessarily cover all aspects of the peers' schema.

Mappings between disparate schemas in a PDMS are provided *locally* between pairs or small sets of peers. When a query is posed at a peer, the system can obtain relevant data from any peer in the PDMS that is connected through a *semantic path* of mappings. In the PDMS of Figure 1, supporting a web of database research-related data, a query on the Roma node can obtain data from the Paris node (and vice versa) by following a path through DB-Projects and Stanford. Unlike a hierarchy of data integration systems or mediators, a PDMS supports any arbitrary network of relationships between peers. In fact, PDMSs are a strict generalization of data integration systems, as some peers in the PDMS may act as mediators to other peers. Furthermore, a PDMS also provides a platform for web-scale sharing of data, as envisioned by the Semantic Web [6].

The main advantages of PDMSs are (1) peers can share data in diverse and overlapping domains without a mediated schema, (2) joining a PDMS can be done opportunistically, i.e., a peer can provide a mapping to the most convenient (e.g., similar) peer(s) already in the PDMS, and (3) a peer can pose a query using its *own* schema without having to learn a different one.

Query reformulation: The key step in query processing in a PDMS is *reformulating* a peer's query over other peers on the available semantic paths. Broadly speaking, the PDMS starts from the querying peer and reformulates the query over its immediate neighbors, then over their immediate neighbors, and so on. Whenever the reformulation reaches a peer that stores data, the

SIGMOD 2004 June 13-18, 2004, Paris, France.

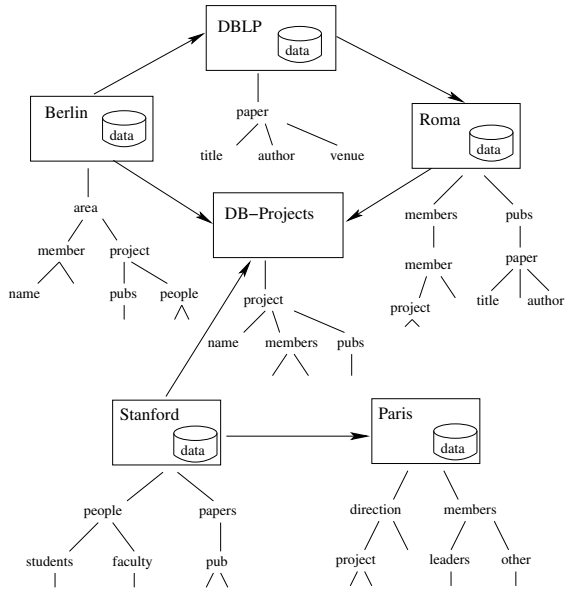

Figure 1: A PDMS for the database research domain. A fragment of each peer's XML schema is shown as a labeled tree. Arrows indicate that there is mapping between the schemas of the peers.

appropriate query is posed on that peer, and additional answers may be found. Since peers typically do not contain *complete* information about a domain, any relevant peer may add new answers. Furthermore, different paths to the *same* peer may yield different answers. For example, in Figure 1, there are two paths between the Roma peer and the Berlin peer. Since DBLP does not model projects, data about projects in Roma cannot be used for answering a query on the Berlin peer if we use the path that goes through DBLP. In contrast, the path through DB-Projects does enable that data flow.

However, following all semantic paths naively leads to several inefficiencies. First, the algorithm may follow many paths that can be pruned early on. Second, the algorithm follows many paths that result in redundant reformulations; each such reformulation results in an unneeded query on a peer, significantly degrading performance. Finally, in many cases the algorithm yields inefficient reformulations, i.e., queries on peers that must be heavily optimized before they can be executed. As our experiments show, these inefficiencies are a major impediment to efficient query processing in PDMS.

Our contributions: This paper describes several methods for optimizing query reformulation in a PDMS, and evaluates their impact with a detailed set of experiments on a fully implemented PDMS, Piazza. In Piazza, data is modeled in XML and peers represent their schemas in XML Schema. To enable our optimizations we developed a set of practical algorithms for containment and minimization of XML queries, which are of independent interest and applicable elsewhere. In particular, the optimizations we describe are the following:

- **Pruning and minimization:** we describe algorithms for pruning redundant reformulation nodes and for minimizing reformulations, and show experimentally that it is critical to perform these optimizations in order for reformulation to scale to large PDMS. To support pruning and minimization we describe the first practical algorithms for containment of XML queries with nesting and for minimization of such queries.
- **Search strategies:** reformulation can be viewed as a search through a space of reformulations. We study the effects of the search strategy on the reformulation time. In particular, we show that the choice of search strategy is especially important when we want to start execution as soon as possible by pipelining reformulation and query execution. We show that a variation on best-first search ends up exploring a smaller space and being most appropriate for pipelining.
- **Pre-computing semantic paths:** we show that pre-computing certain paths in the network can offer many benefits, and examine the tradeoffs involved. To support pre-computation we describe a practical algorithm for composing mappings between XML peers in a PDMS.

While the optimization techniques we describe are, at a high-level, intuitive, the main contributions of the paper are the techniques we developed to apply the optimizations to our context, and the experimental study of their practical impact. All put together, our optimizations speed up reformulation by up to several orders of magnitude, thereby enabling efficient reformulations on significantly sized PDMS.

The paper is organized as follows. Section 2 defines the reformulation problem and describes the opportunities for optimization. Section 3 describes the novel algorithms that are needed to support the optimizations. Section 4 describes the optimization methods we propose and provides a detailed experimental evaluation of their utility. Section 5 discusses related work, and Section 6 concludes.

2. QUERY REFORMULATION IN PDMS

This section describes the challenges raised by query reformulation in a PDMS and the different optimizations that may be considered. We begin with a brief overview of PDMS and of query reformulation.

2.1 Data Sharing with a PDMS

A PDMS consists of a network of nodes referred to as *peers*. Peers can be serve one or more of the following roles: servers of data, mediators translating between schemas of other peers, and clients posing queries. We denote the set of peers by $\mathcal{P} = \{P_1, \ldots, P_n\}$, and we use P_i to refer both to the peer and to its schema. Piazza is based on an XML data model, and therefore their peer schemas are described in XML Schema.

