University of Pennsylvania
ScholarlyCommons

Database Research Group (CIS) | Department of Computer & Information Science

5-20-2003

Piazza: Data Management Infrastructure for Semantic Web Applications

Alon Y. Halevy
University of Pennsylvania

Zachary G. Ives
University of Pennsylvania, zives@cis.upenn.edu

Peter Mork
University of Pennsylvania

Igor Tatarinov
University of Pennsylvania

Postprint version. Published in *Proceedings of the Twelfth International Conference on World Wide Web (2003)*, 2003, pages 556-567.
Publisher URL: http://doi.acm.org/10.1145/775152.775231

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/db_research/32
For more information, please contact repository@pobox.upenn.edu.

Piazza: Data Management Infrastructure for Semantic Web Applications

Abstract
The Semantic Web envisions a World Wide Web in which data is described with rich semantics and applications can pose complex queries. To this point, researchers have defined new languages for specifying *meanings* for concepts and developed techniques for *reasoning* about them, using RDF as the data model. To flourish, the Semantic Web needs to be able to accommodate the huge amounts of existing data and the applications operating on them. To achieve this, we are faced with two problems. First, most of the world's data is available not in RDF but in XML; XML and the applications consuming it rely not only on the domain structure of the data, but also on its document structure. Hence, to provide interoperability between such sources, we must map between both their domain structures and their document structures. Second, data management practitioners often prefer to exchange data through local point-to-point data translations, rather than mapping to common mediated schemas or ontologies. This paper describes the Piazza system, which addresses these challenges. Piazza offers a language for mediating between data sources on the Semantic Web, which maps both the domain structure and document structure. Piazza also enables interoperation of XML data with RDF data that is accompanied by rich OWL ontologies. Mappings in Piazza are provided at a local scale between small sets of nodes, and our query answering algorithm is able to chain sets mappings together to obtain relevant data from across the Piazza network. We also describe an implemented scenario in Piazza and the lessons we learned from it.

Keywords
Semantic web, peer data management systems, XML

Comments
Postprint version. Published in *Proceedings of the Twelfth International Conference on World Wide Web (2003)*, 2003, pages 556-567.
Publisher URL: http://doi.acm.org/10.1145/775152.775231

This journal article is available at ScholarlyCommons: http://repository.upenn.edu/db_research/32

Piazza: Data Management Infrastructure for Semantic Web Applications

Alon Y. Halevy Zachary G. Ives* Peter Mork Igor Tatarinov
University of Washington
Box 352350
Seattle, WA 98195-2350
{alon,zives,pmork,igor}@cs.washington.edu

ABSTRACT

The Semantic Web envisions a World Wide Web in which data is described with rich semantics and applications can pose complex queries. To this point, researchers have defined new languages for specifying *meanings* for concepts and developed techniques for *reasoning* about them, using RDF as the data model. To flourish, the Semantic Web needs to be able to accommodate the huge amounts of existing data and the applications operating on them. To achieve this, we are faced with two problems. First, most of the world's data is available not in RDF but in XML; XML and the applications consuming it rely not only on the domain structure of the data, but also on its document structure. Hence, to provide interoperability between such sources, we must map between both their domain structures and their document structures. Second, data management practitioners often prefer to exchange data through local point-to-point data translations, rather than mapping to common mediated schemas or ontologies.

This paper describes the Piazza system, which addresses these challenges. Piazza offers a language for mediating between data sources on the Semantic Web, which maps both the domain structure and document structure. Piazza also enables interoperation of XML data with RDF data that is accompanied by rich OWL ontologies. Mappings in Piazza are provided at a local scale between small sets of nodes, and our query answering algorithm is able to chain sets mappings together to obtain relevant data from across the Piazza network. We also describe an implemented scenario in Piazza and the lessons we learned from it.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services–Data sharing; H.2.5 [**Database Management**]: Heterogeneous Databases; H.2.3 [**Database Management**]: Languages—Data description languages (DDL)

General Terms

Algorithms, Management, Languages

Keywords

Semantic web, peer data management systems, XML

1. INTRODUCTION

HTML and the World Wide Web have had amazing impact on the process of distributing human-readable data to even casual computer users. Yet these technologies are actually quite limited in scope: Web data lacks machine-understandable semantics, so it is generally not possible to automatically extract concepts or relationships from this data or to relate items from different sources. The Web community is attempting to address this limitation by designing a Semantic Web [4]. The Semantic Web aims to provide data in a format that embeds semantic information, and then seeks to develop sophisticated query tools to interpret and combine this information. The result should be a much more powerful knowledge-sharing environment than today's Web: instead of posing queries that match text within documents, a user could ask questions that can only be answered via inference or aggregation; data could be automatically translated into the same terminology; information could be easily exchanged between different organizations.

Much of the research focus on the Semantic Web is based on treating the Web as a knowledge base defining meanings and relationships. In particular, researchers have developed knowledge representation languages for representing *meanings* — relating them within custom ontologies for different domains — and *reasoning* about the concepts. Well-known examples include RDF and RDF Schema, as well as languages that build upon these data models: DAML+OIL and OWL, the recent standard emerging from the W3C.

The progress on developing ontologies and representation languages leaves us with two significant problems. The first problem (also noted by [28]) is that there is a wide disconnect between the RDF world and most of today's data providers and applications. RDF represents everything as a set of classes and properties, creating a graph of relationships. As such, RDF is focused on identifying the *domain structure*. In contrast, most existing data sources and applications export their data into XML, which tends to focus less on domain structure and more around important objects or entities. Instead of explicitly spelling out entities and relationships, they often nest information about related entities directly *within* the descriptions of more important objects, and in doing this they sometimes leave the relationship type unspecified. For instance, an XML data source might serialize information about books and authors as a list of book objects, each with an embedded author object. Although book and author are logically two related objects with a particular association (e.g., in RDF, author writes book), applications using this source may know that this *document structure* implicitly represents the logical writes relationship.

The vast majority of data sources (e.g., relational tables, spreadsheets, programming language objects, e-mails, and web logs) use hierarchical structures and references to encode both objects and domain structure-like relationships. Moreover, most application

*Present address: Dept. of Computer and Information Science, University of Pennsylvania, 200 South 33rd St., Philadelphia, PA 19104.

Copyright is held by the author/owner(s).
WWW2003, May 20–24, 2003, Budapest, Hungary.
ACM xxx.

development tools and web services rely on these structures. Clearly, it would be desirable for the Semantic Web to be able to interoperate with existing data sources and consumers — which are likely to persist indefinitely since they serve a real need. From the perspective of building semantic web applications, we need to be able to map not only between different domain structures of two sources, but also between their document structures.

The second challenge we face concerns the scale of ontology and schema mediation on the semantic web. Currently, it is widely believed that there will not exist a single ontology for any particular domain, but rather that there will be a few (possibly overlapping) ones. However, the prevailing culture, at least in the data management industry, entails that the number of ontologies/schemas we will need to mediate among is actually substantially higher. Suppliers of data are not used to mapping their schemas to a select small set of ontologies (or schemas): it is very hard to build a consensus about what terminologies and structures should be used. In fact, it is for this reason that many data warehouse projects tend to fail precisely at the phase of schema design [33]. Interoperability is typically attained in the real world by writing translators (usually with custom code) among small sets of data sources that are closely related and serve similar needs, and then gradually adding new translators to new sources as time progresses. Hence, this practice suggests a practical model for how to develop a large-scale system like the Semantic Web: we need an architecture that enables building a web of data by allowing incremental addition of sources, where each new source maps to whatever sources it deems most convenient — rather than requiring sources to map to a slow-to-evolve and hard-to-manage standard schema. Of course, in the case of the Semantic Web, the mappings between the sources should be specified declaratively. To complement the mappings, we need efficient algorithms that can follow *semantic paths* to obtain data from distant but related nodes on the web.

