The Piazza Peer Data Management System

Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, Igor Tatarinov

Alon Y. Halevy, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov are with Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350 Email: {alon, jayant, pmork, suciu, igor}@cs.washington.edu

Zachary G. Ives is with the Department of Computer and Information Science, University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104-6389, Email: zives@cis.upenn.edu

Abstract

Intuitively, data management and data integration tools should be well-suited for exchanging information in a semantically meaningful way. Unfortunately, they suffer from two significant problems: they typically require a comprehensive schema design before they can be used to store or share information, and they are difficult to extend because schema evolution is heavyweight and may break backward compatibility. As a result, many small-scale data sharing tasks are more easily facilitated by non-database-oriented tools that have little support for semantics. The goal of the peer data management system (PDMS) is to address this need: we propose the use of a decentralized, easily extensible data management architecture in which any user can contribute new data, schema information, or even mappings between other peers' schemas. PDMSs represent a natural step beyond data integration systems, replacing their single logical schema with an interlinked collection of semantic mappings between peers' individual schemas. This paper describes several aspects of the Piazza PDMS, including the schema mediation formalism, query answering and optimization algorithms, and the relevance of PDMSs to the Semantic Web.

Index Terms

Peer data management, data integration, schema mediation, web and databases

I. Introduction

While databases and data management tools excel at providing semantically rich data representations and expressive query languages, they have historically been hindered by a need for significant investment in design, administration, and schema evolution. Schemas must generally be predefined in comprehensive fashion, rather than evolving incrementally as new concepts are encountered; schema evolution is typically heavyweight and may "break" existing queries. As a result, many people find that database techniques are obstacles to lightweight data storage and large-scale data sharing tasks, rather than facilitators. They resort to simpler and less expressive tools, ranging from spreadsheets to text files, to store and exchange their data. This provides a simpler administrative environment (although some standardization of terminology and description is always necessary), but with a significant cost in functionality. Worse, when a lightweight repository grows larger and more complex in scale, there no easy migration path to a semantically richer tool.

Conversely, the strength of HTML and the World Wide Web has been easy and intuitive

support for ad hoc extensibility — new pages can be authored, uploaded, and quickly linked to existing pages. However, as with flat files, the Web environment lacks rich semantics. Initially, that shortcoming spurred a movement towards XML, which allows data to be semantically tagged. Unfortunately, XML carries many of the same requirements and shortcomings as data management tools: for rich data to be shared among different groups, all concepts need to be placed into a common frame of reference. XML schemas must be completely standardized across groups, or mappings must be created between all pairs of related data sources.

More recently, the desire to complement the Web with more semantics spurred the vision of the Semantic Web [1], which calls for sharing of structured data at Web scale, with queries spanning large numbers of Web sites. Much of the research focus on the Semantic Web is based on treating the Web as a knowledge base defining concepts 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. The best-known example is RDF and the languages that build upon it: RDF Schema and OWL [2]. While there has been much investigation of how to define the meaning of data locally, the issues of large-scale data sharing have yet to be addressed.

Data integration systems have been proposed as a partial solution to the problem of large-scale data sharing [3], [4], [5], [6], [7], [8], [9], [10]. These systems support rich queries over large numbers of autonomous, heterogeneous data sources by exploiting the semantic relationships between the different sources' schemas. An administrator defines a global *mediated schema* for the application domain and specifies semantic mappings between the sources and the mediated schema. We get the strong semantics needed by many applications, and data sources can evolve independently — and, it would appear, relatively flexibly. Yet in reality, the mediated schema, the integrated part of the system that actually facilitates all information sharing, becomes a bottleneck in the process. Mediated schema design must be done carefully and globally; data sources cannot change significantly or they might violate the mappings to the mediated schema; concepts can only be added to the mediated schema by the central administrator. The ad hoc extensibility of the Web is missing, and as a result, data integration systems provide limited support for large-scale data sharing.

We believe that there is a clear need for a new class of data sharing tools that preserves semantics and rich query languages, but which facilitates ad hoc, decentralized sharing and

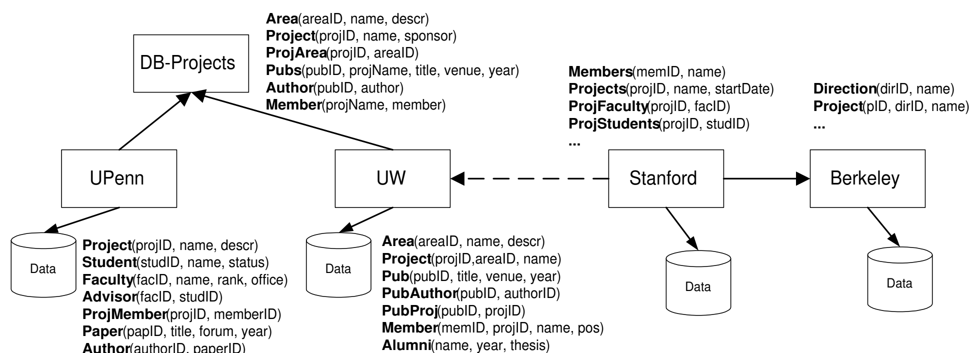

Fig. 1. A PDMS for the database research domain. Arrows indicate that there is (at least a partial) mapping between the relations of the peer schemas. The figure illustrates how two semantic networks can be joined by establishing a single mapping between a pair of peers (UW and Stanford in this case).

administration of data and defining of semantic relationships. Every participant in such an environment should be able to contribute new data and relate the data to existing concepts and schemas, define new schemas that others can use as frames of reference for their queries, or define new relationships between existing schemas or data providers. We believe that a natural implementation of such a system will be based on a peer-to-peer architecture, and hence call such a system a *peer data management system* (PDMS). The vision of a PDMS is to blend the extensibility of the HTML Web with the semantics of data management applications.

Example 1.1: The extensibility of a PDMS can best be illustrated with a simple example. Figure 1 illustrates a peer data management system for supporting a web of database research-related data. This will be a running example throughout the paper so we only describe the functionality here. Unlike a hierarchy of data integration systems or mediators, a PDMS supports any arbitrary network of relationships between peers. The true novelty lies in the PDMS's ability to exploit transitive relationships among peers' schemas. The figure shows that two semantic networks can be fully joined together with only a few mappings between similar members of each semantic network (in our example, we only required a single mapping). The new mapping from Stanford to UW enables any query at any of the five peers to access data at *all* other peers through transitive evaluation of semantic mappings. Importantly, the mappings can be defined between the most similar nodes in the two semantic networks; this is typically much easier than

attempting to map a large number of highly dissimilar schemas into a single mediated schema (as in conventional data integration). □

This paper describes the main contributions of the Piazza PDMS that we have been building at the University of Washington. Most of the data integration literature is based on the relational data model, largely because it is the simplest and cleanest model in which to define properties and analyze complexity; accordingly, we begin our description of the Piazza system using the relational context. Section II presents the logical model underlying, and it defines the problem of semantic mediation in a PDMS. Section III outlines some of the theoretical results concerning query answering and mediation in a PDMS. Section IV describes a query answering algorithm for Piazza, and explains some of the current research directions we are pursuing for query optimization. One of these directions is the development of techniques for *mapping composition*, which we explain in Section V.

Although the relational model is ideal for defining properties of the PDMS, in the real world XML is the most useful representation for sharing semantically rich data: most relational and semistructured data sources export to XML, and XML is also used for the RDF data format that has been the focus of efforts in the Semantic Web. Thus our actual Piazza system uses XML as the data model. In Section VII we describe Piazza's XML support and explain how it can provide the infrastructure for supporting both XML- and RDF-based Semantic Web applications.

II. Logical Model of the PDMS

We begin with a description of the logical model underlying a PDMS (defined using the relational data model; Section VI details how this definition changes for XML data). Informally, a PDMS consists of a set of data sources (a.k.a. peers), and they are related through *semantic mappings*. A PDMS can be viewed as a strict generalization of data integration systems. In our discussion, we assume a relational data model, we focus on select-project-join queries with a set semantics, and we use the notation of conjunctive queries. In this notation, joins are specified by multiple occurrences of the same variable. Unless explicitly specified, we assume queries do not contain comparison predicates (e.g., $\neq$, $<$). Views refer to named queries.

We assume that each peer defines its own relational *peer schema* whose relations are called *peer relations*; a query in a PDMS will be posed over the relations from a specific peer schema. Without loss of generality we assume that relation and attribute names are unique to each peer.

Peers may also contribute data to the system, in the form of *stored relations*. Stored relations are analogous to data sources in a data integration system: all queries in a PDMS will be reformulated strictly in terms of stored relations that may be stored locally or at other peers. (Note that not every peer needs to contribute stored relations to the system, as some peers may strictly serve as logical mediators to other peers.) We assume that the names of stored relations are distinct from those of peer relations.