Peer mappings: Peers in a PDMS are linked through *peer mappings*. Peer mappings describe the semantic relationship between the schemas of pairs (or small sets of) peers. Given a query over a peer P_i, the system will use the peer mappings to *reformulate* the query over

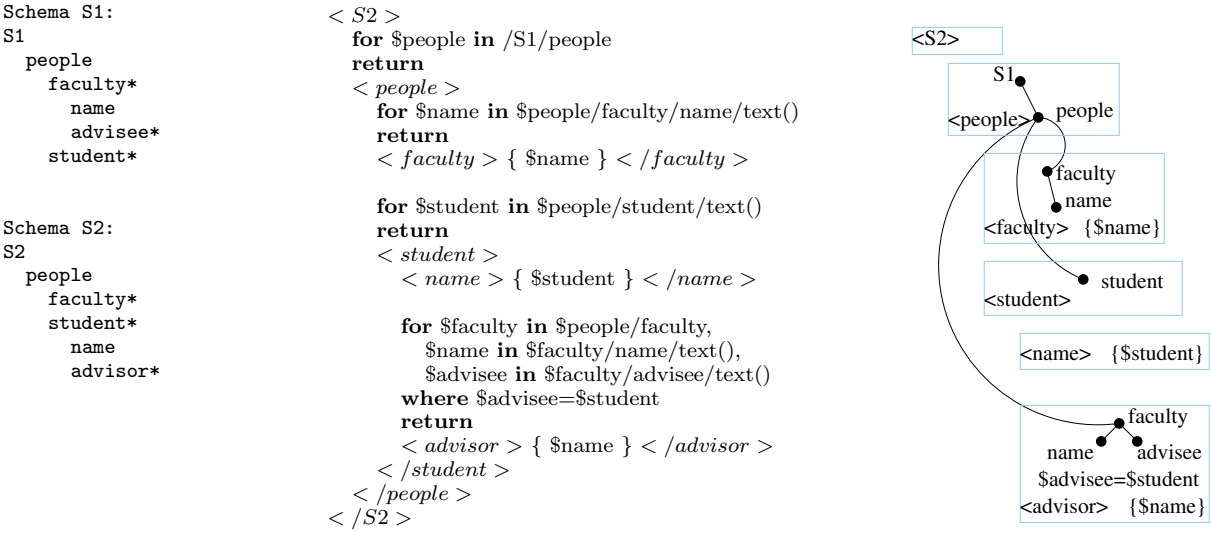

Figure 2: An example pair of peer schemas (left), a Piazza mapping between the schemas expressed as an XML query (middle) and its query block representation (right).

the schemas of the neighboring peers. Typically, when a peer joins a PDMS, it will supply mappings, as is most convenient, to a small number of peers. This paper is *not* concerned with the problem of generating the peer mappings. We refer the reader to [28, 11, 12] for semi-automatic schema mapping techniques, which directly apply our context.

Peer mappings in Piazza are described as query expressions using a subset of XQuery. We describe the details of the mapping language in Section 3. From the perspective of the reformulation algorithms, the important aspect of peer mappings is that they are *directional*: a mapping defines how to construct a fragment of a *target* schema P_i from data in a *source* schema P_j, or set of source schemas. Mappings that have a single source schema are called *pairwise* mappings, and others are called *join mappings*. We denote a pairwise peer mapping where P_i is the target and P_j is the source, by $M_{i,j}$. (A similar notation can be given for non-pairwise mappings, but don't require it for our discussion).

To describe the data stored at the peers, we use a formalism that is small variation on peer mappings. These descriptions are not germane to our discussion. However, it is important to note that a peer's schema can be broader that just what is needed to model the data it actually stores.

EXAMPLE 2.1. *Figure 2 shows an example of a* Piazza *mapping. The mapping defines the relationship between the schema of* S2 *and the schema of* S1*; the two schemas differ in how they represent the advisor/advisee information. In our figures, schemas are represented using a format in which indentation indicates nesting and a * suffix indicates an arbitrary number of occurrences of the subelement.* □

Since PDMSs are a mechanism for decentralized sharing of data, mappings are not controlled in any central fashion. The only assumption we can make about them is that any peer that joins the system provides some mappings (thereby defining its neighbors). In particular, the system may contain mappings in both directions between a pair of peers (i.e., both $M_{i,j}$ and $M_{j,i}$). Clearly, such decentralization also raises questions regarding the consistency of mappings (as individuals and as a set), and general issues of trust. We discuss these issues briefly in Section 6.

Single-step reformulation: Suppose a query Q is posed over the peer P_1. If P_1 contains its own data, then the PDMS first retrieves the answers to Q based on P_1's data. Then, Q is reformulated on P_1's neighbors and appropriate queries are posed to them, and the process continues recursively. For each individual reformulation step, the reformulation algorithm uses one of two techniques: *unfolding* or *rewriting*, depending on the directionality of the mapping available. Suppose P_2 is a neighbor of P_1. If there is a mapping $M_{1,2}$, (i.e., defining P_1 as a query over P_2), then reformulation amounts to unfolding (which can be done using techniques described in [16]). Mapping unfolding is analogous to view unfolding in traditional database systems, and to query reformulation in GAV-based data integration systems [18]. If the mapping is in the other direction, i.e., $M_{2,1}$, then Q needs to be *rewritten* using $M_{2,1}$, which is similar to rewriting a query using a (materialized) view [21] and to query reformulation in LAV-based data integration systems.

In our discussion we treat the actual reformulation algorithm as a black box. The only property we require from the algorithm is that it be sound, i.e., ultimately lead to only *certain answers* [2] to the query. The theoretical properties of query unfolding for our formalism are well understood. However, for the case of rewriting, the theoretical properties of algorithms for rewriting XML queries with nesting are still a subject of research, and therefore so is the question of returning *all* certain answers. However, there are several algorithms that capture practical cases well [10, 19].

Multi-step reformulation: A PDMS answers queries by chaining individual reformulation steps. In this way, a PDMS can follow arbitrarily long semantic paths and retrieve answers from peers not directly connected to

the query peer. Given a single-step reformulation algorithm, a template algorithm for query answering in a PDMS could be implemented iteratively as follows. Suppose that the PDMS includes only pairwise mappings. At every point, we maintain a tree of *goals* $\mathcal{G}$, each of which is a query on a particular peer. Initially, $\mathcal{G}$ includes a single goal with the original query and peer, i.e., (Q,P). At each iteration, we choose one of the leaf goals $(Q', P') \in \mathcal{G}$. First, if P' contains data, we pose the query Q' on the peer P' and add the set of answers to the set of answers to Q. Second, we reformulate Q' on all the neighbors of P' and add the newly created goals to $\mathcal{G}$.

The reason we follow all possible semantic paths are twofold. First, as in data integration systems, since peers rarely contain *complete* data w.r.t. their schema, two peers with overlapping domains may contain different (or overlapping) sets of data instances. Second, different paths to a peer may produce different reformulations, because the reformulation depends on the intermediate nodes on the path. At the extreme, one path may produce an empty reformulation, while another will not.

In [20] it is shown that if a PDMS contains certain kinds of cycles, then query answering, and our procedure in particular, may not terminate. In Piazza we consider all paths, except that we only follow a cycle at most once. With this termination condition we are guaranteed to find all the answers to a query when possible according to the conditions in [20], but still obtain answers in other cases as well.

When the PDMS includes join mappings, i.e., a mapping that defines a target as a join over multiple source schemas, the procedure is conceptually similar but more elaborate. Instead of a simple tree of goals, we build an AND-OR tree. Reformulation using a join mapping creates an AND node in the tree whose children are the queries on the source peers of the mapping. Answers to AND nodes in the tree are obtained by performing the appropriate join on answers of its children.