This paper describes the Piazza system, which provides an infrastructure for building Semantic Web applications, and addresses the aforementioned problems. A Piazza application consists of many nodes, each of which can serve either or both of two roles: supplying source data with its schema, or providing only a schema (or ontology). A very simple node might only supply data (perhaps from a relational database); at the other extreme, a node might simply provide a schema or ontology to which other nodes' schemas may be mapped. The semantic glue in Piazza is provided by *local* mappings between small sets (usually pairs) of nodes. When a query is posed over the schema of a node, the system will utilize data from any node that is transitively connected by semantic mappings, by *chaining* mappings. Piazza's architecture can accommodate both local point-to-point mappings between data sources, as well as collaboration through select mediated ontologies. Since the architecture is reminiscent of peer-to-peer architectures, we refer to Piazza as a *peer data management system* (PDMS).

We make the following specific contributions.

- We propose a language for mediating between nodes that allows mapping simple forms of domain structure *and* rich document structure. The language is based on XQuery [6], the emerging standard for querying XML. We also show that this language can map between nodes containing RDF data and nodes containing XML data.

- We describe an algorithm for answering queries in Piazza that chains semantic mappings specified in our language. The challenge in developing the algorithm is that the mappings are *directional,* and hence may sometimes need to be traversed in reverse. In fact, the algorithm can also go in reverse through mappings from XML to RDF that flatten out the document structure. Previous work [16] has presented an analogous algorithm for the simple case where all data sources are relational. Here we extend the algorithms considerably to the XML setting.

- Finally, we describe an implemented scenario using Piazza and several observations from this experience. The scenario includes 15 nodes (based on the structures and data of real web sites) that provide information about different aspects of the database research community.

At a more conceptual level, we believe that Piazza paves the way for a fruitful combination of data management and knowledge representation techniques in the construction of the Semantic Web. In fact, we emphasize that the techniques offered in Piazza are *not* a replacement for rich ontologies and languages for mapping between ontologies. Our goal is to provide the missing link between data described using rich ontologies and the wealth of data that is currently managed by a variety of tools. See [19] for a discussion of additional challenges in this area.

The paper is organized as follows. Section 2 provides an overview of Piazza, and Section 3 describes the language for mapping between nodes in Piazza. Section 4 presents the key algorithm underlying query answering in Piazza. In Section 5 we offer our experiences from implementing the scenario. Section 6 describes related work, and Section 7 concludes.

2. SYSTEM OVERVIEW

We begin by providing an overview of the concepts underlying Piazza and our approach to building Semantic Web applications.

2.1 Data, Schemas, and Queries

Our ultimate goal with Piazza is to provide query answering and translation across the full range of data, from RDF and its associated ontologies to XML, which has a substantially less expressive schema language. The main focus of this paper is on sharing XML data, but we explain how to accommodate richer data as we proceed.

Today, most commercial and scientific applications have facilities for automatically exporting their data into XML form. Hence, for the purpose of our discussion, we can consider XML to be the standard representation of a wide variety of data sources (as do others [28]). In some cases, accessing the actual data may require an additional level of translation (e.g., with systems like [13, 31]). Perhaps of equal importance, many applications, tools, and programming languages or libraries have facilities for loading, processing, and importing XML data. In the ideal case, one could map the wealth of existing XML-style data into the Semantic Web and query it using Semantic Web tools; correspondingly, one could take the results of Semantic Web queries and map them back into XML so they can be fed into conventional applications.

RDF is neutral with respect to objects' importance: it represents a graph of interlinked objects, properties, and values. RDF also assigns uniform semantic meaning to certain reserved objects (e.g., containers) and properties (e.g., identifiers, object types, references). Relationships between pairs of objects are explicitly named.

The main distinctions between RDF and unordered XML[1] are that XML (unless accompanied by a schema) does not assign semantic meaning to any particular attributes, and XML uses hierarchy (membership) to implicitly encode logical relationships. Within

[1] In this paper we consider only unordered XML; order information can still be encoded within the data.

an XML hierarchy, the central objects are typically at the top, and related objects are often embedded as subelements within the *document structure*; this embedding of objects creates binary relationships. Of course, XML may also include links and can represent arbitrary graphs, but the predominant theme in XML data is nesting. Whereas RDF names all binary relationships between pairs of objects, XML typically does not. The semantic meaning of these relationships is expressed within the schema or simply within the interpretation of the data. Hence, it is important to note that although XML is often perceived as having only a syntax, it is more accurately viewed as a semantically grounded encoding for data, in a similar fashion to a relational database. Importantly, as pointed out by Patel-Schneider and Simeon [28], if XML is extended simply by reserving certain attribute names to serve as element IDs and IDREFs, one can maintain RDF semantics in the XML representation.

As with data, the XML and RDF worlds use different formalisms for expressing schema. The XML world uses XML Schema, which is based on object-oriented classes and database schemas: it defines classes and subclasses, and it specifies or restricts their structure and also assigns special semantic meaning (e.g., keys or references) to certain fields. In contrast, languages such as RDFS, DAML+OIL [17] and OWL [9] come from the Knowledge Representation (KR) heritage, where ontologies are used to represent sets of objects in the domain and relationships between sets. OWL uses portions of XML Schema to express the structure of so-called *domain values*. In the remainder of this paper, we refer to OWL as the representative of this class of languages.

It is important to note that some of the functionality of KR descriptions and concept definitions can be captured in the XML world (and more generally, in the database world) using *views*. In the KR world, concept definitions are used to represent a certain set of objects based on constraints they satisfy, and they are compared via *subsumption* algorithms. In the XML world, queries serve a similar purpose, and furthermore, when they are named as views, they can be referenced by other queries or views. Since a view can express constraints or combine data from multiple structures, it can perform a role like that of the KR concept definition. Queries can be compared using *query containment* algorithms. There is a detailed literature that studies the differences between the expressive power of description logics and query languages and the complexity of the subsumption and containment problem for them (e.g., [21]). For example, certain forms of negation and number restrictions, when present in query expressions, make query containment undecidable, while arbitrary join conditions cannot be expressed and reasoned about in description logics.

Many different types of semantic mappings are required in converting within and between the XML and RDF worlds: one-to-one correspondences may occur between concepts, requiring simple renamings; more complex, n-to-m-entity correspondences may require join-like operations; there may be complex restructurings of concept definitions in going from one format to another (especially when XML is involved); and some complex concept definitions may require significant inference capabilities. For several reasons we focus on an XQuery-based approach to defining mappings: (1) it is important to be able to map existing XML data into RDF, and this requires the strong restructuring, joining, and renaming capabilities of XQuery; (2) existing, scalable, and practical techniques have been developed for reasoning about query-based mappings in the database community, and we can leverage these; (3) while XQuery views are less expressive than OWL concept definitions, they can capture many common types of semantic mappings, and we expect that they can be supplemented with further OWL constructs as necessary.

2.2 Data Sharing and Mediation

Logically, a Piazza system consists of a network of different sites (also referred to as peers or nodes), each of which contributes resources to the overall system. The resources contributed by a site include one or more of the following: (1) ground or *extensional* data, e.g., XML or RDF data instances, (2) models of data, e.g., XML schema or OWL ontologies. In addition, nodes may supply *computed data*, i.e., cached answers to queries posed over other nodes.

When a new site (with data instance or schema) is added to the system, it is semantically related to some portion of the existing network, as we describe in the next paragraph. Queries in Piazza are always posed from the perspective of a given site's schema, which defines the preferred terminology of the user. When a query is posed, Piazza provides answers that utilize all semantically related XML data within the system[2].