Example 2.1: In our example PDMS in Figure 1, only peer relations are shown. The lines between peers indicate that there is a mapping (described later) between the two peers.

Stored relations containing actual data are provided by the universities: the UPenn, UW, Stanford, and Berkeley peers. DB-Projects is a virtual peer that provides a uniform view over the domain. Stanford and Berkeley, as neighboring universities, came to an agreement to map their schemas directly. The flexibility of the PDMS (due to its ability to evaluate transitive relationships between schemas) becomes evident when two PDMSs are joined. In our example, once a mapping between the Stanford-Berkeley PDMS and the UPenn-UW-DBProjects PDMS is established, queries over any of the five peers will be able to access *all* of the stored relations. □

Note that our approach can support evolving schemas very naturally. A new schema version can be treated as an additional peer schema. In general, the new version is likely to be very similar to the previous version making the problem of specifying a mapping between the versions rather easy. In addition, the resulting mapping is likely to be very accurate.

A. Mapping Language for PDMS

Obviously, the power of the PDMS lies in its ability to exploit semantic mappings between peer and stored relations. In particular, there are two types of mappings that must be considered: mappings describing the data within the stored relations (generally with respect to one or more peer relations), and mappings between peer schemas. At this point it is instructive to recall the formalisms used in the context of data integration systems, since we build upon them in defining our mapping description language.

1) Mappings in Data Integration: Data integration systems provide a uniform interface to a multitude of data sources through a logical, *mediated* schema. Mappings are established between the mediated schema and the relations at the data sources, forming a two-tier architecture in which

queries are posed over the mediated schema and evaluated over the underlying source relations. A data integration system can be viewed as a special case of a PDMS.

Two main formalisms have been proposed for schema mediation in data integration systems. In the first, called *global-as-view* (GAV) [11], [3], [4], [5], the relations in the mediated schema are defined as views over the relations in the sources. In the second, called *local-as-view* (LAV) [6], [7], [8], the relations in the sources are specified as views over the mediated schema. In fact, in many cases the source relations are said to be *contained* in a view over the mediated schema, as opposed to being exactly equal to it. We illustrate both below.

Example 2.2: The DB-Projects' Member peer relation, which mediates UPenn and UW peers, may be expressed using a GAV-like definition. The definition specifies that Member in DB-Projects is obtained by a union over the UPenn and UW schemas. Note in our examples, that peer relations are named using a peer-name:relation-name syntax:

$$\begin{aligned}
\mathsf{DBProjects : Member(projName, member)} \; :- \; & \mathsf{UPenn : Student(sid, member, _)}, \\
& \mathsf{UPenn : ProjMember(pid, sid)}, \\
& \mathsf{UPenn : Project(pid, projName, _)} \\
\mathsf{DBProjects : Member(projName, member)} \; :- \; & \mathsf{UPenn : Faculty(sid, member, _)}, \\
& \mathsf{UPenn : ProjMember(pid, sid)}, \\
& \mathsf{UPenn : Project(pid, projName, _)} \\
\mathsf{DBProjects : Member(projName, member)} \; :- \; & \mathsf{UW : Member(,pid, member, _)}, \\
& \mathsf{UW : Project(pid, _, projName)}
\end{aligned}$$

We may use the LAV formalism to specify the UW peer relations as views over mediated DB-Projects relations. This formalism is especially useful when there are many data sources that are related to a particular mediated schema. In such cases, it is more convenient to describe the data sources as views over the mediated schema rather than the other way around. In our scenario, DB-Projects may eventually mediate between many universities, and hence LAV is appropriate for future extensibility. The following illustrates an LAV mapping for UW:

$$\begin{aligned}
\mathsf{UW : Project(projID, areaID, projName)} \subseteq \; & \mathsf{DBProjects : Project(projID, projName)}, \\
& \mathsf{DBProjects : ProjArea(projID, areaID)}
\end{aligned}$$

□

The fundamental difference between the two formalisms is that GAV specifies how to extract tuples for the mediated schema relations from the sources, and hence query answering amounts to view unfolding. In contrast, LAV is source-centric, describing the contents of the data sources. Query answering requires algorithms for answering queries using views [12], but in exchange LAV provides greater extensibility: the addition of new sources is less likely to require a change to the mediated schema.

Before proceeding, we note that all the languages we discuss here are for mapping *schemas*, rather than *data values*. For example, it is very common that a person or company name may appear differently in two data sources. The topic of object identification is currently a very active area of research and beyond the scope of this paper. In commercial systems (e.g., [13]) the problem has usually been addressed by *concordance tables*, which are binary tables relating the different ways of referring to the same object.

2) Mappings for PDMS: We now present the relational-model version of $\mathcal{PPL}$ the Piazza Peer Language. Our goal in $\mathcal{PPL}$ is to preserve the features of both the GAV and LAV formalisms, but to extend them from a two-tiered architecture to our more general network of interrelated peer and source relations. Semantic relationships in a PDMS will be specified between pairs (or small sets) of peer (and optionally source) relations. Ultimately, a query over a given peer relation may be reformulated over source relations on any peer in the transitive closure of peer mappings.

First, we formally define our two types of mappings, which we refer to as storage descriptions and peer mappings.

Storage descriptions: Each peer contains a (possibly empty) set of *storage descriptions* that specify the data stored at a peer by relating its stored relations to its peer relations. Formally, a storage description of the form $A : R = Q$, where Q is a conjunctive query over the schema of peer A and R is a stored relation at the peer. The description specifies that A stores in relation R the result of the query Q over its schema.

In many cases the data that is stored is not *exactly* the definition of the view, but only a subset of it. As in the context of data integration, this situation arises often when the data at the peer

may be incomplete (this is often called the *open-world assumption* [14]).[1] Hence, we also allow storage descriptions of the form $A : R \subseteq Q$. We call the latter descriptions *containment* (or inclusion) descriptions versus *equality* descriptions.

Example 2.3: A storage description might relate the stored students relation at peer UPenn to the peer relations:

$$\begin{aligned} \mathsf{UPenn} : \mathsf{students}(\mathsf{sid}, \mathsf{name}, \mathsf{advisor}) \subseteq\ & \mathsf{UPenn} : \mathsf{Student}(\mathsf{sid}, \mathsf{name}, _), \\ & \mathsf{UPenn} : \mathsf{Advisor}(\mathsf{sid}, \mathsf{fid}), \\ & \mathsf{UPenn} : \mathsf{Faculty}(\mathsf{fid}, \mathsf{advisor}, _, _) \end{aligned}$$

This storage description says that UPenn:students stores a subset of the join of Student, Advisor and Faculty, which reflects the fact that UPenn:students is unlikely to contain information about *all* students in the world; it will probably contain data on "local" students only. Hence, if a UPenn:Affiliation peer relation with the corresponding semantics was available, the above storage description could be specified more precisely as follows:

$$\begin{aligned} \mathsf{UPenn} : \mathsf{students}(\mathsf{sid}, \mathsf{name}, \mathsf{advisor}) =\ & \mathsf{UPenn} : \mathsf{Student}(\mathsf{sid}, \mathsf{name}, _), \\ & \mathsf{UPenn} : \mathsf{Advisor}(\mathsf{sid}, \mathsf{fid}), \\ & \mathsf{UPenn} : \mathsf{Faculty}(\mathsf{fid}, \mathsf{advisor}, _, _), \\ & \mathsf{UPenn} : \mathsf{Affiliation}(\mathsf{sid}, '\mathsf{UPenn}') \end{aligned}$$

□

Peer mappings: *Peer mappings* provide semantic glue between the schemas of different peers. We have two types of peer mappings in $\mathcal{PPL}$. The first are *inclusion* and *equality* mappings (similar to the concepts for storage descriptions). In the most general case, these mappings are of the form $Q_1(\bar{A}_1) = Q_2(\bar{A}_2)$, (or $Q_1(\bar{A}_1) \subseteq Q_2(\bar{A}_2)$ for inclusions) where Q_1 and Q_2 are conjunctive queries with the same arity and $\bar{A}_1$ and $\bar{A}_2$ are *sets* of peers. Query Q_1 (Q_2) can refer to any of the peer relation in $\bar{A}_1$ ($\bar{A}_2$, resp.). Intuitively, such a statement specifies a semantic mapping by stating that evaluating Q_1 over the peers $\bar{A}_1$ will always produce the same answer (or a subset in the case of inclusions) as evaluating Q_2 over $\bar{A}_2$. Note that since $\mathcal{PPL}$ allows queries on both sides of the equation, we can accommodate both GAV and LAV-style mappings

[1]Sometimes it may be possible to describe the exact contents of a data source with a more refined query, but very often this cannot be done.

(and thus we can express any of the types of mappings used in data integration. This approach is also known as GLAV [12].