As experienced in our initial implementation of Piazza, the above algorithm has several obvious inefficiencies. The algorithm may follow redundant paths (which later result in redundant queries to the peers), or paths that can be pruned early on. Second, in many cases the algorithm yields large reformulations, which need to be heavily optimized before they can be executed.

We now describe the reformulation optimizations we consider in this paper. We assume that query reformulation is performed at the query peer and that the entire PDMS catalog is available to every peer (the catalog can still be distributed, e.g., using a distributed hash table).

2.2 Optimization Opportunities

In order for query processing in a PDMS to be viable, we need to optimize the above procedure significantly. We now outline several optimization opportunities we consider in this paper.

Optimization goals: depending on the specific environment, different metrics can be used to optimize query reformulation in a PDMS. The optimizations we describe below are common to all of them. One possible optimization goal is to reduce the execution time of queries on peers. This metric requires computing the most efficient set of queries to be executed on the peers. A primary example of an inefficiency that we would like to avoid is executing a redundant query, i.e., a query that returns a subset of the results of a previously executed query. Alternatively, in interactive applications the primary goal is to minimize *response time* (time to first answers). Here too, reducing redundancy is a concern since redundant reformulations are more likely to return no answers. Finally, the optimizations we describe here are also crucial for off-line analyses of networks of mappings, where the goal is to remove redundancies in the mappings. Section 4 shows experimentally the impact of each of the following optimizations.

2.2.1 Pruning reformulation goals

A PDMS is likely to include multiple semantic paths between most pairs of peers. On the one hand, this is an advantage because different paths can yield flow of different fragments of data. In particular, there may be cases when a path between a pair of peers leads to an empty reformulation, and then an alternate path may be more successful. On the other hand, multiple paths may lead redundant reformulations, and hence redundant queries on the peers.

Pruning is a common technique used to avoid wasting work in a search algorithm. In our case, pruning involves checking *query containment* between a previously obtained reformulation and a new one. In Section 3.2 we describe a query containment algorithm that enables us to perform pruning.

2.2.2 Minimizing reformulations

As we follow longer paths in the PDMS, the repetitive application of query unfolding and rewriting results in very large reformulations containing redundant subexpressions. (A similar observation regarding query unfolding was made in [16]). In Section 3.3 we describe a minimization algorithm for XML queries.

2.2.3 Pre-computation of Semantic Paths

Another approach to improve efficiency of query reformulation in a PDMS is to pre-compute some (or all) semantic paths. To pre-compute a semantic path we need to *compose* the individual peer mappings along the path [25, 15]. The composed mappings offer two types of benefits. First, some semantic paths can be eliminated a priori, independent of the input query if the composition is found to yield an empty reformulation. Second, the composed mappings can be pre-optimized to remove redundancies, therefore yielding better reformulations at query time. Composition needs can be done judiciously to exploit the most commonly used paths.

Composition of peer mappings was first considered in [25] for mappings between relational schemas. (Note that mapping composition is quite a bit more complex than query composition). [25] shows that composition can be quite expensive, and even lead to mappings with unbounded size. In Section 3.4 we describe a practical mapping composition algorithm for the XML context.

2.2.4 Search Strategies

The reformulation process can be viewed as a search in the space of reformulations. Unlike typical search problems, where we need to find a *single* goal state, here we need to find *all* possible reformulations. However, the search strategy can still make a difference. If a reformulation Q is ultimately going to be pruned, we would like to prune it before we expand any (or much) of its subtree. This is particularly important if we are pipelining reformulation and execution, i.e., we execute reformulations on peers as they are generated. Here, producing a reformulation Q before a different reformulation that subsumes it will result in a wasted query to the peer.

3. THE CORE ALGORITHMS

We now describe the core algorithms underlying our optimizations, including the first algorithm for containment of XML queries with nesting, an algorithm for query minimization and for composition of peer mappings. Although they are developed in the context of PDMS optimization, they are also of independent interest. We begin by describing the specific query and peer-mapping language we use in Piazza.

3.1 Query and Mapping Languages

In our discussion, queries to the PDMS are posed in a fragment of XQuery. Our mapping language has a different syntax, but peer mappings can be viewed as queries in the same language fragment. Recall that a peer mapping defines a target schema in terms of a source schema (in the spirit of DTD-to-DTD mappings in [8]).

In what follows we describe the exact limitations on queries in our language fragment. An example peer mapping is shown in the middle column of Figure 2. Our queries include the FOR, WHERE and RETURN clauses of XQuery.

- Path expressions in the query can include only the *child* and *descendant* axes and alternation (|).
- Nested blocks are supported. Note that while user queries may often consist of a single block, mappings are typically nested queries because the target schema is nested.
- A query block may contain equality predicates in one of the two forms: $\$var = literal$ or $\$var1 = \$var2$. In both cases, the variables must be bound to *text* values. We consider only $=$ and $\neq$.
- A leaf query block can "return" a value that can be either a variable or a literal.

Our fragment of XQuery ignores order between sibling XML elements. We also do not support returning mixed content. Note that since we assume knowledge of the schemas of peers, then the fact that we do not support node comparisons or returning of elements is not a real restriction, because we can always unfold them according to the schemas (assuming the schema is not recursive). Throughout our discussion we assume set semantics for queries, rather than the sequence semantics of the XQuery.

When appropriate, we use the notation $A \subseteq Q(B)$ (or $A \supseteq Q(B)$) to denote mappings. Here, B denotes the source schema and A denotes the target schema. The left-hand side need not mention the entire schema of A, so it can be a projection on A's schema. Hence, in comparison to the GLAV formalism for describing mappings among relational peers [23], our formalism is restricted so that the left-hand side can only contain projections. The $\subseteq$ enables describing the typical case where a source does not contain *all* of the data in a particular domain (as in the open-world assumption [2]). Complete sources can be described with two descriptions.

Tree structure in queries: for analysis purposes, it is convenient to identify two tree structures in our queries, the *head tree* and the *path tree*. The head tree is defined as follows. Nodes in the tree correspond to query blocks, and the labels on the the nodes correspond to the tag returned by that block. The example mapping in Figure 2 has the following head tree:

```
S2
  people
    faculty ($name)
    student
      name ($student)
      advisor ($name)
```

Here indentation corresponds to the block-subblock relationship in the query.

The path tree (which can actually be a forest) is defined as follows. Each edge in a tree pattern corresponds to a path expression in the query. The source node of an edge corresponds to the base variable of the path expression, and the target node corresponds to the variable bound by the expression.

The right column of Figure 2 illustrates both trees. A box in the graph corresponds to a query block. The box label, such as $< people >$ corresponds to the tag of the XML element constructed by the query block. The XPath expressions in each block are shown as connected tree patterns. The value returned by the block, if any, is shown in braces, e.g., $\{\$name\}$.

3.2 XML Query Containment

Checking containment of XML queries is fundamental to several of our optimizations, including pruning nodes, eliminating redundant reformulations, and minimizing reformulated queries. While query containment of XPath expressions has been well studied [26, 9], containment of XML queries with nesting has not been addressed. The two problems are fundamentally different because in the former case the query returns a set of nodes, while in the latter it returns a nested structure. In what follows we describe an algorithm for containment of XML queries with nesting.