In order to exploit data from other sites, there must be *semantic glue* between the sites, in the form of semantic mappings. Mappings in Piazza are specified between small numbers of sites, usually pairs. In this way, we are able to support the two rather different methods for semantic mediation mentioned earlier: *mediated mapping*, where data sources are related through a mediated schema or ontology, and *point-to-point mappings*, where data is described by how it can be translated to conform to the schema of another site. Admittedly, from a formal perspective, there is little difference between these two kinds of mappings, but in practice, content providers may have strong preferences for one or the other.

The actual formalism for specifying mappings depends on the kinds of sites we are mapping. There are three main cases, depending on whether we are mapping between pairs of OWL/RDF nodes, between pairs of XML/XML Schema nodes, or between nodes of different types.

Pairs of OWL/RDF nodes: OWL itself already provides the constructs necessary for mapping between two OWL ontologies. Specifically, OWL's owl:equivalentProperty construct declares that two edge labels denote the same relationship. The owl:equivalentClass construct is even more powerful: one can use it to create a boolean combination of the classes in a source ontology and equate that to a class (or even another boolean combination) in a target ontology. In principle, the reasoning procedures for OWL can be used to provide reasoning across ontologies, and hence integrate data from multiple nodes. Performing such reasoning efficiently raises many interesting research questions.

Pairs of XML/XML Schema nodes: This case is more challenging because it does not make sense to simply assert that two structures should be considered the same. To illustrate the challenges associated with designing a language for mapping between two XML nodes, consider the following example.

EXAMPLE 2.1. *Suppose we want to map between two sites. Suppose the target contains* books *with nested* authors*; the source contains* authors *with nested* publications*. We illustrate partial schemas for these sources below, using a format in which indentation illustrates nesting and a * suffix indicates "0 or more occurrences of...", as in a BNF grammar.*

[2]It is also be possible to let the user narrow the set of sites considered in a query; this does not introduce any difficulties.

Target:

```
pubs
  book*
    title
    author*
      name
    publisher*
      name
```

Source:

```
authors
  author*
    full-name
    publication*
      title
      pub-type
```

In general, it should be possible to specify mappings in either direction (for reasons we discuss in the next section), and mappings must have two important capabilities:

- **Translation of domain structure and terminology:** In the simple case, we must be able to perform simple **renamings** from one concept (XML tag label) to another, either globally or within a certain subtree or context. For instance, we want to state that every occurrence of the full-name tag in S2 matches the name tag in S1. On the other hand, if we create a mapping in the reverse direction, name in S1 only corresponds to full-name in S2 when it appears within an author tag. In some cases, the terminological translations involve additional conditions. For instance, a title entry in site S2 is only equivalent to a book title in S1 if the pub-type is book.
- **Translation of document structure:** We must be able to map between different nesting structures. Source S1 is book-centric and S2 is author-centric. In order to do this, we must be able to coalesce groups of items when they are associated with the same entity — every time we see a book with the same name in S1, we should insert the book's title (within a publication element and with a pub-type of book) into the *same* author element in S2.

Section 3 describes our mapping specification language for mappings between XML/XML Schema nodes, which achieves these goals. The language is based on features of the XQuery XML query language [6], which is able to specify rich transformations.

XML-to-RDF mappings: There are two issues when mapping between XML to RDF/OWL data. The first is expressive power — clearly, we cannot map all the concepts in an OWL ontology into an XML schema and preserve their semantics. It is inevitable that we will lose some information in such a mapping. In practice, we need to ensure that the XML schema of a node is rich enough for the queries that are likely to be posed at the node.

The second issue is how to rebuild the appropriate document structure when transferring data from the OWL ontology into XML. We illustrate the challenge below.

EXAMPLE 2.2. *Suppose we have a simple network with three nodes: A and B are XML nodes and P is an RDF node with an associated OWL ontology. XML Node A contains author information, including books written by the author, nested within* author *elements (and a given book may appear under multiple* author *elements). Node B contains book information, including authors, nested within* book *elements (again, an author may appear within multiple* books*). Hence, nodes A and B contain the same data but in different structures. Finally, Node P is a rich OWL ontology describing the Publishing world. Among other concept definitions, it contains two classes (*Author *and* Book*) and one property (*writes*). The relationship in P can be encoded in RDF using the definition:*

```
<rdf:Description rdf:about="authorID"
                 rdf:type="Author">
  <P:writes rdf:resource="bookID"/>
</rdf:Description>
```

The important point to note is that once data has been mapped (using the mapping language described in Section 3) from nodes A or B to RDF, it loses its original document structure. In fact, the two different structures of nodes A and B are mapped to the *same* RDF. Our mapping language can be used to map from the XML of A and B into XML-encoded RDF at P. We could also write mappings in the opposite direction, from the RDF to XML, that restore the document structure. However, we would like to avoid having to write *two* mappings in every case. In fact, as we explain in the next section, we may compromise expressive power by forcing mappings in both directions.

Hence, suppose we have two mappings $A \rightarrow P$ and $B \rightarrow P$ from the XML to the RDF. Answering a query over the RDF is conceptually easy. Note that the RDF query is oblivious to document structure. The interesting case occurs when a query is posed over one of the XML sources, say node B. Here, we must use P as an intermediate node for getting data from node A. Data from A is first mapped into RDF form using the $A \rightarrow P$ mapping, "flattening" it and relating it to the ontology at P. Then, we need to somehow use the mapping $B \rightarrow P$ in *reverse* in order to answer the B query. In Section 4 we describe an algorithm that is also able to use XML-to-RDF mappings in the reverse direction. With that algorithm, we can follow any semantic path in Piazza, regardless of the direction in which the mappings are specified.

In summary, the language we describe in Section 3 offers a mechanism for inter-operation of XML/XML Schema nodes and RDF/OWL nodes. It enables mapping between XML nodes and between an XML node and an RDF node.

2.3 Query Processing

Given a set of sites, the semantic mappings between them, and a query at a particular site, the key problem we face is how to process queries. The problem is at two levels: (1) how to obtain semantically correct answers, and (2) how to process the queries efficiently. In this paper we focus mostly on the first problem, called *query reformulation*. Section 4 describes a query answering algorithm for the Piazza mapping language: given a query at a particular site, we need to expand and translate it into appropriate queries over semantically related sites, as well. Query answering may require that we follow semantic mappings in both directions. In one direction, composing semantic mappings is simply query composition for an XQuery-like language. In the other direction, composing mappings requires using mappings in the reverse direction, which is known as the problem of answering queries using views [15]. These two problems are well understood in the relational setting (i.e., when data is relational and mappings are specified as some restricted version of SQL), but they have only recently been treated in limited XML settings.

3. MAPPINGS IN PIAZZA

In this section, we describe the language we use for mapping between sites in a Piazza network. As described earlier, we focus on nodes whose data is available in XML (perhaps via a wrapper over some other system). For the purposes of our discussion, we ignore the XML document order. Each node has a schema, expressed in XML Schema, which defines the terminology and the structural constraints of the node. We make a clear distinction between the intended domain of the terms defined by the schema at a node and the actual data that may be stored there. Clearly, the stored data conforms to the terms and constraints of the schema, but the intended domain of the terms may be much broader than the particular data stored at the node. For example, the terminology for publications applies to data instances beyond the particular ones stored at the

node.

Given this setting, mappings play two roles. The first role is as *storage descriptions* that specify which data is actually stored at a node. This allows us to separate between the intended domain and the actual data stored at the node. For example, we may specify that a particular node contains publications whose topic is Computer Science and have at least one author from the University of Washington. The second role is as *schema mappings*, which describe how the terminology and structure of one node correspond to those in a second node. The language for storage mappings is a subset of the language for schema mappings, hence our discussion focuses on the latter.

