Representing and Querying Data Transformations

Yannis Velegrakis
AT&T Labs - Research
velgias@research.att.com

Renée J. Miller
University of Toronto, Canada
miller@cs.toronto.edu

John Mylopoulos
University of Toronto, Canada
jm@cs.toronto.edu

Abstract

Modern information systems often store data that has been transformed and integrated from a variety of sources. This integration may obscure the original source semantics of data items. For many tasks, it is important to be able to determine not only where data items originated, but also why they appear in the integration as they do and through what transformation they were derived. This problem is known as data provenance. In this work, we consider data provenance at the schema and mapping level. In particular, we consider how to answer questions such as "what schema elements in the source(s) contributed to this value", or "through what transformations or mappings was this value derived?" Towards this end, we elevate schemas and mappings to first-class citizens that are stored in a repository and are associated with the actual data values. An extended query language, called MXQL, is also developed that allows meta-data to be queried as regular data and we describe its implementation. scenario.

1. Introduction

Large amounts of information from many domains of activity are currently available on the web in a number of autonomous heterogeneous sources and repositories. Modern information systems and e-commerce applications use data integration tools to locate, collect, translate and integrate this information in order to provide the user with a unified view of the data. This integration is achieved through a set of transformation queries or views (usually called *mappings*) that map instance data from the sources to instance data of the integrated schema [16]. Data obtained from well-known sources such as government organizations, research institutes or private companies may be well-understood and accepted. However, data coming from the integration of independent, physically distributed, heterogeneous sources are not always well-understood. Some of the original data semantics may be lost during the integration, and new data may also be added in the process. Furthermore, as a consequence of the transparent access provided by the integration, the notion of distinct sources and their structures often disappears from queries and results. Hence, searching the integrated information for data that is not only relevant, but also best suited to a task at hand, is really a challenge.

On the other hand, everyday life includes numerous applications where users need to know and reason about the origin of data [5, 8, 13, 24]. One reason may be to evaluate the quality of the retrieved results. Knowing from where each particular data element was drawn and how it was computed allows users to apply their own judgment to the credibility of that information and decide whether some particular data is a semantically correct answer to their query. In systems where information from multiple sources is used, such knowledge may assist in interpreting the data semantics and resolving potential conflicts among data retrieved from different sources. In several other emerging applications, the ability to analyze "what-if" scenarios in order to reason about the impact of the data coming from specific sources (or parts of them) is of paramount importance.

The problem of determining the origin of a specific data element in an integrated view is known to the research community as the problem of *data lineage* [8] or *data provenance* [5]. Most of the work on lineage tracing has concentrated on instance-level tracing, that is, finding the specific values in the base tables that justify the appearance of a data element in a view. One way the problem is approached is by developing methods to generate the right queries on the source schema to return the data elements that are "responsible" for the appearance of a data value in the view. To generate such a query, the information of the view instance and the view query is required. Computing the inverse of the view query may be required [6]. In many cases, such a costly computation may be avoided if the interest is not in the originating data values themselves, but in the schema elements used or the way in which the data was transformed. This new paradigm suggests a new kind of lineage that is at the schema level and uses schema elements and mappings instead of specific data values. Such information can provide users with a better understanding of the semantics of the data, and can help in resolving semantic conflicts that may arise from the integration of data from disparate heterogeneous sources [4]. In large scale integration systems, where the integrated instance is populated from many large schemas through numerous complex mappings, schema level provenance is a required first step to identify the mappings that generated a specific data value in the integration and the schema elements to which it was applied. This information can be provided to the user, or can be used to compute the data-level provenance if the exact originat-

ing values need to be identified or can be used to locate a piece of data according to the mappings applied to them.