The second kind of peer mappings are called *definitional mappings*. They are datalog rules whose relations (both head and body) are peer relations, i.e., the body cannot contain a query. Formally, as long as a peer relation appears only once in the head of a definitional description, such mappings can be written as equalities. We include definitional mappings in order to obtain the full power of GAV mappings. We distinguish definitional mappings for the following reasons:

- As we show in Section III, the complexity of answering queries when equality mappings are restricted to being definitional is more attractive than the general case, and
- Definitional mappings can easily express disjunction: e.g., $P(x) : -P_1(x)$ and $P(x) : -P_2(x)$ means that P is the union of P_1 and P_2 (while the pair of mappings $P(x) = P_1(x)$ and $P(x) = P_2(x)$ means that P, P_1 and P_2 are equal).

In summary, a PDMS N is specified by a set of peers $\{P_1, ..., P_n\}$, a set of peer schemas $\{S_1, ..., S_m\}$ and a mapping function from peers to schemas, a set of stored relations $\mathcal{R}_i$ at each peer P_i, a set of peer mappings $\mathcal{L}_N$, and a set of storage descriptions $\mathcal{D}_N$. The storage descriptions and peer mappings provided by a peer P_i may reference stored or peer relations defined by other peers; thus, any peer can extend another peer's relations or use its data.

B. Semantics of $\mathcal{PPL}$

Given the peer and stored relations, their mappings, and a query over some peer schema, the PDMS needs to answer the query using the data from the stored relations. To formally specify the problem of query answering, we need to define the semantics of queries. We show below how the notion of *certain answers* [14] from the data integration context can be generalized to our context. Our goal is to formally define what is the set of answers to a conjunctive query Q posed over the relations of a peer A. The challenge arises because the peer schemas are virtual; in fact, some data may only exist partially, if at all, in the system.

Formally, we assume that we are given a PDMS N and an instance for the stored relations, D, i.e., a set of tuples $D(R)$ for each stored relation $R \in (\mathcal{R}_1 \cup \ldots \cup \mathcal{R}_n)$. A *data instance* I for a PDMS N is an assignment of a set of tuples to each relation in each peer (both the peer and stored relations). We denote by $I(R)$ the set of tuples assigned to the relation R by I, and we denote by $Q(I)$ the result of computing the query Q over the extensional data in I.

To define certain answers, we will consider only the data instances that are consistent with the specification of N:

Definition 2.1: (**Consistent data instance**) A data instance I is said to be *consistent* with a PDMS N and an instance D for N's stored relations if:

- For every storage description in $\mathcal{D}_N$, if it is of the form $A : R = Q_1$ ($A : R \subseteq Q_1$), then $D(R) = Q_1(I)$ ($D(R) \subseteq Q_1(I)$).
- For every peer description in $\mathcal{L}_N$:
  - if it is of the form $Q_1(\mathcal{A}_1) = Q_2(\mathcal{A}_2)$, then $Q_1(I) = Q_2(I)$,
  - if it is of the form $Q_1(\mathcal{A}_1) \subseteq Q_2(\mathcal{A}_2)$, then $Q_1(I) \subseteq Q_2(I)$,
  - if it is a definitional description whose head predicate is p, then let $r_1, \ldots, r_m$ be all the definitional mappings with p in the head, and let $I(r_i)$ be the result of evaluating the body of r_i on the instance I. Then, $I(p) = I(r_1) \cup \ldots \cup I(r_m)$.

□

Intuitively, a data instance I is consistent with N and D if it describes *one* possible state of the world (i.e., extension for each of the peer relations) that is allowable given the data and peer mappings and D. We define the certain answers to be those that hold in *every* possible consistent data instance:

Definition 2.2: (**Certain answers**) Let Q be a query over the schema of a peer A in a PDMS N, and let D be an instance of the stored relations of N. A tuple $\bar{a}$ is a *certain answer* to Q if $\bar{a}$ is in $Q(I)$ for every data instance that is consistent with N and D. □

Note that in the last bullet of Definition 2.1 we did not require that the extension of p be the least-fixed point model of the datalog rules. However, since we defined certain answers to be those that hold for *every* consistent data instance, we actually do get the intuitive semantics of datalog for these mappings.

Query answering Now we can define the query answering problem for PDMS: given a PDMS N, an instance of the stored relations D and a query Q, find all certain answers of Q.

III. Complexity of Query Answering

Before we present an algorithm for answering queries in Piazza (the focus of Section IV), it is important to understand how tractable query answering is in the PDMS model. In this section

we briefly review some basic results relating to the complexity of finding certain answers with our PDMS mapping language (full details are given in [15]).

The computational complexity of finding all certain answers is well understood for the data integration context with a two-tiered architecture of a mediator and a set of data sources [14]. Here we show the complexity of query answering in the global context of a PDMS, when the data integration formalisms are used locally. The complexity will depend on the restrictions we impose on peer mappings in $\mathcal{PPL}$.

The focus of our analysis is on data complexity — the complexity of query answering in terms of the total size of the data stored in the peers. Typically, the complexity of query answering is either polynomial, Co-NP-hard but decidable, or undecidable. In the polynomial case, it is possible to find a *reformulation* of the query into a query that refers only to the stored relations, and this reformulation is then optimized and executed. In the latter two cases, it is not possible to find *all* certain answers efficiently; but it is possible to develop an efficient reformulation algorithm that does not provide all certain answers, but that *only* returns certain answers.

A key result: Cyclicity of peer mappings plays a very significant role in the complexity of answering queries.

Definition 3.1: (**Acyclic peer mappings**) A set $\mathcal{L}$ of *inclusion* peer mappings in $\mathcal{PPL}$, is said to be acyclic if the following directed graph is acyclic. The graph contains a node for every peer relation mentioned in $\mathcal{L}$. There is an arc from the node corresponding to R to the node corresponding to S if there is a peer description in $\mathcal{L}$ of the form $Q_1(\bar{\mathcal{A}}_1) \subseteq Q_2(\bar{\mathcal{A}}_2)$ where R appears in Q_1 and S appears in Q_2. □

The following theorem characterizes two extreme cases of query answering in PDMS:

Theorem 3.1: Let N be a PDMS specified in $\mathcal{PPL}$.

1) The problem of finding all certain answers to a conjunctive query Q, for a given PDMS N, is undecidable.
2) If N includes only inclusion peer mappings and storage descriptions and the peer mappings are acyclic, then a conjunctive query can be answered in polynomial time data complexity.

The proof of this Theorem appears in [15]. The difference in complexity between the first and second bullets shows that the presence of cycles is the culprit for achieving query answerability in a PDMS (note that equalities automatically create cycles). In a sense, the theorem also establishes a limit on the arbitrary combination of the formalisms of LAV and GAV.

The second bullet points out a powerful schema mediation language for PDMS for which query answering can be done efficiently. It shows that LAV and GAV style reformulations can be chained together arbitrarily, and extends the results of [16], which combined one level of LAV followed by one level of GAV.

In [15] we also show how the complexity is affected by allowing comparison predicates, identify a few cases where query answering in cyclic PDMS is tractable, and establish the complexity of the *PDMS consistency* problem. In summary, while the arbitrary use of the data integration formalisms in a PDMS, query answering is undecidable, we have identified a large and useful subset of $\mathcal{PPL}$ for which query answering is tractable. Our results are based on an open-world assumption [14], in which peers have incomplete rather than full information. The closed-world assumption, which is necessary for supporting negation in mappings and queries, is known to make the problem of finding all certain answers much harder (co-NP hard in the size of the data [14]) — even for two peers (and, in fact, even without negation).

IV. Query Reformulation Algorithm

In this section we describe an algorithm for query reformulation for PDMSs. The input of the algorithm is a set of peer mappings and storage descriptions and a query Q. The output of the algorithm is a query expression Q' that refers to stored relations *only*. To answer Q we need to evaluate Q' over the stored relations. The precise method of evaluating Q' is beyond the scope of this paper, but we note that recent techniques for adaptive query processing [17] are well suited for our context. Furthermore, in this discussion we assume that all the peer mappings are available at a single location, and hence all the reformulation is done in a single place. We are currently investigating methods for distributed query reformulation.

The algorithm is sound and complete in the following sense. Evaluating Q' will always *only* produce certain answers to Q. When all the certain answers can be found in polynomial time (according to Section III), Q' will produce all certain answers. Q' is called the *maximally-contained rewriting* of Q [6]: it is a query over the sources that produces all the answers to Q that are possible from any such query.