The ultimate goal of the Piazza system is to use mappings to answer queries; we answer each query by rewriting it using the information in the mapping. Of course, we want to capture structural as well as terminological correspondences. As such, it is important that the mapping capture maximal information about the relationship between schemas, but also about the data instances themselves — since information about content can be exploited to more precisely answer a query.

The field of data integration has spent many years studying techniques for precisely defining such mappings with relational data, and we base our techniques on this work. In many ways, the vision of Piazza is a broad generalization of data integration: in conventional data integration, we have a *mediator* that presents a *mediated schema*, and a set of data sources that are mapped to this single mediated schema; in Piazza, we have a web of sites and semantic mappings.

The bulk of the data integration literature uses queries (views) as its mechanism for describing mappings: views can relate disparate relational structures, and can also impose restrictions on data values. There are two standard ways of using views for specifying mappings in this context: data sources can be described as views over the mediated schema (this is referred to as *local-as-view* or LAV), or the mediated schema can be described as a set of views over the data sources (*global-as-view* or GAV). The direction of the mapping matters a great deal: it affects both the kinds of queries that can be answered and the complexity of using the mapping to answer the query. In the GAV approach, query answering requires only relatively simple techniques to "unfold" (basically, macro-expand) the views into the query so it refers to the underlying data sources. The LAV approach requires more sophisticated query reformulation algorithms (surveyed in [15]), because we need to use the views in the *reverse* direction. It is important to note that in general, using a view in the reverse direction is not equivalent to writing an inverse mapping.

As a result of this, LAV offers a level of flexibility that is not possible with GAV. In particular, the important property of LAV is that it enables to describe data sources that organize their data *differently* from the mediated schema. For example, suppose the mediated schema contains a relationship Author, between a paper-id and an author-id. A data source, on the other hand, has the relationship CoAuthor that relates two author-id's. Using LAV, we can express the fact that the data source has the join of Author with itself. This description enables us to answer certain queries — while it is not possible to use the source to find authors of a particular paper, we can use the source to find someone's co-authors, or to find authors who have co-authored with at least one other. With GAV we would lose the ability to answer these queries, because we lose the association between co-authors. The best we could say is that the source provides values for the second attribute of Author.[3] (Recall that the relational data model is very weak at modeling incomplete information.)

This discussion has a very important consequence as we consider mappings in Piazza. When we map between two sites, our mappings, like views, will be directional. One could argue that we can always provide mappings in both directions, and even though this doubles our mapping efforts, it avoids the need for using mappings in reverse during query reformulation. However, when two sites organize their schemas differently, some semantic relationships between them will be captured only by the mapping in *one* of the directions, and this mapping cannot simply be inverted. Instead, these semantic relationships will be exploited by algorithms that can reverse through mappings on a per-query basis, as we illustrated in our example above. Hence, the ability to use mappings in the reverse direction is a key element of our ability to share data among sites, and therefore the focus of Section 4.

Our goal in Piazza is to leverage this work — both LAV and GAV — from data integration, but to extend it in two important directions. First, we must extend the basic techniques from the two-tier data integration architecture to the peer data management system's heterogeneous, graph-structured network of interconnected nodes; this was the focus of our work in [16]. Our second direction, which we discuss in this paper, is to move these techniques into the realms of XML as well as its serializations of RDF.

Following the data integration literature, which uses a standard relational query language for both queries and mappings, we might elect to use XQuery for both our query language and our language for specifying mappings. However, we found XQuery inappropriate as a mapping language for the following reasons. First, an XQuery user thinks in terms of the input documents and the transformations to be performed. The mental connection to a required schema for the output is tenuous, whereas our setting requires thinking about relationships between the input and output schemas. Second, the the user must define a mapping in its entirety before it can be used. There is no simple way to define mappings incrementally for different parts of the schemas, to collaborate with other experts on developing sub-regions of the mapping, etc. Finally, XQuery is an extremely powerful query language (and is, in fact, Turing-complete), and as a result some aspects of the language make it difficult or even impossible to reason about.

3.1 The Mapping Language

Our approach is to define a mapping language that borrows elements of XQuery, but is more tractable to reason about and can be expressed in *piecewise* form. Mappings in the language are defined as one or more *mapping definitions*, and they are *directional* from a source to a target: we take a fragment of the target schema and annotate it with XML query expressions that define what source data should be mapped into that fragment. The mapping language is designed to make it easy for the mapping designer to visualize the target schema while describing where its data originates.

Conceptually, the results of the different mapping definitions are combined to form a complete mapping from the source document to the target, according to certain rules. For instance, the results of different mapping definitions can often be concatenated together to form the document, but in some cases different definitions may create content that should all be combined into a single element; Piazza must "fuse" these results together based on the output element's unique identifiers (similar to the use of Skolem functions in languages such as XML-QL [10]). A complete formal description

[3] Note that in principle it is possible to define a CoAuthor view in the mediated schema, and map the data source to the view. However, the algorithmic problem of query answering would be identical to the LAV scenario.

of the language would be too lengthy for this paper. Hence, we describe the main ideas of the language and illustrate it via examples.

Each mapping definition begins with an XML template that matches some path or subtree of a legal instance of the target schema, i.e., a prefix of a legal string in the target DTD's grammar. Elements in the template may be annotated with query expressions (in a subset of XQuery) that bind variables to XML nodes in the source; for each combination of bindings, an instance of the target element will be created. Once a variable is bound, it can be referenced anywhere within its scope, which is defined to be the enclosing tags of the template. Variable bindings can be output as new target data, or they can be referenced by other query expressions to *correlate* data in different areas of the mapping definition. The following is a basic example of the language for the sites in Example 2.1.

```
<pubs>
  <book>
    {: $a IN document("source.xml")\
                /authors/author,
       $t IN $a/publication/title,
       $typ IN $a/publication/pub-type
       WHERE $typ = "book" :}
    <title>{ $t }</title>
    <author>
      <name> {: $a/full-name :} </name>
    </author>
  </book>
</pubs>
```

Where we make variable references within {} braces and delimit query expression annotations by `{: :}`. This mapping definition will instantiate a new book element in the target for every occurrence of variables `$a`, `$t`, and `$typ`, which are bound to the author, title, and publication-type elements in the source, respectively. We construct a title and author element for each occurrence of the book. The author name contains a new query expression annotation (`$a/full-name`), so this element will be created for each match to the XPath expression (for this schema, there should only be one match).

The example mapping will create a new book element for each author-publication combination. This is probably not the desired behavior, since a book with multiple authors will appear as multiple book entries, rather than as a single book with multiple author subelements. To enable the desired behavior in situations like this, Piazza reserves a special `piazza:id` attribute in the target schema for mapping multiple binding instances to the same output: if two elements created in the target have the same tag name and ID attribute, then they will be *coalesced* — all of their attributes and element content will be combined. This coalescing process is repeated recursively over the combined elements. We can modify our mapping to the following:

EXAMPLE 3.1.

```
<pubs>
  <book piazza:id={$t}>
    {: $a IN document("source.xml")\
                /authors/author,
       $t IN $a/publication/title,
       $typ IN $a/publication/pub-type
       WHERE $typ = "book" :}
    <title piazza:id={$t}>{ $t }</title>
    <author piazza:id={$t}>
      <name> {: $a/full-name :} </name>
    </author>
  </book>
</pubs>
```

The sole difference from the previous example is the use of the piazza:id *attribute. We have determined that book titles in our collection are unique, so every occurrence of a title in the data source refers to the* same *book. Identical books will be given the same* piazza:id *and coalesced; likewise for their* title *and* author *subelements (but not author* names*). Hence, in the target we will see all authors nested under each book entry. This example shows how we can* invert *hierarchies in going from source to target schemas.*

Sometimes, we may have detailed information about the values of the data being mapped from the source to the target — perhaps in the above example, we know that the mapping definition only yields book titles starting with the letter "A." Perhaps more interestingly, we may know something about the possible values of an attribute present in the target but *absent* in the source — such as the publisher. In Piazza, we refer to this sort of meta-information as *properties*. This information can be used to help the query answering system determine whether a mapping is relevant to a particular query, so it is very useful for efficiency purposes.