3.2.1 Definition of XML Query Containment

In order to define containment of XML queries, we first need to define containment of XML instances. An

XML instance can be viewed as a node-labeled tree, which is special case of complex objects [3]. Therefore, we can specialize the definition of containment of complex objects [24] to node-labeled trees. The definition is based on homomorphisms between trees:

DEFINITION 3.1. *(Tree homomorphism.) Given a pair of node-labeled trees T_1 and T_2, a mapping ψ from the nodes of T_1 to the nodes of T_2 is a tree homomorphism if the following conditions hold:*

- *ψ maps the root of T_1 to the root of T_2,*
- *if node n_2 is a child of node n_1 in T_1, then $\psi(n_2)$ is a child of $\psi(n_1)$ in T_2, and*
- *for every node $n \in T_1$, the label of n in T_1 is the same as the label of $\psi(n)$ in T_2.*

DEFINITION 3.2. *(XML instance containment.) An XML instance D_1 is contained in an XML instance D_2, denoted as $D_1 \sqsubseteq D_2$, if there exists a tree homomorphism from the node-labeled tree corresponding to D_1 to that of D_2.* □

DEFINITION 3.3. *(XML query containment.) An XML Query Q_1 is contained in an XML query Q_2, denoted $Q_1 \sqsubseteq Q_2$, if for any XML instance D, $Q_1(D) \sqsubseteq Q_2(D)$.* □

3.2.2 Algorithm for XML Query Containment

We now describe an algorithm for XML query containment. We note that in concurrent work [13], we have studied the complexity of query containment for XML queries with nesting, and described a sound and complete containment algorithm. However, the analysis in [13] has shown that the complexity of query containment, in the worst case, can be super-exponential. The algorithm we describe here is incomplete in general, but is useful for practical applications. As we show, under certain conditions, our algorithm is complete. We comment on the sources of incompleteness of our algorithm in Section 4. To simplify our discussion, we do not allow sibling query blocks to have the same tag. Such blocks introduce a form of union, which requires additional bookkeeping.

To check whether the query Q_1 is contained in the query Q_2 our algorithm finds a *containment mapping* from Q_2 to Q_1. Specifically, in our context, a containment mapping consists of a pair of embeddings: a *query-head embedding* from Q_1 to Q_2 and a *query-body embedding* that extends the query-head embedding from Q_2 to Q_1. We now explain each of these embeddings.

Query-head embedding: A query-head embedding, $E_{head}(Q_1, Q_2)$, from a query Q_1 into a query Q_2 is a tree homomorphism from the head tree of Q_1 to the head tree of Q_2.

Figure 3 shows an example of a query-head embedding from Q_1 to Q_2. Note that in addition to mapping query blocks, the query-head embedding establishes a mapping from the variables returned by Q_1 to a subset of those returned by Q_2. In this example, `Q1.$student` is mapped to `Q2.$student` and `Q1.$name` is mapped to `Q2.$name`. Note that a query-head embedding is not onto, i.e., Q_2 may output some subelements that Q_1 does not. For example, the `faculty` block of Q_2 in the example is not covered by the embedding.

Query-body embedding: A query-body embedding, $E_{body}(Q_2, Q_1)$, from a query Q_2 into a query Q_1 is a tree homomorphism from the path tree of Q_2 to the path-tree of Q_1. We say that a query-body embedding $E_{body}(Q_2, Q_1)$ *extends* a query-head embedding $E_{head}(Q_1, Q_2)$ if the following condition holds for every node n_2 in the path-tree pattern of Q_2: suppose $E_{body}(Q_2, Q_1)$ maps the node n_2 to the node n_1 in the path-tree of Q_1, and $E_{head}(Q_1, Q_2)$ maps the query block of n_1 to the block B in Q_2. Then B is either the same as the block of n_2 or one of its descendents.

Figure 4 shows an example of a query-body embedding from query Q_2 to query Q_1 that extends the query-head embedding from Q_1 to Q_2 shown in Figure 3. The figure illustrates how the tree patterns of Q_2 can be mapped "higher-up" in Q_1. For example, the node `faculty` that belongs to the `advisor` block of Q_2 is mapped into the `faculty` node in the `people` block of Q_1. Based on these embeddings, we can now define containment mappings.

DEFINITION 3.4. *(Containment mapping.) A containment mapping from an XML query Q_2 into a query Q_1 consists of a query-head embedding $E_{head}(Q_1, Q_2)$ from Q_1 to Q_2, a query-body embedding $E_{body}(Q_2, Q_1)$ from Q_2 to Q_1 that extends $E_{head}(Q_1, Q_2)$, where the following conditions hold:*

- *For every query block B_1 of Q_1 and its image B_2 under E_{head}, the predicates of B_1 and its ancestor blocks imply those of B_2.*
- *Let v_1 be either a variable or constant returned by a query block B_1 of Q_1, and let v_2 be the variable or constant returned by the B_2, the image of B_1 under E_{head}. Let $E_{head}(v_1)$ be the image of v_1 under E_{head}. Then, either $E_{head}(v_1) = v_2$, or the predicates in B_1 and its ancestor blocks entail that $v_1 = v_3$, such that $E_{head}(v_3) = v_2$.*

To test containment, our algorithm searches for a containment mapping. The following theorem shows that our containment algorithm is sound, and in some cases, complete.

THEOREM 3.1. *Let Q_1 and Q_2 be a pair of queries in the fragment of XQuery as described in Section 3.1. Assume the queries do not contain same-tag sibling blocks. If there exists a containment mapping from Q_2 into Q_1 then $Q_1 \subseteq Q_2$. Moreover, if the queries are not nested, i.e., each query consists of a single block, then $Q_1 \subseteq Q_2$ implies that there is a containment mapping from Q_2 into Q_1.* □

It has been observed that a containment mapping is not a necessary condition for containment of XPath queries only if both // and * are allowed. The above theorem entails that with nesting, even if both // and * are disallowed, a containment mapping may still be

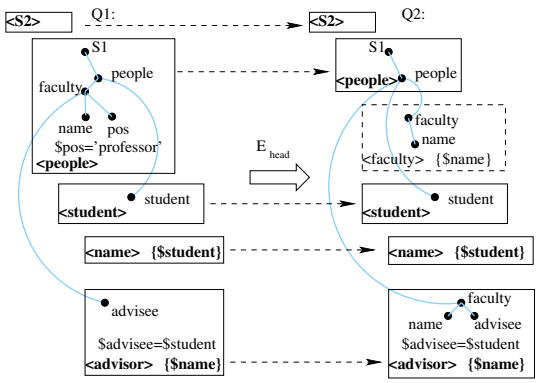

Figure 3: A query head embedding from query Q_1 to Q_2. The Q_1 blocks and the matching Q_2 blocks are highlighted. The matching return variables are shown in bold.