Before we describe the algorithm, we first provide some intuition on its working and the challenges it faces. Consider a PDMS in which all peer mappings are definitional (similar to GAV mappings in data integration). In this case, the algorithm is a simple construction of a rule-

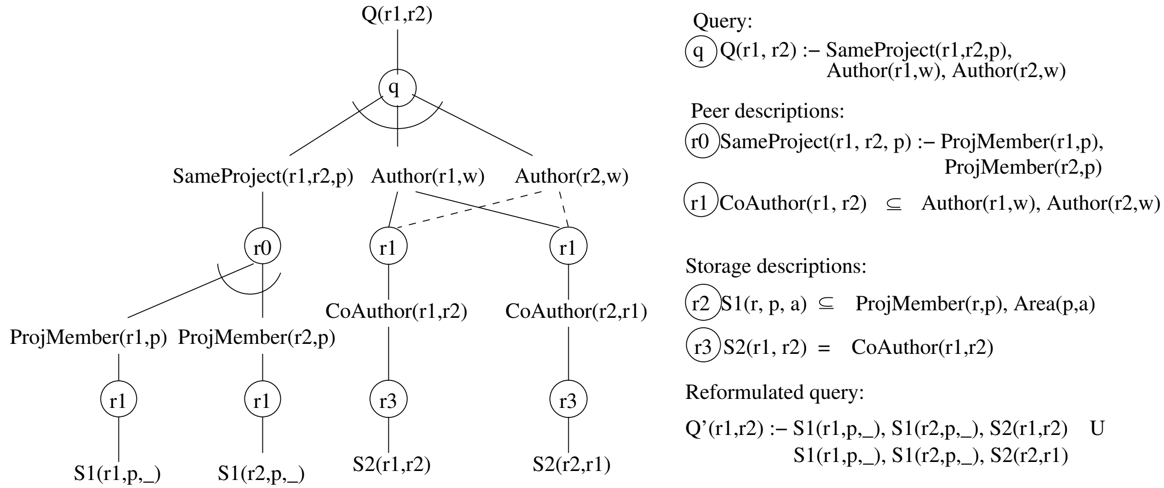

Fig. 2. Reformulation rule-goal tree for Emergency Services domain. Dashed lines represent nodes that are included in the unc label (see text).

goal tree: goal nodes are labeled with atoms of the peer relations, and rule nodes are labeled with peer mappings. We begin by expanding each query subgoal according to the relevant definitional peer mappings in the PDMS. When none of the leaves of the tree can be expanded any further, we use the storage descriptions for the final step of reformulation in terms of the stored relations.

At the other extreme, suppose all peer mappings in the PDMS are inclusions in which the left-hand side has a single atom (similar to LAV mappings in data integration). In this case, we begin with the query subgoals and apply an algorithm for answering queries using views (e.g., [12]). We apply the algorithm to the result until we cannot proceed further, and as in the previous case, we use the storage descriptions for the last step of reformulation.

The first challenge of the complete algorithm is to combine and interleave the two types of reformulation techniques. One type of reformulation replaces a subgoal with a set of subgoals, while the other replaces a set of subgoals with a single subgoal. The algorithm will achieve this by building a rule-goal tree, while it simultaneously marks certain nodes as covering not only their parent node, but also their uncle nodes. We illustrate the algorithm by an example below.

Example 4.1: To illustrate the rule-goal tree[2], Figure 2 shows an example for a simple query. We begin with the query, Q, which asks for researchers who have worked on the same project and

[2]More precisely, we actually build a rule-goal DAG, as illustrated in the example.

also co-authored a paper. Q is expanded into its three subgoals, each of which appears as a goal node. The SameProject peer relation (indicating which researchers work on the same project) is involved in a single definitional peer description (r_0), hence we expand the SameProject goal node with the rule r_0, and its children are two goal nodes of the ProjMember peer relation (each specifying the projects an individual researcher is involved in).

The Author relation is involved in an inclusion peer description (r_1). We expand Author(r1,w) with the rule node r_1, and its child becomes a goal node of the relation CoAuthor. This "expansion" is of different nature because of the LAV-style reformulation. Intuitively, we are reformulating the Author(r1,w) subgoal to use the left-hand side of r_1. The right-hand side of r_1 includes two subgoals of Author (with the appropriate variable patterns), so we also mark r_1 as covering its *uncle* node. (In the figure, this annotation is indicated by a dashed line.) Since the peer relation Author is involved in a single peer description, we do not need to expand the subgoal Author(r2,w) any further. Note, however, that we must apply description r_1 a second time with the head variables reversed, since CoAuthor may not be symmetric (because it is $\subseteq$ rather than $=$).

At this point, since we cannot reformulate the peer mappings any further, we consider the storage descriptions. We find stored relations for each of the peer relations in the tree (S_1 and S_2), and produce the final reformulation. Reformulations of peer relations into stored relations can also be either in GAV or LAV style. In this simple example, our reformulation involves only one level of peer mappings, but in general, the tree may be arbitrarily deep. □

Optimizations

The second challenge we face is that the rule-goal tree may be huge. First, the tree may be very deep, because it may need to follow any path through semantically related peers. Second, the branching factor of the tree may be large because data is replicated at many peers. Hence, it is crucial that we develop effective methods for pruning the tree and for generating first solutions quickly. It is important to emphasize that while many sophisticated methods have been developed for constructing rule-goal trees in the context of datalog analysis (e.g., [18], [19]), the focus in these works has been developing termination criteria that provide certain guarantees, rather than optimizing the construction of the tree itself.

Several optimization methods can immediately be borrowed from the techniques developed

for evaluation of datalog and logic programs, but *lifted* from the data level to the expression level: (1) memoization of nodes, (2) detection of dead ends and useless paths [15].

A more subtle case in which useless paths can be detected is as follows. Suppose we have two sibling goal nodes with labels $p_1(\bar{X})$ and $p_2(\bar{Y})$, and suppose that p_1 appears in a *single* inclusion peer description of the form $V(\bar{Z}) \subseteq p_1(\bar{X}), p_2(\bar{Y})$, and that predicate p_2 appears on the right-hand side of numerous inclusion peer mappings. In this case, the only way to reformulate p_1 will be through V, and V already satisfies the subgoal $p_2(\bar{Y})$. Hence, there is no need to explore any of the other ways of reformulating p_2: they are all redundant.

Finally, it is likely that many paths in the PDMS will be traversed frequently, and therefore we would like to develop a set of techniques that may judiciously *pre-compose* a select set of mapping chains in the network. Composition, in this context, means the following: given semantic mappings between data sources A and B, and between B and C, is it possible to generate a direct mapping between A and C that is equivalent to the original mappings? Here, equivalence means that for any query in a given class of queries $\mathcal{Q}$, and for any instance of the data sources, using the composed mapping yields exactly the same answer that would be obtained by the two original mappings.

The composition problem is also relevant to a number of static analysis questions that arise in a PDMS. First, by pre-computing the composition of mappings, we can also remove redundancies from the resulting reformulation, leading to significant run-time savings. Second, we would like to find redundant paths in the network: two paths between a pair of nodes A and B are equivalent if given any query on A, reformulating the query along both paths will result in equivalent queries on B. Third, we note that data from source A can be used in source B only if the necessary concepts are modeled in each of the nodes on the path between A and B. As a result, when paths in the network get longer, we may witness *information loss*. Hence, we would like to determine whether a path between A and B can possibly be useful for some query, and if not, find the weak links and try to improve the mappings there. We now discuss composition of mappings.

V. MAPPING COMPOSITION

In this section, we assume that all semantic mappings are of the form $Q_A(\bar{X}) \subseteq Q_B(\bar{X})$, where Q_A and Q_B are conjunctive queries over R_A and R_B respectively. We denote a mapping between A and B by $M_{A \rightarrow B}$. For brevity, we sometimes slightly abuse notation by specifying

only the bodies of Q_A and Q_B. The variables that appear in both bodies are assumed to be the head variables in both.

The composition problem is the following. Given three data sources, A, B and C, and two mappings $M_{A\to B}$ and $M_{B\to C}$. We are interested in computing a direct mapping $M_{A\to C}$ that is guaranteed to be equivalent to the two original mappings. Formally, the problem is as follows.

Definition 5.1: (**Composition**) The mapping $M_{A\to C}$ is a **composition** of the mappings $M_{A\to B}$ and $M_{B\to C}$ w.r.t. a query language $\mathcal{Q}$ if for all databases D_A for R_A, and for all queries Q over R_C such that Q is in the language $\mathcal{Q}$, the certain answers for Q w.r.t. $M_{A\to C}$ are the same as the certain answers w.r.t. $M_{A\to B}$ and $M_{B\to C}$. □ □

Note that Definition 5.1 does not define a unique composition. Instead, it defines what it means for a set of formulas to be one of several equivalent compositions. Unless otherwise mentioned, we are interested in composition w.r.t. the set of conjunctive queries. We illustrate mapping composition with two examples.