EXAMPLE 3.2. *Continuing with the previous schema, consider the partial mapping:*

```
<pubs>
  <book piazza:id={$t}>
    {: $a IN document("source.xml")\
                /authors/author,
       $t IN $a/publication/title,
       $typ IN $a/publication/pub-type
       WHERE $typ = "book"
       PROPERTY $t >= 'A' AND $t < 'B'
    :}
    [:
    <publisher>
      <name>
        {: PROPERTY $this IN
           {"PrintersInc", "PubsInc"} :}
      </name>
    </publisher>
    :]
  </book>
</pubs>
```

The first PROPERTY definition specifies that we know this mapping includes only titles starting with "A." The second defines a "virtual subtree" (delimited by `[: :]`) in the target. There is insufficient data at the source to insert a value for the publisher name*; but we can define a PROPERTY restriction on the values it* might *have. The special variable `$this` allows us to establish a known invariant about the value at the current location within the virtual subtree: in this case, it is known that the publisher name must be one of the two values specified. In general, a query over the target looking for books will make use of this mapping; a query looking for books published by* BooksInc *will not. Moreover, a query looking for books published by* PubsInc *cannot use this mapping, since Piazza cannot tell whether a book was published by* PubsInc *or by* PrintersInc*.*

3.2 Semantics of Mappings

We briefly sketch the principles underlying the semantics of our mapping language. At the core, the semantics of mappings can be defined as follows. Given an XML instance, I_s, for the source node S and the mapping to the target T, the mapping defines a *subset* of an instance, I_t, for the target node. The reason that I_t is a subset of the target instance is that some elements of the target may not exist in the source (e.g., the publisher element in the examples). In fact, it may even be the case that required elements of the target are not present in the source. In relational terms, I_t is a *projection* of some complete instance I'_t of T on a subset of its elements and attributes. In fact, I_t defines a *set* of complete instances of T

whose projection is I_t. When we answer queries over the target T, we provide only the answers that are consistent with *all* such I_t's (known as the *certain answers* [1], the basis for specifying semantics in the data integration literature). It is important to note that partial instances of the target *are* useful for many queries, in particular, when a query asks for a subset of the elements. Instances for T may be obtained from multiple mappings (and instances of the sources, in turn, can originate from multiple mappings), and as we described earlier, may be the result of coalescing the data obtained from multiple bindings using the piazza:id attribute.

A mapping between two nodes can either be an *inclusion* or an *equality* mapping. In the former case, we can only infer instances of the target from instances of the source. In the latter case, we can also infer instances of the source from instances of the target. However, since the mapping is defined from the source to the target, using the mapping in reverse requires special reasoning. The algorithm for doing such reasoning is the subject of Section 4. Finally, we note that storage descriptions, which relate the node's schema to its actual current contents allow for both the *open-world assumption* or the *closed-world assumption*. In the former case, a node is not assumed to store all the data modeled by its schema (it describes a general concept more inclusive than the data it provides, e.g., all books published, and new data sources may provide additional data for this schema), while in the latter case it holds the complete set of all data relevant to its concept (e.g., all books published by major publishers since 1970). In practice, very few data sources have complete information.

3.3 Discussion

To complete the discussion of our relationship to data integration, we briefly discuss how our mapping language relates to the LAV and GAV formalisms. In our language, we specify a mapping from the perspective of a particular *target schema* — in essence, we define the target schema using a GAV-like definition relative to the source schemas. However, two important features of our language would require LAV definition in the relational setting. First, we can map data sources to the target schema even if the data sources are missing attributes or subelements required in the source schema. Hence, we can support the situation where the source schema is a projection of the target. Second, we support the notion of data source *properties*, which essentially describes scenarios in which the source schema is a selection on the target schema.

Hence, our language combines the important properties of LAV and GAV. It is also interesting to note that although query answering in the XML context is fundamentally harder than in the relational case, specifying mappings between XML sources is more intuitive. The XML world is fundamentally semistructured, so it can accommodate mappings from data sources that lack certain attributes — without requiring null values. In fact, during query answering we allow mappings to pass along elements from the source that do not exist in the target schema — we would prefer not to discard these data items during the transitive evaluation of mappings, or query results would always be restricted by the lowest-common-denominator schemas along a given mapping chain. For this reason, we do not validate the schema of answers before returning them to the user.

4. QUERY ANSWERING ALGORITHM

Given a set of mappings, our goal is to be able to answer queries posed over any peer's schema, making use of all relevant (mapped) data. We do this at runtime rather than mapping the data once and later answering queries: this allows us to provide "live" answers as source data changes, and we can sometimes exploit "partial" mappings to answer certain queries, even if those mappings are insufficient to entirely transform data from one schema to another.

This section describes Piazza's query answering algorithm, which performs the following task: given a network of Piazza nodes with XML data, a set of semantic mappings specified among them, and a query over the schema of a given node, efficiently produce *all* the possible *certain answers* that can be obtained from the system. A user's query is posed over a node's logical schema, which may be defined in terms of incomplete data sources (e.g., we may define the concept "all books published" but may not have complete knowledge of these books). Certain answers are those results that are *guaranteed* to be in the logical schema in order for it to be consistent with the mappings and the existing source data.

From a high level, an algorithm proceeds along the following lines. Given a query Q posed over the schema of node P, we first use the storage descriptions of data in P (i.e., the mappings that describe which data is actually stored at P) to rewrite Q into a query Q' over the data stored at P. Next, we consider the *semantic neighbors* of P, i.e., all nodes that are related to elements of P's schema by semantic mappings. We use these mappings to expand the reformulation of query Q to a query Q'' over the neighbors of P. In turn, we expand Q'' so it only refers to stored data in P and its neighbors; then we union it with Q', eliminating any redundancies. We repeat this process recursively, following all mappings between nodes' schemas, and the storage mappings for each one, until there are no remaining useful paths.

Ignoring optimization issues, the key question in designing such an algorithm is how to reformulate a query Q over its semantic neighbors. Since semantic mappings in Piazza are directional from a source node S to a target node T, there are two cases of the reformulation problem, depending on whether Q is posed over the schema of S or over that of T. If the query is posed over T, then query reformulation amounts to query composition: to use data at S, we compose the query Q with the query (or queries) defining T in terms of S. Our approach to query composition is based on that of [13], and we do not elaborate on it here.

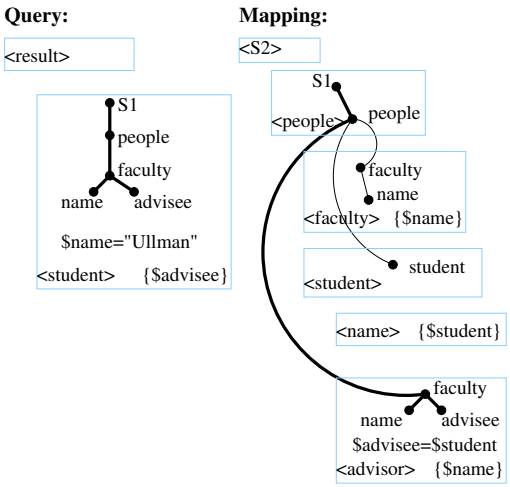
The second case is when query is posed over S and we wish to reformulate it over T. Now both Q and T are defined as queries over S. In order to reformulate Q, we need to somehow use the mapping in the *reverse* direction, as explained in the previous section. This problem is known as the problem of answering queries using views (see [15] for a survey), and is conceptually much more challenging. The problem is well understood for the case of relational queries and views, and we now describe an algorithm that applies to the XML setting. The key challenge we address for the context of XML is the nesting structure of the data (and hence of the query) — relational data is flat.