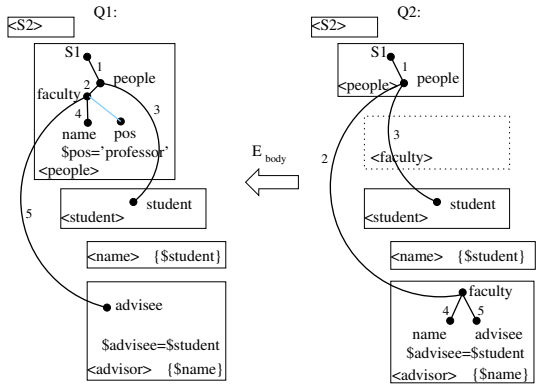

Figure 4: A query body embedding from query Q_2 to Q_1. The Q_2 tree patterns and the matching Q_1 tree patterns are highlighted; matching edges are labeled with numbers.

unnecessary for XQuery containment. A similar result is shown in [24] for queries over complex objects.

Based on the definition of a containment mapping, one can design an algorithm for testing XML query containment. The algorithm works by considering all possible head embeddings between a given pair of queries. Then, for each head embedding it checks if there is a possible query body embedding that is valid. If Q_1 and Q_2 do not contain comparison predicates ($=$ and $\neq$), then the containment algorithm runs in polynomial time. In general, in the worst case the algorithm has exponential complexity. As we see in the next section, the algorithm is efficient in practice.

3.3 XML Query Minimization

Following semantic paths in a PDMS can result in queries that have many redundant fragments. The reason is, as has already been observed in [16], that unfolding often results in redundant queries. Intuitively, unfolding boils down to copying fragments of the mapping into the query. Some fragments turn out to be unnecessary, and others get copied multiple times also leading to redundancy. In order to avoid a blow up of the reformulated queries, it is important to *minimize* these queries after every unfolding.

Like query containment, the problem of query minimization has been studied in the literature for XPath queries [4], but not for queries with nesting. The problem of XML query minimization can be defined formally as follows. Given an XML query Q, find a query Q' that is equivalent to Q but that is simpler, i.e., Q' should have fewer tree pattern nodes. In [26] it is shown that minimization of XPath queries when both // and * are allowed is NP-hard. If either // or * is not allowed then minimization can be done in polynomial time.

We can show, by reduction from conjunctive query minimization, that XML query minimization is also coNP-hard, even if XPath expressions are not allowed to contain // or *. In fact, the problem can be shown coNP-hard even if predicates are not allowed but nested blocks are. Moreover, it can be shown that the minimization problem is at least as hard as the containment problem similar to the case of XPath queries [26, 17].

Given the containment algorithm described above, a query minimization algorithm can be implemented as follows. Given a query Q, our algorithm establishes a *maximal* containment mapping $M_{self}(Q)$ from Q to itself. The mapping must be maximal in terms of the number of path-tree nodes that are not mapped to themselves but are mapped to other nodes with the same tag. Once such a containment mapping is found, the query can be minimized by removing the nodes that are not mapped into themselves. The soundness of this algorithm follows straightforwardly from the soundness of the containment algorithm.

3.4 Mapping Composition

As described earlier, one of the possible optimizations in a PDMS is to pre-compute certain paths. The technical problem that needs to be addressed in order to pre-compute paths is *mapping composition*: how do we replace a "chain" of two or more mappings with an equivalent single mapping.

Mapping composition was studied for the relational case in [25]. There it was shown that the exact composition of two mappings may be infinite. In [25] mappings were of the form $Q_1(A) \subseteq Q_2(B)$, where Q_1 and Q_2 are conjunctive queries over the schemas A and B, respectively. In Piazza, the queries are over XML data and the left hand-side is effectively constrained to be a simple projection over the target schema, but even with that restriction we can get into the same complications described in [25].

The composition algorithm we describe here builds on the query reformulation algorithm (in particular, the algorithm for rewriting a query using a mapping). Hence, our composition algorithm is sound, and it becomes more complete as the underlying rewriting algorithm does. This is important because we can benefit directly from improvements to the reformulation algorithm. For example, it may be easier to incorporate support of integrity constraints directly into a reformulation algorithm than into a mapping composition algorithm.

3.4.1 Mapping Composition through Inversion

Given peers A, B and C, and mappings M_1 between

```
for $S1 in /S1 return
< S1 > {
  for $people in $S1/people return
  < people > {
    for $faculty in $people/faculty return
    < faculty > {
      for $name in $faculty/name/text() return
        < name > {$name} < /name >
      for $advisee in $faculty/advisee/text() return
        < advisee > {$advisee} < /advisee >
    }< /faculty >
    for $student in $people/student/text() return
      < student > {$student} < /student >
  }< /people >
}< /S1 >
```

Figure 5: The identity mapping for schema S1.

A and B and M_2 between B and C, our goal is to find a direct mapping between A and C. Note that since peer mappings are directional, composition can result in two mappings, one in each direction. We consider mappings in a single direction, but the treatment for the converse case is similar.

The cases in which composition is possible fall into two categories. In the first case, the input mappings are of the form:

$$M_1 : A \supseteq Q_1(B) \text{ and } M_2 : B \supseteq Q_2(C).$$

This case is straightforward, as it amounts to query composition, and hence the composed mapping is $A \supseteq Q_1 \circ Q_2(C)$.

In the second case the mappings are of the form

$$M_1 : B \subseteq Q_1(A) \text{ and } M_2 : B \supseteq Q_2(C).$$

To compose these two mappings we first compute the *inverse* of M_1, M_1^{-1}. An inverse mapping has the property that if reformulating a query using the original mapping requires unfolding, then reformulating the same query given an inverse mapping would require rewriting, and vice versa (similar to inverse rules in the relational case [14]). The resulting composition is then obtained by query composition: $A \supseteq M_1^{-1} \circ Q_2(C)$.

Mapping inversion: Our method for inverting mappings works in the same spirit as constructing inverse rules for relational views [14]. Consider the following Piazza mapping: $M : A \subseteq Q(B)$. We proceed in two steps:

- Let I_B be the *identity* query over B, i.e., a query that maps B into itself in a nested fashion on a per-element basis. Figure 5 shows the identity mapping for schema S1. Note that it is always possible to write an identity mapping.
- Rewrite the query I_B using the view Q. Denote the result of such rewriting by $[I_B]_Q(A)$. Note that the query $[I_B]_Q(A)$ "generates" B as a query over A, i.e., it inverts the query Q in the mapping M. The inverse mapping of M is $B \supseteq [I_B]_Q(A)$. Figure 6 shows the inverse mapping for the mapping in Figure 2.