To compute schema-level lineage, schemas and mappings need to be stored, managed and allowed to be queried as regular data. Meta-data information has been used in many applications. Commercial relational database systems, for instance, store schema information and view definitions in specialized tables called system catalogs. In our work, we are exploring how to make this information available to a designer in a systematic way by providing facilities for querying not only catalog information, but also very general mappings used to integrate and transform data. We also permit the origin of data to be recorded. For example, if data is integrated using a view that is generated through a union of multiple queries, we ensure that provenance information is not lost when the data is merged, a capability that is typically not automatically provided by commercial materialized view facilities. Currently, in order to deal with this limitation, data administrators usually store, with the integrated data, meta-data information in an ad-hoc way by tagging or annotating specific parts of the data. Our goal here is to make this process systematic by providing data annotations that have a well-known meaning and are guaranteed to be reliable. In that way, we offer a more comprehensive set of meta-data and the functionality of formulating queries that use meta-data information along with the actual data, in a transparent and consistent way. The need for this kind of expressiveness in query languages has already been recognized in the research community [15]. Previous work has considered only schema information as meta-data, but in many applications, mapping level information is also important.

The goal of this work is to go further and provide an integrated solution and a systematic way for uniformly representing and declaratively querying data, its schema-level provenance, and the mappings through which it has been derived. We address issues such as determining what schema components and data sources contributed to the generation of a specific piece of information, through what views or queries this data was generated, and what transformation these queries performed. To achieve this, schemas and mappings are elevated to first-class citizens in the repository and the query language. An important component of our work is the representation of not just the identity of mappings, but their internal, declarative structure to help designers understand, compare and debug mappings.

Our work is motivated by noumerous application scenarios and environments. Scientific data, for example, may be retrieved by third parties, transformed and stored in local databases. For researchers interested in the accuracy and timeliness of scientific data, for example, it is essential to know the original location from which a given piece of data was drawn, and the processing that has been applied to it. In information portals, knowing the source and the transformations through which the data was computed is important in order to assess its quality. In data warehouses, to support what-if scenarios, queries may be parameterized with conditions on originating schemas and view queries that control how the warehouse instance has been generated.

In this work, our main contributions are:

1. We describe and implement a representation model for schemas and transformations.
2. We introduce annotations on data values in order to associate them with their meta-data information. Annotations are not only super-imposed information but can be queried along with data.
3. We develop and implement an extended query language, called MXQL, for uniformly manipulating data and meta-data by utilizing the proposed representation model and data annotations.
4. We show that the semantics of MXQL introduce a new kind of lineage (or provenance) that uses schema elements instead of data values.
5. We manage not only the origin of the data but also the transformations through which it has been derived.
6. We report our experience in using our implementation in a real world scenario.

The paper is organized as follows. Section 2 motivates the problem and Section 3 discusses the related work. Section 4 introduces and formally defines the data model and Section 5 presents an extension to this model to support queries on schemas and transformations. Section 6 investigates the properties of the proposed query language and Section 7 shows how it can be implemented. Our experience with the framework is reported in Section 8.

2. Motivating example

To better understand the problem, consider the case of a specific portal that integrates information from various real estate sites from around the world. Figure 1 indicates a portion of the schema of a European (`EUdb`) and an American (`USdb`) data source, as well as a portion of the portal (`Pdb`) integrated schema. The dotted arrows between the two schemas indicate how their elements correspond. The portal is populated with data retrieved from the two sources through the three mappings m_1, m_2 and m_3 indicated in Figure 1. Mappings are used to describe data transformations from one format to another. Mappings will be formally presented in Section 4.3. For the moment, what is important to note is that mappings are of the form **foreach** Q_s **exists** Q_t where Q_s is a query on a data source schema describing what data to retrieve, and Q_t is a query on the integrated schema describing how the retrieved data will be structured to conform to the integrated schema. The first mapping of Figure 1 populates the integrated schema with data from the American site, in particular, house entries and independent agents. The second performs a similar task, but considers firms instead of independent agents. The third populates `Pdb` with similar data from the European data source.