4.1 Query Representation

Our algorithm operates over a graph representation of queries and mappings. Suppose we are given the following XQuery for all advisees of Ullman, posed over source S1:

```
<result> {
  for $faculty in /S1/people/faculty,
      $name in $faculty/name/text(),
      $advisee in $faculty/advisee/text()
  where $name = "Ullman"
  return
    <student> {$advisee} </student>
}
</result>
```

The query is represented graphically by the leftmost portion of Figure 1. Note that the `result` element in the query simply specifies the root element for the resulting document. Each box in the




```
<S2>

  <people> {: $people=/S1/people :}

    <faculty>{: $name=$people/faculty/name/text():}
      { $name }
    </faculty>

    <student>{: $student=$people/student/text():}
      <name> { $student } </name>

      <advisor>{: $faculty=$people/faculty,
                  $name=$faculty/name/text(),
                  $advisee=$faculty/advisee/text()
                  where $advisee=$student :}
        { $name }
      </advisor>
    </student>
  </people>
</S2>
```

Figure 1: Matching a query tree pattern into a tree pattern of a schema mapping. The matching tree patterns are shown in bold. The schema mapping corresponding to the middle graph is shown on the right.

figure corresponds to a query block, and indentation indicating the nesting structure. With each block we associate the following constructs that are manipulated by our algorithm:

A set of tree patterns: XQuery's FOR clause binds variables, e.g., `$faculty in /S1/people/faculty` binds the variable `$faculty` to the nodes satisfying the XPath expression. The bound variable can then be used to define new XPath expressions such as `$faculty/name` and bind new variables. Our algorithm consolidates XPath expressions into logically equivalent *tree patterns* for use in reformulation[4]. For example, the tree pattern for our example query is indicated by the thick forked line in the leftmost portion of Figure 1.

For simplicity of presentation, we assume here that every node in a tree pattern binds a single variable; the name of the variable is the same as the tag of the corresponding tree pattern node. Hence, the node `advisee` of the tree pattern binds the variable `$advisee`.

A set of predicates: a predicate in a query specifies a condition on one or two of the bound variables. Predicates are defined in the XQuery WHERE clause over the variables bound in the tree patterns. The variables referred to in the predicate can be bound by different tree patterns. In our example, there is a single predicate: `name="Ullman"`. If a predicate involves a comparison between two variables, then it is called a *join* predicate, because it essentially enforces a relational join.

Output results: output, specified in the XQuery RETURN clause, consists of element or attribute names and their content. An element tag name is usually specified in the query as a string literal, but it can also be the value of a variable. This is an important feature, because it enables transformations in which data from one source becomes schema information in another. In our query graph of Figure 1, an element tag is shown in angle brackets. Hence, the element tag of the top-level block is result. The element tag of the inner block is student. The contents of the returned element of a query block may be a sequence of elements, attributes, string literals, or variables. (Note that our algorithm does not support "mixed content," in which subelements and data values may be siblings, as this makes reformulation much harder). We limit our discussion to the case of a single returned item. In the figure, the variable/value returned by a query block is enclosed in curly braces. Thus, the top level block of our example query has empty returned contents, whereas the inner block returns the value of the `$advisee` variable.

We use the same representation for mappings as for queries. In this case, the nesting mirrors the template of the target schema. The middle of Figure 1 shows the graph representation of the mapping shown on the right of the figure. The mapping is between the following schemas. (The schemas differ in how they represent advisor-advisee information. `S1` puts advisee names under the corresponding faculty advisor whereas `S2` does the opposite by nesting advisor names data under corresponding students.)

```
S1                  S2
  people              people
    faculty*            faculty*
      name              student
      advisee*            name
    student*              advisor*
```

4.2 The Rewriting Algorithm

Our algorithm makes the following simplifying assumptions about the queries and the mappings (we note that in the scenario we implemented, all the mappings satisfied these restrictions). First, we assume the query over the target schema contains a single non-trivial block, i.e., a block that includes tree patterns and/or predicates. The mapping, on the other hand, is allowed to contain an arbitrary number of blocks. Second, we assume that all "returned" variables are bound to atomic values, i.e., `text()` nodes, rather than XML element trees (this particular limitation can easily be removed by expanding the query based on the schema). In Figure 1 the variable `$people` is bound to an element; variables `$name` and `$student` are bound to values. Third, we assume that queries are evaluated under a set semantics. In other words, we assume that duplicate results are eliminated in the original and rewritten query. Finally, we assume that a tree pattern uses the *child* axis of XPath only. It is possible to extend the algorithm to work with queries that use the *descendant* axis. For purposes of exposition, we assume that the schema mapping does not contain sibling blocks with the same element tag. Handling such a case requires the algorithm to consider multiple possible satisfying paths (and/or predicates) in the tree pattern.

Intuitively, the rewriting algorithm performs the following tasks. Given a query Q, it begins by comparing the tree patterns of the mapping definition with the tree pattern of Q — the goal is to find a corresponding node in the mapping definition's tree pattern for

[4]We focus on the subset of XPath that corresponds to regular path expressions, so tree patterns capture the required semantics.

every node in the Q's tree pattern. Then the algorithm must restructure Q's tree pattern along the same lines as the mapping restructures its input tree patterns (since Q must be rewritten to match against the *target* of the mapping rather than its source). Finally, the algorithm must ensure that the predicates of Q can be satisfied using the values output by the mapping. The steps performed by the algorithm are:

Step 1: pattern matching. This step considers the tree patterns in the query, and finds corresponding patterns in the target schema. Intuitively, given a tree pattern, t in Q, our goal is to find a tree pattern t' on the target schema such that the mapping guarantees that an instance of that pattern could only be created by following t in the source. The algorithm first matches the tree patterns in the query to the expressions in the mapping and records the corresponding nodes. In Figure 1, the darker lines in the representation of the schema mapping denote the tree pattern of the query (far left) and its corresponding form in the mapping (second from left). The algorithm then creates the tree pattern over the target schema as follows: starting with the recorded nodes in the mapping, it recursively marks all of their ancestor nodes in the output template. It then builds the new tree pattern over the target schema by traversing the mapping for all marked nodes.

Note that t' may enforce additional conditions to t, and that there may be several patterns in the target that match a pattern in the query, ultimately yielding several possible queries over the target that provide answers to Q. If no match is found, then the resulting rewriting will be empty (i.e., the target data does not enable answering the query on the source).

Step 2: Handling returned variables and predicates. In this step the algorithm ensures that all the variables required in the query can be returned, and that all the predicates in the query have been applied. Here, the nesting structure of XML data introduces subtleties beyond the relational case.

To illustrate the first potential problem, recall that our example query returns advisee names, but the mapping does not actually return the advisee, and hence the output of Step 1 does not return the advisee. We must extend the tree pattern to reach a block that actually outputs the `$advisee` element, but the `<advisor>` block where `$advisee` is bound does not have any subblocks, so we cannot simply extend the tree pattern. Fortunately, the `<advisor>` block includes an equality condition between `$advisee` and `$student`, which is output by the `<name>` block. We can therefore rewrite the tree pattern as `$student in /S2/people/student, $advisor in $student/advisor, $name in $student/name`. Of course, it is not always possible to find such equalities, and in those cases there will be no rewriting for that pattern.