Note that the inverse mapping could be improved if we made use of integrity constraints. Specifically, the

```
for $people in /S2/people return
< S1 > {
  < people > {

    for $faculty in $people/faculty return
    < faculty > {
      for $name in $faculty/text() return
        < name > {$name} < /name >
    }< /faculty >

    for $student in $people/student,
      $advisor in $student/advisor return
    < faculty > {
      for $name in $advisor/name/text() return
        < name >{$name}< /name >
      for $advisee in $student/name/text() return
        < advisee >{$advisee}< /name >
    } < /faculty >

    for $student in $people/student/name/text() return
      < student >{$student}< /student >
  } < /people >
}< /S1 >
```

Figure 6: The inverse of the mapping M in Figure 2 obtained through rewriting the identity mapping for S1 using M.

the set of faculty elements in $S1$ is obtained from the corresponding set in $S2$ and the set of advisor elements in $S2$, which can be repetitive. Second, every faculty element in $S1$ gets only one advisee sub-element rather than having all of its advisees grouped under a single advisor element.

To summarize this section, we described a set of practical algorithms for reasoning about XML queries. Like similar algorithms for relational query languages, these algorithms are applicable even beyond our particular context. We can now use them to develop optimizations for PDMS reformulation.

4. OPTIMIZATION TECHNIQUES

We now describe the methods for optimizing query reformulation in a PDMS. For each method we describe experimental results that evaluate their impact in practice. We start by describing our experimental setup.

4.1 Experimental Setup

We experimented with two real-world data sets. The first, which we refer to as the XML.org data set, is based on schemas that were independently contributed to a repository at www.xml.org. We began with 10 of the schemas in that site, and by adding structure variations on them, we created schemas for 27 peers. The schemas are in the domain of customer orders and typically represent customers, vendors, orders, and products.

The second data set, DB-Research, is based on data available on web sites concerning research in our field. We created schemas corresponding to the structure and terminology of 19 such web sites (such as DBLP, CiteSeer, ACM Digital Library, and a few university sites). The sites usually represent researchers, projects, publications, and related material. The schemas in DB-Research tend to be more diverse in their content than the schemas in the XML.org whereas the schemas in the latter data set have more diverse structures.

For each of the data sets we created the mappings between the most similar peers, typically resulting in 4 mappings for each peer. The scenarios we ran were based on randomly sampling from the collection of mappings. In the scenarios we varied the average *rank* of each peer, which refers to the average number of mappings per peer (and hence, a higher rank corresponds to a more interconnected network).

In order to experiment with larger PDMSs we replicated the set of peers in each domain and connected replicas with a small set of mappings. Note that the structure of each replica is different, since we sampled the set of mappings for each replica separately. We experimented with PDMSs having zero, one and two replicas (which we refer to as *scale factor* of 1, 2 and 3, respectively). Hence, in the XML.org domain we had PDMS of up to 81 peers, and in the DB-Research domain up to 57 peers.

We tested each data instance on a set of five "meaningful" queries. In the XML.org case, the queries are relatively simple pattern matching queries (e.g., find suppliers of a particular product, or find customers who are also suppliers). In the DB-Research case, since the schemas are more complex, our queries also included joins (e.g., find PC members who had a paper in that same conference). Join queries typically result in a much larger number of reformulations because we obtain combinations of reformulated subqueries.

The results we show were obtained by executing 20 runs on each topology. We generated 20 random topologies for each set of input parameters (scale factor and rank). We observed that our results were quite consistent within such families of topologies. The results presented in the rest of the section are averaged across all topologies for given scale and rank.

The query reformulator of Piazza is implemented in Java and includes 40,000 lines of code. The system was tested on a Linux Pentium 4 PC running at 2.8GHz with 512M of memory. (We allocated 400MB to the Java interpreter.) The actual evaluation of the queries is done by a data integration engine developed in our group, but the focus of our experiments is on speeding up the reformulation process.

4.2 Pruning and Minimization

The two most important optimizations we consider are pruning redundant reformulation nodes and minimizing the reformulations as they are constructed.

The process of query reformulation builds an AND-OR tree of nodes, where each node is a query on a particular peer. The answers of an AND node are obtained by joining the answers its children, while the answers of an OR node are the union of the answers of its children. One of the first observations we made is that it is crucial to minimize the resulting reformulations as we expand the AND-OR tree. Without minimization, every reformulation step that performs query unfolding can lead to a blowup in the size of the resulting reformulation, and after a few steps the reformulations may become unwieldy. The query minimization algorithm we described in Section 3.3 enables us to tame the growth of the reformulations.

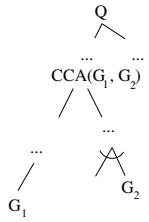

Figure 9: An example of pruning. Goal g_1 subsumes g_2 only if all nodes between g_1 and the closest common ancestor of g_1 and g_2 are OR nodes, and the query in g_1 contains the query in g_2.

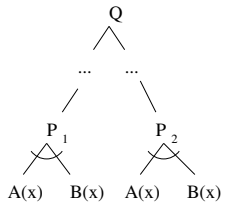

Figure 10: An example when pruning doesn't work. One of the two pairs of goals can be pruned but none of the goals can be pruned on its own.

Pruning is a very effective technique to avoid building unnecessary parts of the AND-OR tree. Informally, pruning removes nodes that are guaranteed to be subsumed by other nodes. Formally, suppose g_1 and g_2 are two nodes in the tree, of the form (q_1, p), and (q_2, p), i.e., they both represent queries on the same peer, p. If the following conditions hold, then the node g_2 can be pruned.

- the query q_2 is contained in q_1, and
- all the nodes between g_1 and the closest common ancestor of g_1 and g_2 are OR nodes.

The second condition guarantees that g_1 is an alternative to g_2 in the tree, and therefore g_2 does not need to be considered further. This situation is depicted in Figure 9. Note that g_2 *may* have siblings, but only if the AND node happens to be performing an intersection of its children.

Given the containment algorithm described in Section 3.2, we can implement pruning. When the tree is extended with a new goal node, we check whether (1) the new goal is subsumed by an existing goal, or (2) the new goal subsumes an existing goal.

Experiments: The effect pruning and minimization on reformulation time is drastic. In fact, when either pruning or minimization were not employed, the system would often run out of memory for the data sets with scale factor of 2 or 3. In other cases, the effect of pruning and minimization was similar to that shown in Figure 7. A relatively small PDMS (DB-Research of scale factor=1) was used for the experiments in this Figure. As we can see, minimization leads to a speedup of up to a factor of 3; pruning leads to a speedup of a factor of 10. The combined speed up factor is almost 30. We conclude that pruning and minimization are crucial for reformulation to scale up, and hence all of our subsequent experiments employ both of them.

Incompleteness of containment checking: As described in Section 3.2, our containment algorithm is not complete for arbitrarily nested queries. Incompleteness means that in some cases the algorithm will say that Q_1

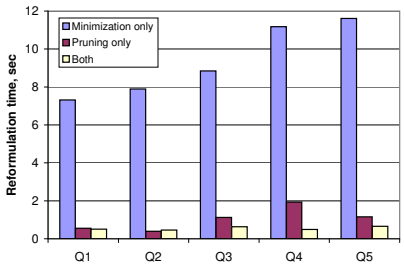

Figure 7: The effect of minimization and pruning on reformulation times. (DB-Research dataset, scale=1, rank=3).