Mapping generation requires good knowledge and understanding of the semantics of the schemas and the language in which the mappings are expressed. A number of tools have recently been developed to assist the user

in this process by increasing the abstraction level [19], semi-automatically generating mappings [20] or preserving the semantics of the transformations while schemas evolve [23]. However, it is in the nature of the problem that data mapped from one format to another may lose part of its semantics. Consider, for example, a user that is interested in estates valued at more than $500K. The following query can be executed against the portal data:

select * from Portal.estates where value>500K

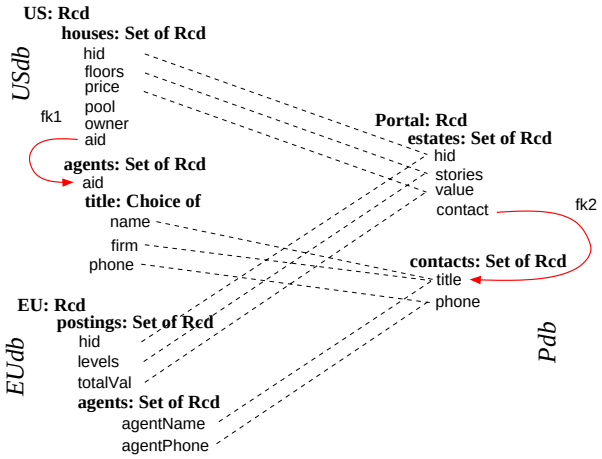
The results returned to this query may not be an answer consistent with what the user wanted. It is common for American companies to not include tax in prices while European companies do. If this is the case in the above scenario, values in the result set of the query that originate from the European source will have the tax included while those originating from the American source may not. Unfortunately, there is no information in the portal schema to distinguish the two different kinds of prices. This information has been lost once the data from the two sources has been merged in the portal instance. It would be helpful if the user could specify in the query whether she is interested in data values originating from one data source or another or if the query could return, along with every data value in the answer set, the data source from which it originates.

As another example, consider a designer who notices some anomalous values in the `contacts` element in the integrated data. First, she would like to perform a query to find out from which elements the anomalous values were derived. In response to such a query, she can learn that elements from houses and agents were used to compute the anomalous values. In particular, one mapping may join `houses` and `agents` on `aid`, while another may join them on the element `owner`. She can then decide that the desired contact information in the portal should include the owner, and may correct or remove the anomalous mapping.

The above examples demonstrate a need for incorporating into queries predicates regarding how and from where the data values were derived. If the portal administrator realizes this need and has the flexibility to alter the portal schema, she can introduce some additional schema elements to keep that information. However, portal schemas are not always allowed to change, or the administrator may not be aware of subtle differences in the semantics of the values, so we may not be able to rely on an administrator to manually annotate the integrated data with relevant information. In this work, we propose an integrated solution to address this problem by providing a systematic way to store meta-data without relying on the integrator, and a declarative query language to use that information.

3. Related Work

There are numerous proposals in the literature that elevate various types of meta-data to first-class citizens so that they can later be queried. The Gaea System [13] is a system propose a system for modeling data derivation processes, but the derivations are not associate with the actual data. Chawathe et al. [7] explicitly represent database changes,



m_1: **foreach**
select h.hid, h.floors, h.price, n, a.phone
from US.houses h, US.agents a, a.title→name n
where h.aid=a.aid
exists
select e.hid, e.stories, e.value, c.title, c.phone
from Portal.estates e, Portal.contacts c
where e.contact=c.title

m_2:**foreach**
select h.hid, h.floors, h.price, f, a.phone
from US.houses h, US.agents a, a.title→firm f
where h.aid=a.aid
exists
select e.hid, e.stories, e.value, c.title, c.phone
from Portal.estates e, Portal.contacts c
where e.contact=c.title

m_3:**foreach**
select p.hid, p.levels, p.totalVal, a.agentName, a.agentPhone
from EU.postings p, p.posting.agents a
exists
select e.hid, e.stories, e.value, c.title, c.phone
from Portal.estates e, Portal.contacts c,
where e.contact=c.title

Figure 1. A Schema Mapping Scenario.

thereby, allowing queries to be executed on different versions of database instances over time. In a similar fashion, our work explicitly models schemas and mappings permitting queries to trace the origin of the transformations used to derive the data. Data annotations have also been used by Wang and Madnick [24] to keep track of all the data sources from which a specific data value originates, but they consider neither schema element information, nor transformations. Querying data and schemas has also been proposed by Lakshmanan et al. [15] who consider schemas as first-class citizens of a query language. However, the main scope of this work is to use schema elements in transformation queries, and naturally, they do not consider using the transformation information in queries.

