Piazza: Data Management Infrastructure for Semantic Web Applications

Alon Y. Halevy, Zachary G. Ives, Peter Mork, Igor Tatarinov

Twelfth World Wide Web Conference, May 2003.

Write-up by Chernov Sergey

International Max Planck Research School for Computer Science

Saarland University, Department of Computer Science

Saarbrücken, 14 December 2003

Content

This write-up presents a paper "PIAZZA: Data Management Infrastructure for Semantic Web Applications" and related work made by the database group at the University of Washington. This review with corresponding presentation was produced as part of the seminar „Peer-to-peer Information Systems" given by Prof. Dr.-Ing. Gerhard Weikum in Saarland University.

1 INTRODUCTION

The following section describes some important concepts of Semantic Web and Data Integration field. It explains problems in this area and proposes a direction of research that leads to design of PIAZZA system.

1.1 Semantic Web

The current Web can be characterized as the second Web generation: the first generation Web started with handwritten HTML pages; the second generation made the step to machine generated and often active HTML pages. These generations of the Web were meant for direct human processing (reading, browsing, form-filling). The third generation Web, which one could call the Semantic Web, aims at machine processable information [7]. 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 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. It will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users [5]. Instead of posing queries that match text within documents, they 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.

1.2 Knowledge Representation

It is highly expectable that XML will be the methods of choice for representing all kinds of documents across the Web and it makes opinion that XML will be a major catalyst in constructing the Semantic Web. However, merely casting all documents into XML format does not necessarily make a document's semantics explicit and more amenable for effective information searching. Rather, to fully leverage XML on a global scale, significant progress is needed on the several issues. One of them is extracting more semantics from existing document collections by constructing structural and ontological skeletons that describe the data at a higher semantic level [8]. The XML-Schema language allows to develop a standard set of XML mappings (i.e. a particular XML schema), and to represent content using a common structure. But sometimes a capability beyond that offered by XML-schema is needed if we need to provide mapping capabilities between divergent schemas. For this reason, a fundamental component of the Semantic Web is the Resource Description Framework (RDF) [9]. It could be quite different opinions about most efficient standard for semantic integration and even it could be old good relational database schemas. But some could see that proposals aiming at semantic interoperability are the results of recent W3C standardization efforts, notably XML/XML Schema and RDF/RDF Schema as well as languages that build upon these data models: DAML+OIL and OWL [7]. Hence, for the purpose of our discussion, we can consider XML to be the standard representation of a wide variety of data sources.

The current progress on developing ontologies and representation languages leaves us with two significant problems. The first problem 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.

1.3 Scalability

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. Interoperability is typically attained in the real world by writing translators 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.

Database group from University of Washington presents the Piazza system that provides an infrastructure for building Semantic Web applications, and addresses the aforementioned problems.

2 PIAZZA: SYSTEM OVERVIEW

This section provides an overview of the concepts underlying Piazza and the system architecture.

2.1 Peer Data Management System Architecture

The 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 that has a substantially less expressive schema language. 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).

In addition, nodes may supply

- computed data, i.e., cached answers to queries posed over other nodes.

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 Piazza is a *peer data management system* (PDMS). It provides an infrastructure on which to build applications of the Semantic Web, which essentially share the vision of large-scale data sharing systems on the Web. So there are several common properties that could clarify PDMS nature:

- PDMS manages structured information like peer-to-peer overlay network
- Has no central logical mediated schema
- Scalable
- Has no central administration.

It is important to notice while a PDMS is based on a peer-to-peer architecture, it is significantly different from a P2P file-sharing system. In particular, joining a PDMS is inherently a more heavyweight operation than joining a P2P file-sharing system, since some semantic relationships need to be specified. Its initial architecture focuses on applications where peers are likely to stay available the majority of the time, but in which peers should be able to join (or add new data) very easily. It could be a spectrum of PDMS applications, ranging from more ad-hoc sharing scenarios to ones in which the membership changes less frequently or is restricted due to security or consistency requirements [4].