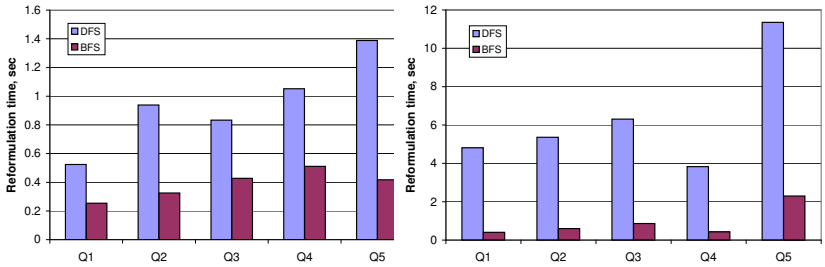

Figure 8: Reformulation times of DFS vs BFS on DB-Research (left) and XML.org (right), scale=2, rank=3. The graph shows that BFS is a more efficient reformulation strategy.

is not contained in Q_2, even though it is, thereby missing opportunities for pruning and minimization (but not leading to incorrectness!). In contrast, a complete algorithm for query containment, recently described in [13], would be impractical. Since our algorithm is complete for unnested queries, this is not an issue in pruning and minimization as long as the user query is not nested (because the resulting reformulations are always unnested), but incompleteness will arise in pre-composition. Clearly, the experiments show that the algorithms are capturing common cases of containment, and therefore we are witnessing the savings. Furthermore, the experience we gleaned from the experiments shows that the most significant source of incompleteness is that we do not exploit integrity constraints (e.g., keys) in our algorithms. We plan to incorporate keys in our next version and expect that it will make the minimization algorithm more effective. □

Finally, we note that when the second condition for pruning is not satisfied, i.e., g_1 and g_2 are related through AND nodes, some optimizations are still possible. Consider the situation in Figure 10, where AND nodes are depicted by the arcs across their children. One of the two pairs of $A(x)$ and $B(x)$ subgoals can be pruned but none of the individual subgoals can be pruned because their parents P_1 and P_2 are AND nodes. While it is possible to extend our pruning technique to handle sets of nodes rather than individual goal nodes, it is likely to be very expensive, with relatively little payoff. For similar reasons, we feel that employing techniques such as memoization will not produce significant benefits in our context.

4.3 Search Strategies

When the entire AND-OR tree needs to be constructed and pruning is not applied, the specific search strategy we use to build the tree, i.e., the choice of which reformulation goal to pursue next, is not important. However, the search technique interacts in an interesting way with pruning. Intuitively, if a node is ultimately going to be pruned, then we would like to prune it before any (or much) of its subtree is constructed.

We considered two strategies: depth-first search (DFS) and breadth-first search (BFS). DFS works by expanding goals in longer semantic paths first. We predicted that DFS would be relatively inefficient because reformulation nodes further down the tree are likely to be pruned by nodes closer to the root that would be generated later. BFS, on the other hand, is likely to produce smaller trees because it creates the non-pruned reformulations first.

Figure 8 confirms our prediction. The figure considers the scaled versions of the two original data sets (the differences between the two search strategies are less pronounced on smaller PDMSs). For both datasets, BFS clearly outperforms DFS by a large margin. The sizes of the *pruned* AND-OR trees constructed by the two methods compare similarly. DFS constructs huge rule-goal trees (several thousand nodes) that eventually get pruned by a factor of ten, whereas BFS does not waste as much work.

Pipelining reformulation and execution: The effect of the search strategy is especially important if we want to pipeline the reformulations, i.e., every time a new reformulation is generated, the execution engine sends it to the appropriate peer. Now the cost of executing a reformulation that would have been pruned is much higher, since execution takes much longer and involves consuming resources that may have other costs.

A simple way to reduce the ratio of redundant queries that get executed is to delay query execution until *more* queries have been generated for each peer. Then, if a redundant query is generated early on, the query is delayed and can still be pruned by later queries. The disadvantage of the above approach is that response time may suffer, because we need to wait for several reformulations for a given peer before we query it. Furthermore, with BFS we expect that the queries generated early are less likely to be redundant.

Based on this observation, we implemented a *layered delay* strategy, when the queries corresponding to a layer of an AND-OR tree are delayed until the entire layer is generated. Since the higher layers of a tree are usually much smaller than the lower layers, we expect better response time with this strategy.

Figure 11 considers the naive delay and the layered delay strategies for two of our queries (but the results for the others were similar). It shows the fraction of queries sent to peers that would be been pruned (i.e., a better reformulation was generated at a later time), as we increase the number of reformulations we wait for (e.g., 3 means that we wait for 3 reformulations for a given peer before sending it a single query). We observe that the layered delay strategy results in much fewer

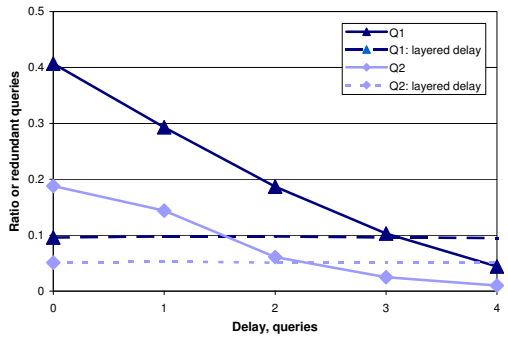

Figure 11: The ratio of redundant queries sent to peers for the *Naive delay* and *Layered delay* strategies. XML.org dataset, scale=2, rank=3. *Layered delay* produces fewer redundant queries and outputs them sooner.

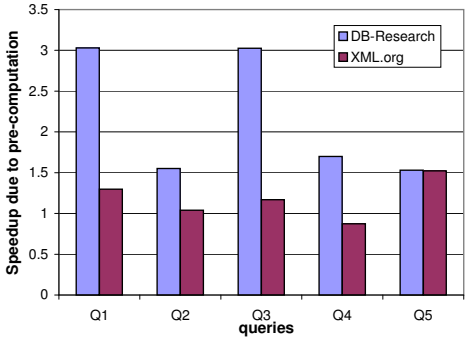

Figure 12: The speed up in reformulation time due to pre-computation (scale=1, rank=3).

redundant queries. In fact, layered delay produces the same ratio of redundant queries as a delay of 2-3 queries, but with the advantage that queries can be sent to peers much earlier.

To study the effect further, we were interested in how the strategies affect the *first* query received by each peer. We observed that with naive delay, up to a quarter of the peers were receiving a redundant query as their first one, while layered delay reduced that ratio to 5-10%.

4.4 Pre-computing Reformulation Paths

Another opportunity for optimization in a PDMS is to pre-compute some or all possible paths between peers, which can be done with the mapping-composition algorithm described in Section 3.4. With pre-computation, given a query Q on p, we only need to reformulate Q using each of the pre-computed mappings. Note that the order in which we apply the mappings makes a difference. Specifically, if we reformulate Q using a reformulation corresponding to a path P in the PDMS, and the result is an empty query, then there is no reason to reformulate Q with the mappings corresponding to paths starting with P. (Recall that a composed mapping may not be empty, but may yield an empty reformulation when used for reformulating a particular query Q). Hence, we apply a reformulation corresponding to a path only after applying all of its prefixes.