Cui and Widom [8] describe *data lineage* of a specific value in a view as the tuples in the base tables of the view that justify its existence. Buneman et al. [5] characterize this same notion as *why-provenance* and distinguish it from *where-provenance*, which specifies the exact position in the *why-provenance* from which a view value was extracted. Fan and Poulovassilis have also considered data provenance through schema transformations [11]. In our work, when we refer to the origin of the data we mean the schema elements and transformations from which they originate and not the data values that the data provenance considers. In that sense, our work complements data provenance. Furthermore, we do not consider only the origin of the data, but also the transformation that has been applied to them.

Currently, there are emerging efforts to build annotation systems (e.g., Annotea [14]) where data annotations defined by the users propagate along with the data. This will facilitate the use of superimposed information on the data [3]. The problem of propagating annotations from a view back to the source has been studied by Buneman et al. [6] and found to be NP-hard. In our case, annotations are queried as regular data, so the complexity of the problem is not more than the complexity of evaluating a query over an instance.

Concurrently to our work, Bhagwat et. al [2] are developing an annotation management system for relational databases. This work is complementary to ours in that it provides a query language, pSQL, and system for efficiently propagating annotations as data is transformed. However, annotations on the queried data are given (not created) and left uninterpreted. In contrast, in our work, we create annotations that record the origin of data and the transformations applied. Furthermore, our query language (unlike pSQL) permits users to query the properties of annotations.

Our work can be seen as a contribution to a broader framework of model management [1] in which meta-data is managed as first-class citizens [19], meaning it can be generated [21], composed [9, 18] or updated [23]. In particular we provide a framework for storing and querying meta-data. The Rondo model management system [19] provides a set of conceptual structures for modeling schema structures, constraints and mappings, but provides no query language for querying them and no mechanism to associate instance data values with their meta-data information. Mappings are binary relationships that establish n:m correspondences between the conceptual schema structures. This representation is limited since it cannot express, for example, that when an `estate` entry is generated in the portal of our motivating example, element `contact` should reference the corresponding contact information. We build on this framework by explicitly modeling the semantics of the translation as nested tuple generated dependencies [10], and allow mappings and their internal structure to be queried.

4. Data Model

The data model we use, is based on the well studied relational model extended to support union types, nested structures and XML-Schema-like constraints, and has successfully been used before [21, 23, 25] as a common model to deal with the heterogeneity of integrated data.

4.1. Schemas and Instances

The model includes a set of atomic types (e.g., **integer**, **string**, etc.), set and complex types. A complex type is either a record or a union. A *Record* type is of the form $\mathsf{Rcd}[A_1{:}\tau_1, \ldots, A_k{:}\tau_k]$, where the symbols $A_1, \ldots, A_k$ are called *labels* or *attributes* and each τ_i represents an atomic, a set, or a complex type. A record value of type $\mathsf{Rcd}[A_1{:}\tau_1, \ldots, A_k{:}\tau_k]$ is a tuple of label-value pairs: $[A_1{:}a_1, \ldots, A_k{:}a_k]$, where $a_1, \ldots, a_k$ are values of type $\tau_1, \ldots, \tau_k$, respectively. A *Union* (or Choice) type is of the form $\mathsf{Choice}[A_1{:}\tau_1, \ldots, A_k{:}\tau_k]$, where τ_i and A_i are as in record types. A value of type $\mathsf{Choice}[A_1{:}\tau_1, \ldots, A_k{:}\tau_k]$ is a label-value pair $[A_i{:}a_i]$, where a_i is a value of type τ_i, with $i{\in}[1,k]$. Repeatable elements are modeled through *set* types. A *set* type is of the form $\mathsf{Set\ of}\ \tau$ where τ is a complex type. A value of type $\mathsf{Set\ of}\ \tau$ is represented by a *set ID* and an associated set $\{v_1, \ldots, v_n\}$ of "children" values, where each v_i is of type τ. The types τ_i and τ in the specifications above are said to be directly used in the complex and set type, respectively.