Example 5.1: Let the schemas R_A, R_B and R_C each have a single binary relation a, b and c respectively. Consider the two mappings,

$$M_{A\to B} = \{a(x,y) \subseteq b(x,x_1), b(x_1,y)\}$$

$$M_{B\to C} = \{b(x,x_1), b(x_1,x_2), b(x_2,y) \subseteq c(x,y)\}$$

If b encodes all the edges of a graph G, then $M_{A\to B}$ states that a is a subset of the node pairs with paths of length 2 in G. Similarly, $M_{B\to C}$ states that all node pairs with paths of length 3 are a subset of c. Note that, for brevity, we later use the notation $v_a(x,y) \subseteq v_b^a(x,y)$ for the formula in $M_{A\to B}$, and $v_b^c(x,y) \subseteq v_c(x,y)$ for the formula in $M_{B\to C}$.

The composition of $M_{A\to B}$ and $M_{B\to C}$ consists of the three formulas:

$$a(x,x_1), a(x_1,x_2) \subseteq c(x,y_1) \quad (1a)$$

$$a(x_1,x_2), a(x_2,x) \subseteq c(y_1,x) \quad (1b)$$

$$a(x,x_1), a(x_1,x_2), a(x_2,y) \subseteq c(x,y_1), c(y_1,y) \quad (1c)$$

Formula 1a captures the fact that if there is a path of length 4 in b emanating from x (guaranteed by the left hand-side and $M_{A\to B}$), then there is a path of length 3 emanating from x, which according to $M_{B\to C}$, means that x is in the projection of c on its first column. Formula 1b is

similar, but with paths ending at x. Formula 1c shows that even though the composition cannot obtain facts for $c(x, y)$, we *can* obtain the endpoints of paths of length two in c, by following paths of length 6 in b, which, in turn, are obtained by paths of length 3 in a. Intuitively these formulas capture all the relationships between R_A and R_C, and any other relationships follow from these. □

Example 5.1 illustrates the key difficulty in constructing a composition. It does not suffice to consider only composition formulas whose right-hand side are the c-views that appear on the right-hand side of formulas in $M_{B \to C}$. The composition may require formulas with more complex c-views, as in Formula 1c. The key challenge is to *bound* the set of c-views that are considered. In fact, the following example shows that the situation is even more subtle; the set of c-views may not even be finite.

Example 5.2: Consider the following mappings. Here, the graph encoded by R_B has red edges (relation b_r) and green edges (relation b_g).

$$\begin{aligned} M_{A \to B} &= \{a_{rg}(x, y) \subseteq b_r(x, x_1), b_g(x_1, y) \\ &\quad a_{gg}(x, y) \subseteq b_g(x, x_1), b_g(x_1, y)\} \\ M_{B \to C} &= \{b_r(x, x_1), b_g(x_1, x_2), b_g(x_2, y) \subseteq c_{rgg}(x, y) \\ &\quad b_g(x, x_1), b_g(x_1, y) \subseteq c_{gg}(x, y)\} \end{aligned}$$

As in the earlier example, a_{rg} is a subset of the node pairs with red-green paths. The other relations a_{gg}, c_{rgg} and c_{gg} can be described similarly. Observe that the following sequence of formulas are all in the composition of $M_{A \to B}$ and $M_{B \to C}$:

$$a_{gg}(x, y) \subseteq c_{gg}(x, y) \tag{2a}$$

$$a_{rg}(x, x_1), a_{gg}(x_1, x_2) \subseteq c_{rgg}(x, y_1) \tag{2b}$$

$$a_{rg}(x, x_1), a_{gg}(x_1, x_2), a_{gg}(x_2, x_3) \subseteq c_{rgg}(x, y_1), c_{gg}(y_1, y_2) \tag{2c}$$

$$a_{rg}(x, x_1), a_{gg}(x_1, x_2), \ldots, a_{gg}(x_n, x_{n+1}) \subseteq c_{rgg}(x, y_1), c_{gg}(y_1, y_2), \ldots, c_{gg}(y_{n-1}, y_n) \tag{2d}$$

The sequence is infinite. Each equation in the infinite sequence captures the formula that a path comprising of one red (b_r) edge followed by $2n + 1$ green (b_g) edges (the query over R_A) is contained within a path comprising of a red edge followed by $2n$ green edges (the query over R_C). None of them can be expressed in terms of the others (e.g., the rule 2c is not implied by rules 2a and 2b). Fortunately, as we will see later, in some cases (including this one) there is a finite encoding of the infinite set of GLAV formulas. □ □

$$
\begin{array}{ll}
M_{A\rightarrow B} & v_a:\ a(x,y)\ \subseteq\ v_b^a:\ b(x,x_1),b(x_1,y) \\
M_{B\rightarrow C} & v_b^c: b(x,x_1),b(x_1,x_2),b(x_2,y)\ \subseteq\ v_c: c(x,y) \\
 & \\
Q(x,y) & q_c(x,y)\ :-\ c(x,y_1),c(y_1,y) \\
 & \Downarrow \\
Rew_C(Q) & q_c'(x,y)\ :-\ v_c(x,y_1),v_c(y_1,y) \\
 & \downarrow \\
Query_B(Q) & q_b(x,y)\ :-\ v_b^c(x,y_1),v_b^c(y_1,y) \\
 & q_b(x,y)\ :-\ b(x,z_1),b(z_1,z_2),b(z_2,y_1), \\
 & b(y_1,z_3),b(z_3,z_4),b(z_4,y) \\
 & \Downarrow \\
Rew_B(Q) & q_b'(x,y)\ :-\ v_b^a(x,z_2),v_b^a(z_2,z_3),v_b^a(z_3,y) \\
 & \downarrow \\
Query_A(Q) & q_a(x,y)\ :-\ v_a(x,z_2),v_a(z_2,z_3),v_a(z_3,y) \\
 & q_a(x,y)\ :-\ a(x,z_2),a(z_2,z_3),a(z_3,y)
\end{array}
$$

Fig. 3. Composing $M_{A\rightarrow B}$ and $M_{B\rightarrow C}$ in Example 5.1 w.r.t. a single query as a sequence of reformulations.

A. Composition algorithm

In [20] we describe a composition algorithm for GLAV mappings. We now provide a brief overview of the algorithm. To see how to construct a set of composition formulas that hold for *every* query, we first describe how to compose $M_{A\rightarrow B}$ and $M_{B\rightarrow C}$ for a *single* query, Q, over R_C. We use the terminology defined here as a basis for the composition algorithm.

1) Composition w.r.t. a single query: Informally, we proceed in two steps. In the first we reformulate the query Q using $M_{B\rightarrow C}$, and in the second, we reformulate the result using $M_{A\rightarrow B}$. Each of the steps has two parts; first we reformulate the query using the right-hand side of the formulas, and then we replace the views on the right-hand side with those appearing on the left. We illustrate this process in Figure 3 for the query $Q(x,y) :- c(x,y_1), c(y_1,y)$ and the mappings in Example 5.1. We proceed by computing the following queries:

$\mathbf{Rew_C(Q)}$: the maximally-contained rewriting of Q in terms of the views on the right-hand sides of the formulas in $M_{B\rightarrow C}$.

$\mathbf{Query_B(Q)}$: a query over R_B obtained by replacing the views in $Rew_C(Q)$ by the corresponding views on the left-hand sides of the formulas in $M_{B\rightarrow C}$, and unfolding these view definitions.

$$
\begin{array}{llll}
Query_B(Q) & = & q_b(x,y) & :- \quad \overbrace{b(x,z_1),b(z_1,z_2),b(z_2,y_1)}^{v_b^c(x,y_1)}, \quad \overbrace{b(y_1,z_3),b(z_3,z_4),b(z_4,y)}^{v_b^c(y_1,y)} \\
Rew_B(Q) & = & q'_b(x,y) & :- \quad \overbrace{b(x,u_1),b(u_1,x_1)}^{v_b^a(x,x_1)}, \quad \overbrace{b(x_1,u_2),b(u_2,x_2)}^{v_b^a(x_1,x_2)}, \quad \overbrace{b(x_2,u_3),b(u_3,y)}^{v_b^a(x_2,y)}
\end{array}
$$

Fig. 4. Query unfolding for $Query_B(Q)$ and $Rew_B(Q)$ from Figure 3

$\mathbf{Rew_B(Q)}$: the maximally-contained rewriting of $Query_B(Q)$ using the views on the right-hand sides of the formulas in $M_{A\to B}$.

$\mathbf{Query_A(Q)}$: a query over R_A obtained by replacing the views in $Rew_B(Q)$ by the corresponding views on the left-hand sides of the formulas in $M_{A\to B}$, and unfolding these view definitions.

Given the procedure above, we can provide the first characterization of the composition of $M_{A\to B}$ and $M_{B\to C}$ as a set of GLAV formulas:

Propostion 5.1: Let $\mathcal{C}$ be the set of all GLAV formulas of the form $Q_A(\bar{x}) \subseteq Q_C(\bar{x})$, where:

- $Q_C(\bar{x})$ is a conjunctive query over R_C, and
- $Q_A(\bar{x})$ is one of the conjunctive queries in $Query_A(Q_C(\bar{x}))$.