2.2 Data Sharing and Mediation

There are some methods for specifying common terminology or common scheme for data representation. 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 (Fig.1a), or the mediated schema can be described as a set of views over the data sources, *global-as-view* or GAV (Fig.1b). 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, macroexpand) the views into the query so it refers to the underlying data sources. The LAV approach requires more sophisticated query reformulation algorithms, because we need to use the views in the *reverse* direction.

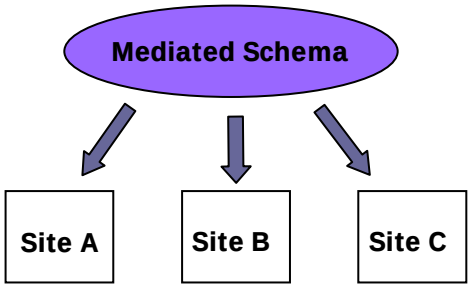

Fig.1a: Local-As-View

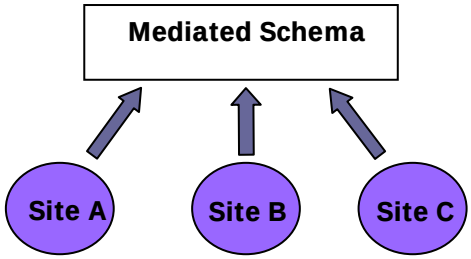

Fig.1b: Global-As-View

When we map between two sites, our mappings, like views, will be directional. Main 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.
- Second, to provide interoperability between data sources, we must map between both their domain structures and their document structures.

Data management practitioners often prefer to exchange data through local point-to-point data translations, rather than mapping to common mediated schemas or ontologies. 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.

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 OWLontologies.

- **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. See the following example (Fig.2).

Suppose we want to map between two sites. Suppose the target contains books with nested authors; the source contains authors with nested publications. On the right is illustration of partial schemas for these sources, using a format in which indentation illustrates nesting and a * suffix indicates "0 or more occurrences of....".

Target (S1):

pubs
book*
title
author*
name
publisher*
name

Source (S2):

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

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. 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. 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.

Fig.2: XML/XML translation example

- **XML-to-RDF mappings:** There are two issues when mapping between XML to RDF/OWL data:

  - Expressive power. 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.

  - Document structure. How to rebuild the appropriate document structure when transferring data from the OWLontology into XML. In fact, the two different structures of XML nodes could be mapped to the *same* RDF.

2.3 System Application

To validate approach, a small semantic web application was implemented in Piazza. Piazza system consists of two main components. The query reformulation engine takes a query posed over a node, and it uses the query reformulation algorithm in order to chain through the semantic mappings and output a set of queries over the relevant nodes. Query evaluation engine is based on the Tukwila XML Query Engine, and it has the important property that it yields answers as the data is streaming in from the nodes on the network.