The primary advantage of pre-computation is reduced reformulation time due to early pruning of redundant paths. The disadvantage of pre-computing mappings is that the mappings need to be maintained in the face of schema and mapping changes. Fortunately, while such changes will happen quite often in a large scale decentralized system, their frequency is still low in comparison to the time taken to pre-compute the paths. A second disadvantage, at least of our composition algorithm, is that composed mappings are *harder* to minimize. Recall that our composition algorithm begins by reformulating the identity mapping on a peer. That mapping tends to be larger than typical queries, and hence harder to minimize as the composition is built. Here too, more aggressive use of integrity constraints in the minimization will help address this problem.

Figure 12 shows the speed up in reformulation time that we were able to obtain through pre-computation. On the DB-Research domain, pre-computation improves performance by up to a factor of 3. We have observed similar results for the scale factor of 2 and different peer ranks. The results for the XML.org dataset were mixed. While we still obtained savings of up to 50%, one of the queries (Q4) actually takes longer to reformulate with pre-computation, due to the aforementioned minimization challenge.

Figure 12 does not show the time for the pre-computation. While this time can be high (upto a minute in our experiments), it should be noted that pre-computation boils down to reformulation of a complex identity query. As a result, pre-computation can pay off rather quickly for larger queries. Furthermore, our experiments show that many of the benefits of pre-computation can be obtained even by pre-computing only to a certain depth as that is likely to prune many redundant paths early.

5. RELATED WORK

Peer-data management systems have attracted significant attention recently (see the September 2003 issue of the SIGMOD Record which is dedicated to this topic). This paper builds on recent work in the Piazza Project [20]. Initially, in [22] we examined the theoretical properties of query reformulation in a PDMS, but in the relational context. We showed how to build a rule-goal tree for query reformulation, and alluded to a few possible optimizations, but did not consider them in detail or evaluate their impact. In [19] we described an XML version of Piazza, including an algorithm for query reformulation on which we build here. In [10] the authors describe an alternative query reformulation algorithm. In this work, we consider the reformulation algorithm a black box.

The paper [5] describes the Hyperion project that focuses on generating schema mappings and on applying rules (triggers) to propagate data in a PDMS according to mapping expressions and mapping tables. The work [27] proposes PeerDB, a P2P-based system for distributed sharing of relational data. Similar to Piazza, PeerDB does not require a global schema. Unlike Piazza, PeerDB does not use schema mappings for

mediating between peers. Instead, PeerDB employs an Information Retrieval based approach for query reformulation. In their approach, a peer relation (and each of its columns) is associated with a set of keywords. Given a query over a peer schema, PeerDB reformulates the query into other peer schemas by matching the keywords associated with the two schemas. Therefore, PeerDB does not have to follow semantic paths to reach a distant peer. The resulting reformulated queries in PeerDB may not be semantically meaningful, and ultimately the system requires user input to decide which queries are to be executed. Xyleme [8], which stores data from multiple sources in a warehouse, uses a more elaborate approach to discovering and defining mappings than PeerDB, based on path-to-path mappings (rather than tag-to-tag). Piazza, on the other hand, is based on schema-to-schema mappings.

6. CONCLUSIONS

Peer data management systems offer a very attractive architecture for data sharing because they do not require any central management. They represent the next natural step, generalizing current data integration architectures. In order for PDMS to be a viable architecture, a query processor must be able to efficiently reformulate a query posed at a peer on all the relevant peers in the network. We described a set of basic reformulation optimizations, which, in aggregate, enable scaling reformulation to large PDMS. In particular, we described how to prune redundant reformulations, to minimize reformulations, to pre-compose paths and to carefully select the search strategy. All put together, our optimizations enable reformulating queries on a PDMS of size 60-80 peers in under 2 seconds. In support of these optimizations we described a set of algorithms of independent interest for XML query containment, minimization and for mapping composition.

There are several opportunities for future work. As a first step, we believe further performance improvements in reformulation can be obtained by incorporating knowledge of integrity constraints into our algorithms. Second, PDMS raise a more general challenge of managing large networks of peer mappings. As networks of peer mappings grow, there are additional opportunities for analysis: finding contradictory mappings, finding sets of peers that are not well connected by mappings (yet still seem to share data in the same domain), and dealing with mappings of different quality.

Acknowledgements

We want to thank Luna Dong and Zack Ives for many helpful discussions. This work was supported by NSF ITR grant IIS-0205635 and NSF CAREER grant IIS-9985114.

7. REFERENCES

[1] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth. A framework for semantic gossiping. *SIGMOD Record*, 31(4), 2002.

[2] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Proc. of PODS*, pages 254–263, Seattle, WA, 1998.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Weseley, 1995.

[4] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proc. of SIGMOD*, 2001.

[5] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The hyperion project: From data integration to data coordination. *SIGMOD Record*, September 2003.

[6] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[7] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing : A vision. In *Proceedings of the WebDB Workshop*, 2002.

[8] S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale XML repository. In *Proc. of VLDB*, 2001.

[9] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath fragments. In *KRDB*, 2001.

[10] A. Deutsch and V. Tannen. Mars: A system for publishing xml from mixed and redundant storage. In *Proc. of VLDB*, 2003.

[11] H.-H. Do and E. Rahm. COMA - a system for flexible combination of schema matching approaches. In *Proc. of VLDB*, 2002.

[12] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: a machine learning approach. In *Proc. of SIGMOD*, 2001.

[13] X. Dong, A. Halevy, and I. Tatarinov. Containment of nested XML queries. Submitted for publication, 2004.

[14] O. M. Duschka and M. R. Genesereth. Query planning in Infomaster. In *Proc. of the ACM Symposium on Applied Computing*, 1997.

[15] R. Fagin, P. Kolaitis, L. Popa, and W.-C. Tan. Composing schema mappings: Second-order dependencies to the rescue. In *Proc. of PODS*, 2004.

[16] M. Fernandez, W.-C. Tan, and D. Suciu. Silkroute: Trading between relations and xml. In *Proc. of the Int. WWW Conf.*, 1999.

[17] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of xpath queries. In *VLDB*, 2003.

[18] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[19] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *Proc. of the Int. WWW Conf.*, 2003.

[20] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. of ICDE*, 2003.

[21] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4), 2001.

[22] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. of ICDE*, 2003.

[23] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of PODS*, 2002.

[24] A. Y. Levy and D. Suciu. Deciding containment for queries with complex objects and aggregations. In *Proc. of PODS*, Tucson, Arizona., 1997.

[25] J. Madhavan and A. Halevy. Composing mappings among data sources. In *Proc. of VLDB*, 2003.

[26] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *Proc. of PODS*, 2002.

[27] B. Ooi, Y.Shu, and K.-L. Tan. Relational data sharing in peer-based data management systems. *SIGMOD Record*, 23(3), 2003.

[28] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.