Query predicates can be handled in one of three ways. First, a query predicate (or one that subsumes it) might already be applied by the relevant portion of the mapping (or might be a known property of the data being mapped). In this case, the algorithm can consider the predicate to be satisfied. A second case is when the mapping does not impose the predicate, but returns all nodes necessary for testing the predicate. Here, the algorithm simply inserts the predicate into the rewritten query. The third possibility is more XML-specific: the predicate is not applied by the portion of the mapping used in the query rewriting, nor can the predicate be evaluated over the mapping's output — but a different sub-block in the mapping may impose the predicate. If this occurs, the algorithm can *add a new path* into the rewritten tree pattern, traversing into the sub-block. Now the rewritten query will only return a value if the sub-block (and hence the predicate) is satisfied.

In our case, the query predicate can be reformulated in terms of the variables bound by the replacement tree pattern as follows:

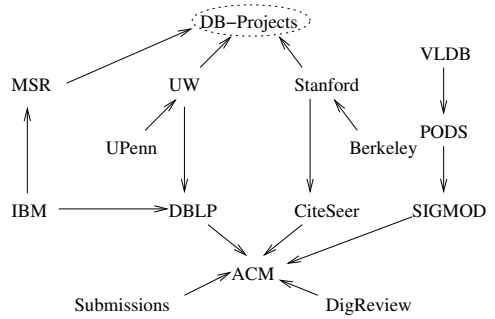


Figure 2: The topology of the DB-Research Piazza application.

`$advisor="Ullman"`. The resulting rewritten query in our example is the following.

```
<result> {
  for $student in /S2/people/student,
      $advisor in $student/advisor/text(),
      $name in $student/name/text()
  where $advisor = "Ullman"
  return
    <student> { $name } </student>
}
</result>
```

Note that in the above discussion, we always made the assumption that a mapping is useful if and only if it returns all output values and satisfies all predicates. In many cases, we may be able to loosen this restriction if we know more information about the relationships within a set of mappings, or about the properties of the mappings. For instance, if we have two mappings that share a key or a parent element, we may be able to rewrite the query to use both mappings if we add a join predicate on the key or the parent element ID, respectively. Conversely, we may be able to make use of properties to determine that a mapping cannot produce any results satisfying the query.

In the full version of the paper we prove the following theorem that characterizes the completeness of our algorithm.

THEOREM 1. *Let S and T be source and target XML schemas, and Q be a query over S, all of which satisfy the assumptions specified in the beginning of this section. Then, our algorithm will compute a query Q' that is guaranteed to produce all the certain answers to Q for any XML instance of T.* □

5. A PIAZZA APPLICATION

To validate our approach, we implemented a small but realistic semantic web application in Piazza. This section briefly reports on our experiences. While our prototype is still relatively preliminary, we can already make several interesting observations that are helping to shape our ideas for future research.

The Piazza system consists of two main components. The query reformulation engine takes a query posed over a node, and it uses the algorithm described in Section 4 in order to chain through the semantic mappings and output a set of queries over the relevant nodes. Our query evaluation engine is based on the Tukwila XML Query Engine [18], and it has the important property that it yields answers as the data is streaming in from the nodes on the network.

We chose our application, DB Research, to be representative of certain types of academic and scientific data exchange. Our prototype relates 15 nodes concerning different aspects of the database research field (see Figure 2, where directed arrows indicate the direction of mappings). The nodes of DB Research were chosen so they cover complementary but overlapping aspects of database

Table 1: The test queries and their respective running times.

Query	Description	Reformulation time	# of reformulations
Q1	XML-related projects.	0.5 sec	12
Q2	Co-authors who reviewed each other's work.	0.9 sec	25
Q3	PC members with a paper at the same conference.	0.2 sec	3
Q4	PC chairs of recent conferences + their projects.	0.5 sec	24
Q5	Conflicts-of-interest of PC members.	0.7 sec	36

research. All of the nodes of DB Research, with the exception of DB-Projects, contribute data. DB-Projects is a schema-only node whose goal is to map between other sources. DB Research nodes represent university database groups (Berkeley, Stanford, UPenn, and UW), research labs (IBM and MSR), online publication archives (ACM, DBLP, and CiteSeer), web sites for the major database conferences (SIGMOD, VLDB, and PODS), and DigReview, which is an open peer-review web site. The Submissions node represents data that is available only to a PC chair of a conference, and not shared with others. The node schemas were designed to mirror the actual organization and terminology of the corresponding web sites. When defining mappings, we tried to map as much information in the source schema into the target schema as possible, but a complete schema mapping is not always possible since the target schema may not have all of the attributes of the source schema. We report our experiences on four different aspects.

Reformulation times: the second and third columns of Table 1 show the reformulation time for the test queries and the number of reformulations obtained (i.e., number of queries that can be posed over the nodes to obtain answers to the query). We observe that even with relatively unoptimized code, the reformulation times are quite low, even though some of them required traversing paths of length 8 in the network. Hence, sharing data by query reformulation along semantic paths appears to be feasible. Although we expect many applications to have much larger networks, we also expect many of the paths in the network to require only very simple reformulations. Furthermore, by interleaving reformulation and query evaluation, we can start providing answers to users almost immediately.

Optimization issues: the interesting optimization issue that arises is reducing the number of reformulations. Currently, our algorithm may produce more reformulations than necessary because it may follow redundant paths in the network, or because it cannot detect a cyclic path until it traverses the final edge. Minimizing the number of reformulations has been considered in two-tier data integration systems, but the graph-structured nature of Piazza presents novel optimization issues.

Management of mapping networks: as we noted, due to differences between schemas of nodes, mappings may lose information. Just because there are two paths between a pair of nodes does not mean that one of them is redundant: one path may preserve information lost in the other. In fact, we sometimes may want to *boost* a set of paths by adding a few mappings that were not originally there. Analyzing mapping networks for information loss and proposing additional mappings presents an interesting set of challenges to investigate.

Locality of concept management: even comprehensive attempts to model the world with schemas or ontologies are ultimately approximations of the real world, and there may always exist some *ambiguity* in their meanings. Most users will have the same understanding of the concepts defined by a schema, but some discrepancies are bound to occur. In this respect, we believe that the point-to-point nature of the Piazza network architecture provides a significant advantage over the use of global ontologies or mediated schemas. Since nodes are free to choose with whom they would like to establish semantic connections, a node is likely to map into other nodes with a similar schema, because that is simply easier. As a result, a large network is likely to have clusters of nodes with a similar view of the world, and it will be easier to track discrepancies across clusters. In contrast, an architecture that maps all nodes to a single mediated schema is likely to suffer more severely from discrepancies because it requires global consistency.

Finally, we note that semantic web applications will differ on how strict answers need to be. If an application affects sensitive resources (e.g., bank accounts or even schedules), we should ensure that mappings are carefully controlled. However, much of the promise of the semantic web lies in systems that use the semantic markup to provide *best effort* query answering. Piazza provides an infrastructure for supporting both kinds of systems, but it emphasizes the flexibility and scalability needed for the latter kind.

6. RELATED WORK

One of the key goals of Piazza is to provide semantic mediation between disparate data sources. Federated databases [32] and data integration systems [27, 22] both address this problem, but they rely on a two-tier mediator architecture, in which data sources are mapped to a global *mediated schema* that encompassed all available information. This architecture requires centralized administration and schema design, and it does not scale to large numbers of small-scale collaborations. To better facilitate data sharing, Piazza adopts a peer-to-peer-style architecture and eliminates the need for a single unified schema — essentially, *every* node's schema can serve as the mediated schema for a query, and the system will evaluate schema mappings transitively to find all related data. Our initial work in this direction focused on the relational model and was presented in [16]; a language for mediating between relational sources has recently been presented in [5]. Mappings between schemas can be specified in many ways. Cluet et al. suggest a classification of mapping schemes between XML documents in [8]; following their framework, we could classify our system as mapping from paths to (partial) DTDs. The important, but complementary issue of providing support for generating semantic mappings between peers has been a topic of considerable interest in the database community [29, 11], and in the ontology literature [23, 12, 26]. The problem of estimating information loss in mappings has also been studied [24]. An important problem that we have not yet addressed is that of potential data source inconsistencies; but this problem has received recent attention in [3, 20].