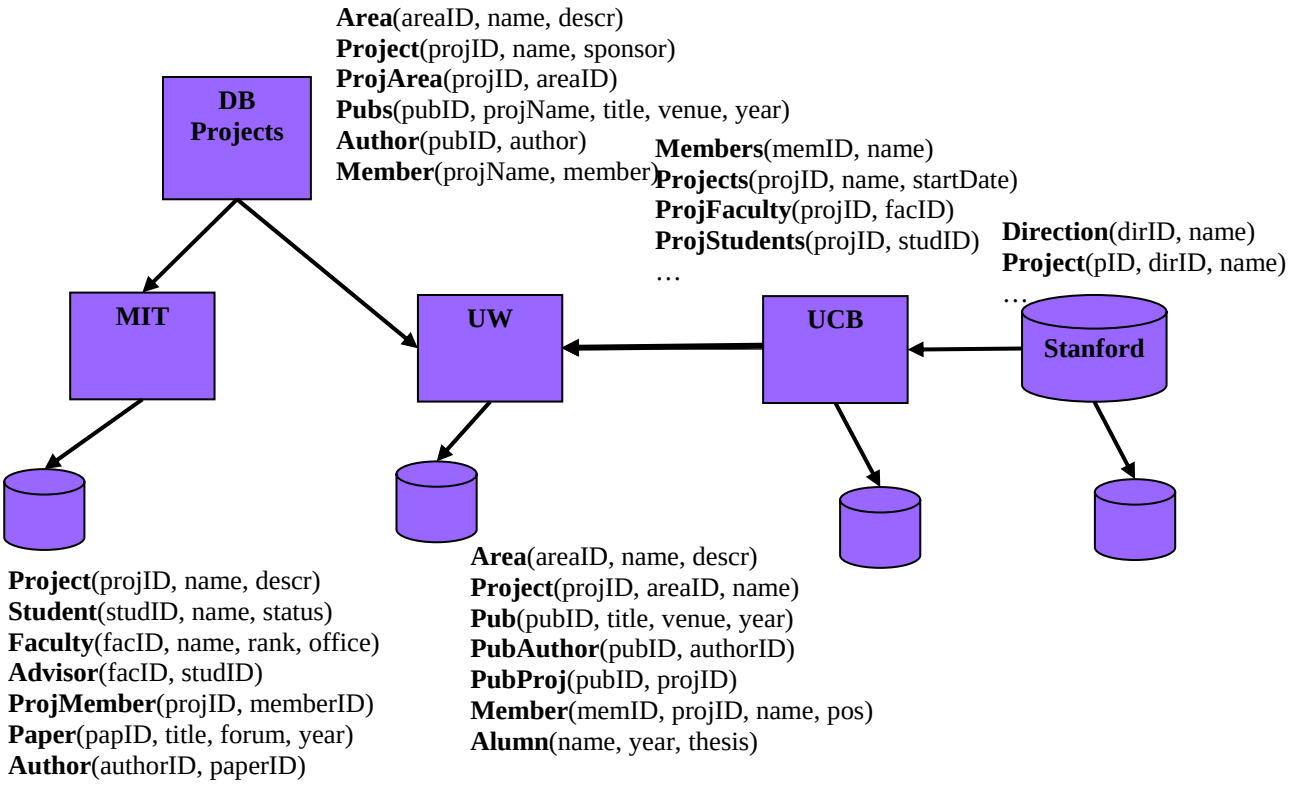

Fig. 3: Fragment of the topology of the DB-Research Piazza application

Prototype relates 15 nodes (see fragment of topology Fig.3) concerning different aspects of the database research field. 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 etc.). 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.

3 IMPLEMENTATION DETAILS

This section describes in details some features of PIAZZA system. It concerned on mapping language and query answering algorithm. The best way to explain specific abilities of proposed solutions is to illustrate them with examples.

3.1 The Mapping Language

Mappings play two roles:

- **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;

- **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.

When we map between two sites, our mappings, like views, will be directional. In general, 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.

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, authors found XQuery inappropriate as a mapping language for the following reasons:

- 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.

- 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.

- XQuery is an extremely powerful query language and as a result some aspects of the language make it difficult or even impossible to reason about.

Current 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.

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 (Fig. 4) for the sites in previous example (Fig.2).

```
<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>
```

Fig.4: Mapping language basic example for sites S1 and S2 from Fig.2

Where we make variable references within braces and delimit query expression annotations by colon. 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.

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* (Fig.5). All of their attributes and element content will be combined.

```
<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>
```

Fig.5: Mapping language example with piazza:id attribute for sites S1 and S2 from Fig.2

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 sub-elements (but not author names). Hence, in the target we will see all authors nested under each book entry.

But 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* (Fig.6). 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.

```
<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>
```

Fig.6: Mapping language example using *properties* for sites S1 and S2 from Fig.2

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.

To complete the discussion about relationship to data integration, let me briefly discuss how the mapping language relates to the LAV and GAV formalisms. In current 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 following important features of this language would require LAV definition in the relational setting:

- It can map data sources to the target schema even if the data sources are missing attributes or sub-elements required in the source schema. Hence, we can support the situation where the source schema is a projection of the target.

- Mapping language support the notion of data source *properties*, which essentially describes scenarios in which the source schema is a selection on the target schema. Hence, current mapping 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.

3.2 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.