Then, $\mathcal{C}$ is a composition of $M_{A\to B}$ and $M_{B\to C}$ w.r.t. the set of conjunctive queries over R_C. □

In general, $\mathcal{C}$ in the above proposition may be infinite, and hence the challenge our algorithm faces is to create the composition in these cases.

Our algorithm is based on several key observations. First, we identify the set of *minimal composition formulas*, and show that they are sufficient for producing a composition. Intuitively, these composition formulas are minimal in the sense that we cannot get the same results by using a combination of smaller formulas. That is, there exists a database instance such that these formulas will produce certain answers that cannot be produced by piecing together smaller formulas in the composition.

The second observation is that minimal formulas can be built in increasing size. That is, if $Q_A(\bar{x}) \subseteq Q_C(\bar{x})$ is a minimal composition formula where Q_C has n subgoals, $n > 1$, then there exists another minimal composition formula $Q'_A(\bar{x}) \subseteq Q'_C(\bar{x})$ where Q'_C has $n-1$ subgoals, a subset of the subgoals in Q_C, and hence possibly a subset of the head variables of Q_C.

Given the two observations above, a mapping composition algorithm can begin with composition formulas whose right-hand sides have only a single atom. At every step, the algorithm considers the minimal formulas computed thus far, and tries to *extend* them by adding another

atom to their right-hand side. If this process terminates, *i.e.* when no new minimal rules can be obtained by extension, the set of all computed minimal rules (a finite set) will be a composition of the given mappings.

The third observation underlying our algorithm is to identify a condition on minimal composition formulas that essentially tells us that the two formulas *can be extended in similar ways*. With that condition, we can partition formulas into equivalence classes and treat all the formulas in an equivalence class identically. If we can show that there is a finite number of equivalence classes, then it follows that the algorithm will terminate with a composition. We formalize the condition with the notion of *residues*. To illustrate the notion of a residue, consider how formula 1a in Example 5.1 could be extended to obtain formula 1c. The intermediate steps in deriving formula 1a are shown in Figure 5.

$$\begin{array}{llll}
q(x,y_1) :- & c(x,y_1) & Rew_C(q): & v_c(x,y_1) \\
Query_B(q): & \overbrace{b(x,z_1),b(z_1,z_2),b(z_2,y_1)}^{v_b^c(x,y_1)}, & & \\
Rew_B(q): & \overbrace{b(x,u_1),b(u_1,x_1)}^{v_b^a(x,x_1),}, & \overbrace{b(x_1,u_2),\mathbf{b(u_2,x_2)}}^{v_b^a(x_1,x_2)} &
\end{array}$$

Fig. 5. Deriving formula 1a in Example 5.1

The containment mapping from $Query_B(q)$ to $Rew_B(q)$ maps the variable y_1 in $Query_B(q)$ to the variable u_2 in $Rew_B(q)$. The last atom in $Rew_B(q)$, $\mathbf{b(u_2,x_2)}$, is not the target of any atom in $Query_B(q)$. Observe that we can extend $Rew_C(q)$ (and the containment mapping) to introduce an atom in $Query_B(q)$ that includes y_1, such that $b(u_2,x_2)$ is in the target of the extended containment mapping.

The extended query would include a join condition (using variable y_1) that cannot be captured using formulas 1a and 1b (since the join variable y_1 maps to the variable u_2). Atom $b(u_2,x_2)$ and the position of variable u_2 in that atom characterize the possible extensions of the formula, and constitute the *residue* of the formula. A complete formula is obtained by extending the maximally contained rewriting $Rew_B(q)$ to cover the new atoms introduced in $Query_B(q)$.

2) The Algorithm: Our algorithm constructs a *Query Rewrite Graph* (QRG) that encodes the composition of two sets of mapping formulas. A QRG consists two kinds of nodes: *Query* nodes and *Rewrite* nodes. Paths in a QRG contain alternate query nodes (Q_is) and rewrite (R_is) nodes.

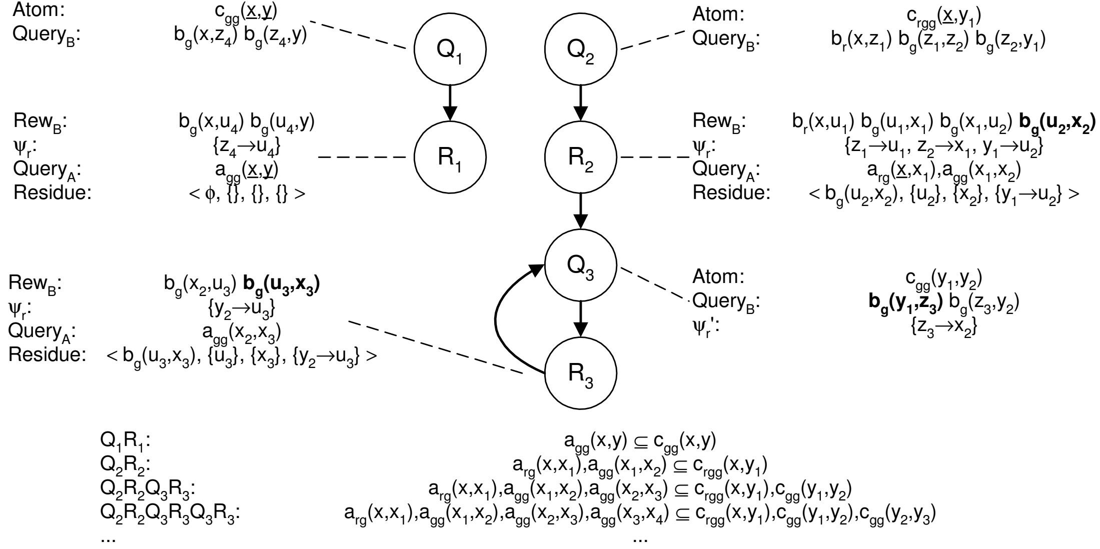

Fig. 6. Query-Rewrite Graph for Example 5.2. The query-rewrite graph consists of query nodes and rewrite nodes. The right-hand sides of formulas in the composition are encoded by paths of query nodes from the root, and the left-hand sides are encoded by the corresponding rewrite nodes. Below the tree we show how the different composition formulas are encoded by paths in the tree.

Every path $p : Q_1 R_1 \ldots Q_n R_n$, from a root node Q_1, encodes a minimal composition formula $r(p) : Q_A(\bar{x}) \subseteq Q_C(\bar{x})$. Each Q_i contains a single atom from R_c such that the Q_is along p can be chained to obtain the query Q_C. Similarly, the R_is can be chained to obtain the query Q_A.

The key to the construction of the QRG is that we do not expand a subgoal in the tree if there is already an expanded subgoal with an equivalent residue. In [20] we show that for a large class of queries (namely, CQ_k queries [21]) there are guaranteed to be only a finite number of non-isomorphic residues, and hence the algorithm will terminate and will generate the exact composition. Note that even if the number of residues is not finite, any subset of the QRG produces only correct composition formulas, albeit not all possible ones. In Figure 6 we show the QRG for the composition of the mappings in Example 5.2.

VI. The Piazza PDMS

Early in this paper, our focus has been understanding query answering in the PDMS from a formal perspective. Our goal has been to establish the semantics of our system implementation, which we call Piazza. Our prototype Piazza system is designed to be a scalable foundation for

data sharing applications.

In this section, we discuss two aspects of our Piazza implementation that are of special interest. First, we overview our architecture for query answering in the PDMS. Second, in any practical system designed to integrate data, XML makes a much better interchange format than relational data. We provide an overview of the XML version of Piazza's $\mathcal{PPL}$ language.

A. System Architecture and Implementation

A Piazza PDMS consists of a set of nodes (physical peers) connected in an overlay network on the Internet. Following the logical PDMS model, every peer may have a peer (mediated) schema, and it may optionally provide source data and query processing capabilities. Finally, a peer may specify schema mappings. We note that in contrast to P2P file sharing systems, we assume the PDMS to be a relative stable environment: joining a PDMS is a heavyweight operation, and data contributors are unlikely to be modem users, so we expect that peers will seldom leave (and if they do, they will notify the system).

Query reformulation is performed at the node that receives the query — this allows enumeration of all possible rewritings and detection of redundant rewritings. The reformulator assumes a global *system catalog* that provides access to all of the mappings that involve a particular peer relation. At each step in the rule-goal tree expansion, the catalog is consulted to expand the frontier. We currently use a centralized catalog and cache the mappings at each peer for performance and robustness. The rewritings from the reformulator are ultimately pipelined to the query processor at the originating node, and it is responsible for delegating portions of the query plan to other nodes.