A second goal of this paper is to address not only mediation between XML sources, but to provide an intermediary between the XML and RDF worlds, since most real-world data is in XML but ontologies may have richer information. Patel-Schneider and Simeon [28] propose techniques for merging XML and RDF into a common, XML-like representation. Conversely, the Sesame [7] stores RDF in a variety of underlying storage formats. Amann et al. [2] discuss a data integration system whereby XML sources are mapped into a simple ontology (supporting inheritance and roles, but no description logic-style definitions).

The Edutella system [25] represents an interesting design point

in the XML-RDF interoperability spectrum. Like Piazza, it is built on a peer-to-peer architecture and it mediates between different data representations. The focus of Edutella is to provide query and storage services for RDF, but with the ability to use many different underlying stores. Thus an important focus of the project is on translating the RDF data and queries to the underlying storage format and query language. Rather than beginning with data in a particular document structure and attempting to translate between different structures, Edutella begins with RDF and uses canonical mappings to store it in different subsystems. As a result of its inherent RDF-mediated architecture, Edutella does not employ point-to-point mappings between nodes. Edutella uses the JXTA peer-to-peer framework in order to provide replication and clustering services.

The architecture we have proposed for Piazza is a peer-to-peer, Web-like system. Recently, there has been significant interest in developing grid computing architectures (see www.mygrid.org.uk, www.gridcomputing.com), modeled after the electric power grid system. The goal is to construct a generic parallel, distributed environment for resource sharing and information exchange, and to allow arbitrary users (especially scientific users) to "plug in" to the grid. As noted in the lively discussion in [30], there will be some interesting relationships between grid computing and the Semantic Web. We believe that Piazza provides a data management infrastructure to support data services on the grid.

Finally, we note that Piazza is a component of the larger Revere Project [14] that attempts to address the entire life-cycle of content creation on the Semantic Web.

7. CONCLUSIONS AND FUTURE WORK

The vision of the semantic web is compelling and will certainly lead to significant changes in how the Web is used, but we are faced with a number of technical obstacles in realizing this vision. Knowledge representation techniques and standardized ontologies will undoubtedly play a major role in the ultimate solution. However, we believe that the semantic web cannot succeed if it requires everything to be rebuilt "from the ground up": it must be able to make use of structured data from non-semantic web-enabled sources, and it must inter-operate with traditional applications. This requires the ability to deal not only with domain structure, but also with document structures that are used by applications. Moreover, mediated schemas and ontologies can only be built by consensus, so they are unlikely to scale.

In this paper, we have presented the Piazza peer data management architecture as a means of addressing these two problems, and we have made the following contributions. First, we described a mapping language for mapping between sets of XML source nodes with different document structures (including those with XML serializations of RDF). Second, we have proposed an architecture that uses the transitive closure of mappings to answer queries. Third, we have described an algorithm for query answering over this transitive closure of mappings, which is able to follow mappings in both forward and reverse directions, and which can both remove and reconstruct XML document structure. Finally, we described several key observations about performance and research issues, given our experience with an implemented semantic web application.

Although our prototype application is still somewhat preliminary, it already suggests that our architecture provides useful and effective mediation for heterogeneous structured data, and that adding new sources is easier than in a traditional two-tier environment. Furthermore, the overall Piazza system gives us a strong research platform for uncovering and exploring issues in building a semantic web. We are currently pursuing a number of research directions. A key aspect of our system is that there may be many alternate "mapping paths" between any two nodes. An important problem is identifying how to prioritize these paths that preserve the most information, while avoiding paths that are too "diluted" to be useful. A related problem at the systems level is determining an optimal strategy for evaluating the rewritten query. We are also interested in studying Piazza's utility in applications that are much larger in scale, and in investigating strategies for caching and replicating data and mappings for reliability and performance.

Acknowledgments

The authors would like to express their gratitude to Natasha Noy, Rachel Pottinger, and Dan Weld for their invaluable comments and suggestions about this paper.

8. REFERENCES

[1] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *PODS '98*, pages 254–263, Seattle, WA, 1998.

[2] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Ontology-based integration of XML web resources. In *Int'l Semantic Web Conference '02*, pages 117–131, 2002.

[3] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. pages 68–79, 1999.

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

[5] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. In *ACM SIGMOD WebDB Workshop '02*, June 2002.

[6] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, 30 April 2002. W3C working draft.

[7] J. Broekstra, A. Kampan, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Int'l Semantic Web Conference '02*, pages 54–68, 2002.

[8] S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale XML repository. In *VLDB '01*, pages 271–280, September 2001.

[9] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. OWL web ontology language 1.0 reference, 2002. Manuscript available from `http://www.w3.org/2001/sw/WebOnt/`.

[10] A. Deutsch, M. F. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Eighth International World Wide Web Conference*, 1999.

[11] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD '01*, 2001.

[12] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to map between ontologies on the semantic web. In *Proc. of the Int. WWW Conf.*, 2002.

[13] M. Fernandez, W.-C. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. In *Ninth International World Wide Web Conference*, November 1999.

[14] A. Halevy, O. Etzioni, A. Doan, Z. Ives, J. Madhavan, L. McDowell, and I. Tatarinov. Crossing the structure chasm. In *Proceedings of the First Biennian Conference on Innovative Data Systems Research (CIDR)*, 2003.

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

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

[17] I. Horrocks, F. van Harmelen, and P. Patel-Schneider. DAML+OIL. http://www.daml.org/2001/03/daml+oil-index.html, March 2001.

[18] Z. Ives, A. Halevy, and D. Weld. An xml query engine for network-bound data. *VLDB Journal, Special Issue on XML Query Processing*, 2003.

[19] V. Kashyap. The semantic web: Has the db community missed the bus (again)? In *Proceedings of the NSF Workshop on DB & IS*

Research on the Semantic Web and Enterprises, Amicalola, GA, 2002.

[20] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *KRDB '02*, April 2002.

[21] A. Levy and M.-C. Rousset. Combining Horn rules and description logics in carin. *Artificial Intelligence*, 104:165–209, September 1998.

[22] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, pages 251–262, Bombay, India, 1996.

[23] D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder. The Chimæra ontology environment. In *AAAI '00*, 2000.

[24] E. Mena, V. Kashyap, A. Illarramendi, and A. P. Sheth. Imprecise answers in distributed environments: Estimation of information loss for multi-ontology based query processing. *International Journal of Cooperative Information Systems*, 9(4):403–425, 2000.

[25] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P networking infrastructure based on RDF. In *Eleventh International World Wide Web Conference*, pages 604–615, 2002.

[26] N. F. Noy and M. A. Musen. PROMPT: Algorithm and tool for ontology merging and alignment. In *AAAI '00*, 2000.

[27] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE '95*, pages 251–260, 1995.

[28] P. Patel-Schneider and J. Simeon. Building the Semantic Web on XML. In *Int'l Semantic Web Conference '02*, June 2002.

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

[30] D. D. Roure, I. Foster, E. Miller, J. Hendler, and C. Goble. The semantic grid: The grid meets the semantic web. Panel at the WWW Conference, Honolulu, Hawaii, 2002.

[31] M. Rys. Bringing the internet to your database: Using SQLServer 2000 and XML to build loosely-coupled systems. In *ICDE '01*, pages 465–472, 2001.

[32] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[33] P. Westerman. *Data Warehousing: Using the Wal-Mart Model*. Morgan Kaufmann Publishers, 2000.