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. 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. This approach is well developed and doesn't introduce any difficulties.
  Example Query $QP_1(x)$:- DbResearcher(x);
  Query Composition: DbResearcher(x)⊇Researcher(x),Area(x,DB)
  ➔$QP_2(x)$⊇Researcher(x),Area(x,DB)

- 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. This problem is known as the problem of answering queries using views [1], and is conceptually much more challenging.
  Example $QP_1(x)$:- DbResearcher(x);
  Query Rewriting: DbResearcher(x),Office(x,DBLab)=DbLabMember(x)
  ➔$QP_3(x)$⊇DbLabMember(x)

The problem is well understood for the case of relational queries and views, and authors describe an algorithm that applies to the XML setting. The key challenge of XML is the nesting structure of the data (and hence of the query) – relational data is flat.

3.2.1 Query Representation

Current algorithm operates over a graph representation of queries and mappings. Suppose for two sources S1 and S2 (Fig.7a) we are given the following XQuery for all advisees of Ullman, posed over source S1 (Fig.7b). Note that the result element in the query simply specifies the root element for the resulting document.

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.	**Target (S1):** **people** **faculty*** **name** **advisee*** **student***	**Source (S2):** **people** **faculty*** **student** **name** **advisor***

Fig.7a: Query representation example. Data sources S1 and S2.

```
<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>
```

Fig.7b: Query posed over S1.

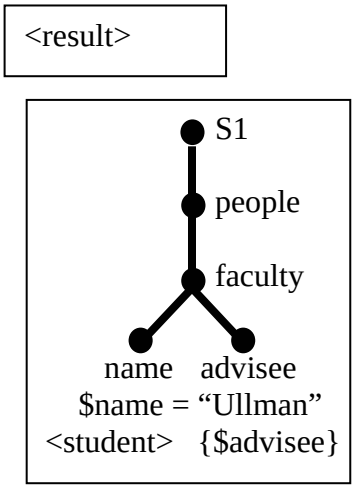

Fig.7c: Query tree pattern.

We could also represent mapping both literally and graphically (Fig.8). Each box in the figure corresponds to a query block, and indentation indicating the nesting structure. With each block associated the following constructs that are manipulated by 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. 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 (Fig.7c). 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.

```
<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>
```

Fig.8a: Schema mapping between S1 and S2

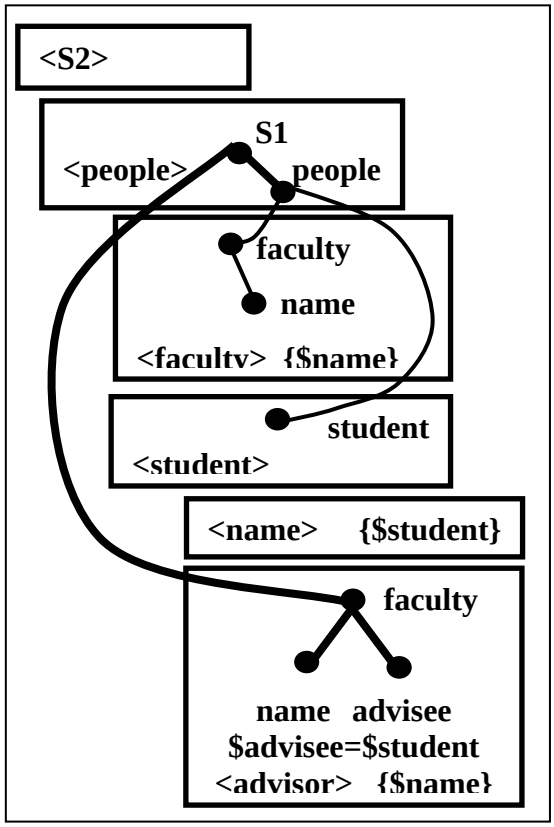

Fig.8b: Tree pattern for schema mapping between S1 and S2

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. It enables transformations in which data from one source becomes schema information in another. In our query graph (Fig.7c), 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. 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 Fig.8a and Fig.8b show the mapping between the following schemas and its graph representation.

3.2.2 The Rewriting Algorithm

Rewriting algorithm makes the following simplifying assumptions about the queries and the mappings.

- The query over the target schema contains a single nontrivial 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.

- 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). The variable *$people* (Fig.8a) is bound to an element; variables *$name* and *$student* are bound to values.

- Queries are evaluated under a set semantics. In other words, we assume that duplicate results are eliminated in the original and rewritten query.