We believe that this area has many opportunities for future work. We plan to reimplement the catalog using distributed hash table techniques (e.g. [22], [23]), which will increase scalability and robustness. We hope to investigate improvements to the rewriting algorithm, both in terms of optimizations and in terms of distributing the work. Finally, we hope to investigate adaptive approaches to distributing the query processing itself across the PDMS.

B. Mapping XML Data

Earlier in this paper, we presented the relational version of $\mathcal{PPL}$, our peer-mapping language. We now briefly describe the language we use for mapping between XML nodes in a Piazza

overlay network, which is more complex due to the richer XML data model. The algorithm for using these mappings for query answering is described in [24], and that paper also discusses issues relating to soundness and completeness of answers.

Each Piazza node has an XML Schema that 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. As in the relational case, mappings play two roles. The first 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 that have at least one author from the University of Washington. The second role is as *peer 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.

Following the data integration literature, which uses a standard relational query language for both queries and mappings, we might elect to use XQuery [25] for both our query language and our language for specifying mappings. However, we found XQuery inappropriate as a mapping language. 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 (in fact, Turing-complete) query language, and as a result some aspects are difficult or even impossible to reason about.

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 restricted XQuery 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. 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 "fuses" these results together based on the output element's unique identifiers (similar to the use of Skolem functions in languages such as XML-QL [26]). A complete formal description of the language is given in [24]. Here, we describe the main ideas of the language and illustrate them through examples.

Each mapping definition begins with an XML template that matches some path or a subtree of a legal instance of the target schema, i.e., a prefix of a legal string in the target schema's grammar. Elements in the template may be annotated with restricted XQuery expressions 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.

Figure 7 shows an example of two peer XML schemas, *Source1.xml* and *Source2.xml*. Figure 8(a) defines a simple mapping from the schema of Source2.xml of Figure 7 to Source1.xml. 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 `$typ`, which are bound to the author, title, and publication-type elements in the source, respectively. We construct a title and author element for each occurrence of book. The author name contains a new query expression annotation (`$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

Source1.xml schema:

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

Source2.xml schema:

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

Source3.rdf OWL class definition:

```
Class id = "book"
 DataTypeProperty id = "bookTitle"
   domain = "#book"
   range = "&xsd;string"
 DataTypeProperty id = "bookAuthor"
   domain = "#book"
   range = "#author"
Class id = "author"
 DataTypeProperty id = "authorName"
   domain = "#author"
   range = "&xsd;string"
```

Fig. 7. An example of three peer (data source) schemas (two are XML sources and one is an RDF source with an OWL ontology). Source1.xml contains `books` with nested `authors`; Source2.xml contains `author`'s with nested `publications`. Indentation illustrates nesting and a * suffix indicates "0 or more occurrences of...", as in a BNF grammar. Source3.rdf is a set of OWL class and property definitions with a slightly simplified notation.

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.

Example 6.1: See Figure 8(b) for an improved mapping that does coalescing of book elements. 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

```
<pubs>
  <book>
  {: $a IN document("Source2.xml")\
              /authors/author,
     $t IN $a/publication/title/text(),
     $typ IN $a/publication/pub-type/text()
     WHERE $typ = "book" :}
    <title>{ $t }</title>
    <author>
      <name>{: $a/full-name/text() :}</name>
    </author>
  </book>
</pubs>
```

(a) Initial mapping

```
<pubs>
  <book piazza:id={$t}>
  {: $a IN document("Source2.xml")\
              /authors/author,
     $t IN $a/publication/title/text(),
     $typ IN $a/publication/pub-type/text()
     WHERE $typ = "book" :}
    <title piazza:id={$t}>{ $t }</title>
    <author piazza:id={$t}>
      <name>{: $a/full-name/text() :}</name>
    </author>
  </book>
</pubs>
```

(b) Refined mapping that coalesces entries

Fig. 8. Simple examples of mappings from the schema of Source2.xml in Figure 7 to Source1.xml's schema.

— 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 6.2: Continuing with the previous schema, consider the partial mapping:

```
<pubs>
  <book piazza:id={$t}>
  {: $a IN document("Source2.xml")/authors/author,
     $t IN $a/publication/title/text(),
     $typ IN $a/publication/pub-type/text()
     WHERE $typ = "book"
     PROPERTY $t >= 'A' AND $t < 'B'
  :}
    <title piazza:id={$t}>{ $t }</title>
    <author piazza:id={$t}>
      <name>{: $a/full-name/text() :}</name>
    </author>
    [:  <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: here, 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. □

VII. PDMS AS INFRASTRUCTURE FOR THE SEMANTIC WEB

A careful comparison of Piazza's operation with the efforts of the Semantic Web community reveals many shared goals: both the PDMS and the Semantic Web are designed to provide Web-scale sharing of structured data, thereby enabling more sophisticated queries, and performing more complex tasks involving Web sites. While there has been much research on defining the meaning of data on the Semantic Web and relationships between data, little has been said about how one would piece together large numbers of Web data sources and perform queries that span many of them. We believe that a PDMS provides a solid basis for building Semantic Web applications. The PDMS architecture enables the creation of webs 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.

The Piazza system has been implemented with an XML data model to provide the greatest flexibility in accommodating real-world data, since most relational, semistructured, and even RDF data is available in XML form. Throughout this paper, we have explained the relationship between the PDMS model and traditional data integration techniques. It is also important to

consider how Piazza relates to the current Semantic Web efforts.

As noted, most of the research on the Semantic Web to date has focused on RDF as an underlying data model. In [27], Patel-Schneider and Simeon note 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 so 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 development tools and web services rely on these structures. Clearly, it would be desirable for the Semantic Web to be able to inter-operate 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.

Hence, we argue that a PDMS based on XML as a data model provides infrastructure for supporting rich Semantic Web applications. 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. Here we explain how RDF and OWL data instances can already be incorporated into our current, XML-based PDMS.

The main distinctions between RDF and unordered XML[3] are that XML (unless accompanied by a schema) does not assign semantic meaning to any particular attributes, and XML uses

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

hierarchy (membership) to implicitly encode logical relationships. Within 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. 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 [27], 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 defining schemas. The XML world uses XML Schema, which is based on object-oriented classes and database schemas. An XML schema 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 [28] and the various levels of OWL [2] 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.

Much 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; concepts 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 extensive 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., [29]). 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.

Subject to the differences in expressiveness of concepts/views, we can fairly straightforwardly map RDF *data instances* into an XML structure, and vice-versa. It is in converting from trees to graphs that Piazza's piazza:id feature becomes especially important, as it allows us to coalesce multiple occurrences of an RDF/OWL class into one entry. We conclude this section with a brief example that maps from XML to RDF.

Example 7.1: Suppose that we are attempting to map from Source2.xml of Figure 7 and the OWL instance Source3.rdf. We can use the following mapping:

```
<book piazza:id={$t} rdf:about={"http://myorg.org/Source3.rdf#" + $s3}>
  {:  $a IN document("Source2.xml")/authors/author,
       $t IN $a/publication/title/text(),
       $an IN $a/full-name/text()
       $typ IN $a/publication/pub-type/text(),
      WHERE $typ = "book" :}
    <bookTitle rdf:resource={ "#" + $an }> {$t} </bookTitle>
</book>
<author rdf:about={"http://myorg.org/Source3.rdf#" + $an}>
  {:  $a IN document("Source2.xml")/authors/author,
       $typ IN $a/publication/pub-type/text(),
       $an IN $a/full-name/text()
      WHERE $typ = "book" :}
    {$an}
</author>
```

□

VIII. RELATED WORK

The idea of mediating between different databases using local semantic relationships was explored in federated databases and cooperative databases (e.g., [30], [31], [32], [33]). There, it was assumed that each database in the federation stored data, and the focus was on mapping between the stored relations in the federation. Our work differs in several ways. First, the scale of a PDMS is assumed to be much larger and its structure more ad hoc. Joining and leaving a PDMS should be much easier than in a federated database. As a consequence, the relationships between the peers are much looser. Second, peers can play different roles — some provide data, others provide integration services between other peers, and some provide both. As a result, we

need to be able to map both relationships among stored relations and among conceptual relations (i.e., extensional vs. intentional relations). Third, our focus is on algorithms for chaining through multiple peer mappings in order to locate data relevant to a query.

In [34] we described some of the challenges involved in building a PDMS, focusing on *intelligent data placement*, a technique for materializing views at nodes in the network in order to improve performance and availability. In [35] the authors study a variant of the data placement problem, and focus on intelligently caching and reusing queries in an OLAP environment. Recently, [36] described *local relational models* as a formalism for mediating between different peers in a PDMS, and a sound and complete algorithm for answering queries using the formalism, but do not describe the expressive power of the formalism compared to previous ones in the data integration literature.

Edutella [37] represents an interesting design point in the XML-RDF interoperability spectrum. Like Piazza, it is built on a peer-to-peer architecture (based on JXTA) and it mediates between different data representations — but it provides query and storage services for RDF, using a variety of 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.

Several other projects in the database community are developing peer-to-peer architectures, with slightly different emphases. The Chatty Web [38] focuses on gossip protocols for exchanging semantic mapping information, where mappings are selection-projection queries that they evaluate for information loss. Hyperion [39] focuses on problems relating to mappings between *objects* in different relations, which is another important aspect of mapping between sources. PeerDB [40] takes another approach to mapping between peers: instead of schema mappings, PeerDB employs an Information Retrieval-based approach to query reformulation. A peer relation (and each of its columns) is associated with a set of keywords. PeerDB reformulates a query over one schema into other peers' schemas by matching the keywords associated with the two schemas. Keywords can be matched directly between any pair of schemas, so chaining of reformulation steps is not required; however, keyword matching may give irrelevant query reformulations, so the user must decide which queries are to be executed.

In the KR community, work on the OBSERVER [41] and Kraft [42] systems have explored a number of issues in distributed ontologies, including mappings from structured sources and approximate mappings between concepts in ontologies.

IX. CONCLUSIONS

The concept of the peer data management system emphasizes not only an ad-hoc, scalable, distributed peer-to-peer computing environment (which is compelling from a distributed systems perspective), but it provides an easily extensible, decentralized environment for sharing data with rich semantics. This is in contrast to data integration systems, which have a centralized mediated schema and administrator, and which, in our experience, impede small, point-to-point collaborations. It also complements the knowledge representation work of the Semantic Web by providing a mechanism for translating between different ontologies' data representations.

We described some of the main the highlights of the Piazza PDMS, including (1) a solution to schema mediation in peer data based on a language that uses previous mediation formalisms at the local level to form a network of semantically related peers, (2) a characterization the theoretical limitations on answering queries in a PDMS, (3) an algorithm for answering queries in such a system, and (4) results concerning the composition of semantic mappings. We also argued that a PDMS can provide a basis for building applications for the Semantic Web, and we showed how to extend Piazza to the XML data model.

Though we have not described these in this paper, we have implemented a prototype of Piazza and built a small web of database-research related web sites. We are currently focusing on developing and testing effective methods for query optimization in Piazza, and the management and propagation of updates in a PDMS. In addition, we believe that the management of large collections of semantic mappings raises interesting challenges. Our work on query composition lays the basis for studying several fundamental properties of such networks. We are also interested in studying how one can *boost* a collection of such mappings to improve the ability of nodes to obtain relevant data from other distant nodes in the network. Finally, peer data management is a very rich domain that creates a wealth of new problems, such as how to replicate data and how to reconcile inconsistent data.

Acknowledgements: This work was funded in part by NSF ITR grants IIS-0205635 and IIS-9985114 and a gift from Microsoft Research.

REFERENCES

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

[2] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL web ontology language 1.0 reference," Available from http://www.w3c.org/TR/2002-WD-owl-ref-20020729/, 29 July 2002, w3C Working Draft.

[3] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom, "The TSIMMIS project: Integration of heterogeneous information sources," *Journal of Intelligent Information Systems*, vol. 8(2), pp. 117–132, March 1997.

[4] L. Haas, D. Kossmann, E. Wimmers, and J. Yang, "Optimizing queries across diverse data sources," in *Proc. of VLDB*, Athens, Greece, 1997.

[5] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian, "Query caching and optimization in distributed mediator systems," in *Proc. of SIGMOD*, Montreal, Canada, 1996, pp. 137–148.

[6] A. Y. Levy, A. Rajaraman, and J. J. Ordille, "Querying heterogeneous information sources using source descriptions," in *VLDB '96*, 1996, pp. 251–262.

[7] O. M. Duschka and M. R. Genesereth, "Answering recursive queries using views," in *PODS '97*, 1997, pp. 109–116.

[8] I. Manolescu, D. Florescu, and D. Kossmann, "Answering xml queries on heterogeneous data sources," in *Proc. of VLDB*, 2001, pp. 241–250.

[9] J. L. Ambite, N. Ashish, G. Barish, C. A. Knoblock, S. Minton, P. J. Modi, I. Muslea, A. Philpot, and S. Tejada, "ARIADNE: A system for constructing mediators for internet sources (system demonstration)," in *Proc. of SIGMOD*, Seattle, WA, 1998.

[10] E. Lambrecht, S. Kambhampati, and S. Gnanaprakasam, "Optimizing recursive information gathering plans," in *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 1999, pp. 1204–1211.

[11] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. Lin, and E. Wong, "Multibase – integrating heterogeneous distributed database systems," in *Proceedings of the National Computer Conference*. AFIPS Press, Montvale, NJ, 1981, pp. 487–499.

[12] A. Y. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[13] D. Draper, A. Y. Halevy, and D. S. Weld, "The nimble integration system," in *Proc. of SIGMOD*, 2001.

[14] S. Abiteboul and O. Duschka, "Complexity of answering queries using materialized views," in *PODS '98*, Seattle, WA, 1998, pp. 254–263.

[15] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov, "Schema mediation for large-scale semantic data sharing," *VLDB Journal, to appear*, 2003.

[16] M. Friedman, A. Levy, and T. Millstein, "Navigational plans for data integration," in *Proceedings of AAAI*, 1999.

[17] Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Integrating network-bound XML data," *IEEE Data Engineering Bulletin Special Issue on XML*, vol. 24, no. 2, June 2001.

[18] A. Y. Halevy, I. Mumick, Y. Sagiv, and O. Shmueli, "Static analysis in datalog extensions," *Journal of the ACM*, vol. 48(5), pp. 971–1012, September 2001.

[19] D. Srivastava and R. Ramakrishnan, "Pushing constraint selections," in *Proc. of PODS*, San Diego, CA., 1992, pp. 301–315.

[20] J. Madhavan and A. Halevy, "Composing mappings among data sources," in *Proc. of VLDB*, 2003.

[21] M. Y. Vardi, "On the complexity of bounded-variable queries," in *Proc. of PODS*, 1995, pp. 266–276.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proc. of ACM SIGCOMM '01*, 2001.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. of ACM SIGCOMM '01*, 2001.

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

[25] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu, “XQuery 1.0: An XML query language,” 30 April 2002, available from http://www.w3.org/TR/xquery/.

[26] 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.

[27] P. Patel-Schneider and J. Simeon, “Building the Semantic Web on XML,” in *Int’l Semantic Web Conference ’02*, June 2002.

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

[29] A. Levy and M.-C. Rousset, “Combining Horn rules and description logics in carin,” *Artificial Intelligence*, vol. 104, pp. 165–209, September 1998.

[30] W. Litwin, L. Mark, and N. Roussopoulos, “Interoperability of multiple autonomous databases,” *ACM Computing Surveys*, vol. 22 (3), pp. 267–293, 1990.

[31] R. Krishnamurthy, W. Litwin, and W. Kent, “Language features for interoperability of databases with schematic discrepancies.” in *Proc. of SIGMOD*, Denver, Colorado, 1991, pp. 40–49.

[32] M. Rusinkiewicz, A. Sheth, and G. Karabatis, “Specifying interdatabase dependencies ina multidatabase environment,” *IEEE Computer*, vol. 24:12, 1991.

[33] T. Catarci and M. Lenzerini, “Representing and using interschema knowledge in cooperative information systems,” *Journal of Intelligent and Cooperative Information Systems*, pp. 55–62, 1993.

[34] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu, “What can databases do for peer-to-peer?” in *ACM SIGMOD WebDB Workshop 2001*, 2001.

[35] P. Kalnis, W. Ng, B. Ooi, D. Papadias, and K. Tan, “An adaptive peer-to-peer network for distributed caching of olap results,” in *Proc. of SIGMOD*, 2002.

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

[37] 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*, 2002, pp. 604–615.

[38] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth, “The chatty web: Emergent semantics through gossiping,” in *Twelfth International World Wide Web Conference*, 2003.

[39] R. J. M. Anastasios Kementsietsidis, Marcelo Arenas, “Mapping data in peer-to-peer systems: Semantics and algorithmic issues,” in *SIGMOD ’03*, 2003.

[40] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou, “PeerDB: A P2P-based system for distributed data sharing,” in *SIGMOD ’03*, 2003.

[41] E. Mena, V. Kashyap, A. P. Sheth, and A. Illarramendi, “OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies,” *Distributed and Parallel Databases*, vol. 8, no. 2, pp. 223–271, 2000.

[42] A. Preece, K. Hui, and P. Gray, “Kraft: An agent architecture for knowledge fusion,” *IJCIS*, vol. 10, no. 1-2, pp. 171–195, 1999.