- 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 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. The darker lines in the representation of the schema mapping denote the tree pattern of the query (Fig.7c) and its corresponding form in the mapping (Fig.8b). 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. 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 sub-blocks, so we cannot simply extend the tree pattern. Fortunately, the <*advisor*> block enforces equality 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.

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

Fig.9: Reformulated query

In our case, the query predicate can be reformulated in terms of the variables bound by the replacement tree pattern as follows: *$advisor="Ullman"*. The resulting rewritten query in our example is the following (Fig.9). More detailed formalization of current algorithm is available in [3, 10].

4 CONCLUSIONS AND FUTURE WORK

Here is some experimental results on small number of 15 nodes (Table 1). The second and third columns 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). They observe that 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.

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

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

Among major contributions authors mentioned:

- Mapping language for mapping between sets of XML source nodes with different document structures
- Architecture that uses the transitive closure of mappings to answer queries
- Algorithm for query answering over this transitive closure of mappings, which is able to follow mappings in both forward and reverse directions

And there are several directions of the future work:

- More efficient reformulation algorithm
- Semantic network analysis – eliminate redundant mappings and inconsistent mappings
- Query caching to speed up query evaluation

At a more conceptual level Piazza paves the way for a fruitful combination of data management and knowledge representation techniques in the construction of the Semantic Web. Techniques offered in Piazza are *not* a replacement for rich ontologies and languages for mapping between ontologies. Its 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. Finally, we note that Piazza is a component of the larger Revere Project [2] that attempts to address the entire life-cycle of content creation on the Semantic Web.
Just as older database systems suddenly became compatible by adopting a consistent relational model, so unstructured web data, or XML-schema definitions can,

essentially, also adopt a relational model, allowing significantly more power to brought to bear on solving data-modeling problems.

There are still a lot of questions about efficiency if network is quite large, or how we will manage cyclic and redundant paths for algorithm? Analysis of mapping networks for information loss also required. It is also interesting to study Piazza's utility in investigating strategies for caching and replicating data and mappings for reliability and performance.

5 REFERENCES

1. Alon Y. Halevy. Answering queries using views: A survey. VLDB Journal 10, 2001.

2. Alon Halevy, Oren Etzioni, AnHai Doan, Zachary Ives, Jayant Madhavan, Luke McDowell, Igor Tatarinov. Crossing the Structure Chasm. In Proc. of CIDR, 2003.

3. Alon Y. Halevy, Zachary G. Ives, Dan Suciu, Igor Tatarinov. Schema Mediation for Large-Scale Semantic Data Sharing. To appear, VLDB Journal 2003.

4. Alon Y. Halevy, Zachary G. Ives, Dan Suciu, Igor Tatarinov. Schema Mediation in Peer Data Management Systems. In Proc. of ICDE, March 2003.

5. Berners-Lee T., Hendler J., and Lassila O. The semantic web. Scientific American, May 2001.

6. Boag S., Chamberlin D., Fernandez M. F., Florescu D., Robie J., Simeon J. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, 12 November 2003. W3C working draft.

7. Decker S., van Harmelen F., Broekstra J., Erdmann M., Fensel D., Horrocks I., Klein M., Melnik S. The Semantic Web - on the respective Roles of XML and RDF. 2000.

8. Fuhr N., Weikum G. Classification and Intelligent Search on Information in XML. In TCDE, March 2002 Vol. 25 No. 1.

9. Hendler J., Berners-Lee T., Miller E.: Integrating Applications on the Semantic Web. Journal of the Institute of Electrical Engineers of Japan, Vol. 122(10), October, 2002.

10. Igor Tatarinov, Zachary Ives, Jayant Madhavan, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin (Luna) Dong, Yana Kadiyska, Gerome Miklau, Peter Mork. The Piazza Peer Data Management Project. SIGMOD Record, Vol. 32, No. 3, September 2003.

11. Madhan Arumugam, Amit Sheth, and I. Budak Arpinar: Towards Peer-to-Peer Semantic Web. A Distributed Environment for Sharing Semantic Knowledge on the Web. http://lsdis.cs.uga.edu.