A schema S consists of several label names, each with an associated type: $R_1{:}\tau_1, R_2{:}\tau_2, \ldots R_n{:}\tau_n$. Each such pair is referred to as a *root element*. Any label-type pair that appears as a root element or within the type of a root element is referred to as *schema element*. Types within set types, e.g., the type $\mathsf{Rcd}[\ldots]$ in the type $\mathsf{Set\ of\ Rcd}[\ldots]$, are assumed to have the implicit and usually omitted label "*".

Due to the way types are specified, a schema can be represented as a set of trees where each node corresponds to a schema element and the tree roots correspond to schema roots. Hence, a schema can be modeled as a set of nodes over which a parent-child relationship is defined. In particular,

Definition 4.1 *A* schema S *is a pair* $<\mathcal{E}, f^{parent}>$ *where* $\mathcal{E}$ *is a set of label-type pairs, referred to as* schema elements, *and* f^{parent} *is a total function* $f^{parent}{:}\mathcal{E}{\rightarrow}\mathcal{E}{\cup}\{null\}$, *such that* $\forall e{\in}\mathcal{E}\ f^{parent}(e){=}e'$ *if element* e *is directly used in the type of* e', *or* $f^{parent}(e){=}null$ *in which case element* e *is a* root element.

Relational schemas can be expressed as nested relational schemas by representing each table T as a schema root $T{:}\mathsf{Set\ of\ Rcd}[A_1{:}\tau_1, \ldots, A_n{:}\tau_n]$ where the attributes in the record correspond to the attributes of the table. To emphasize the relationship with the relational model, every schema root of type $\mathsf{Set\ of\ Rcd}[...]$ with atomic type attributes in the record is referred to as *relation* and every record value in such a set is referred as *tuple*. For an XML schema, each global element is represented as schema root and each local element as an attribute of a complex type unless it is repeated in which case it is represented through a set [21].

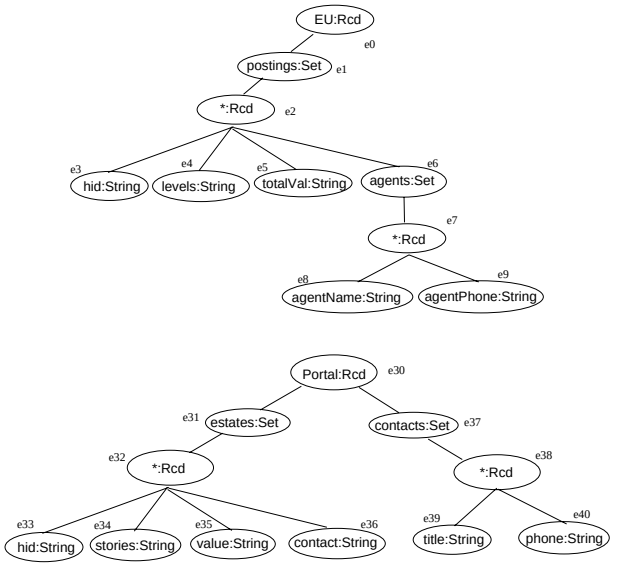


Figure 2. `EUdb` **and** `Pdb` **schema graphs.**

Definition 4.2 *An* instance $\mathcal{I}$ *is a set of label-value pairs* $l{:}v$ *and* conforms *to schema* S *if there is a total injective function "elementOf" that maps every label-value pair* $l{:}v$ *to a schema element* $l{:}\tau$ *of* S *such that value* v *is of type* τ*. Given an instance, the* interpretation *of a schema element* $l{:}\tau$*, noted as* $\mathcal{I}[l{:}\tau]$*, is the set of pairs* $l{:}v$ *in the instance for which elementOf*$(l : v) = l : \tau$*.*

For brevity, we will usually refer to a label-value pair in an instance as a *value* if there is no risk of confusion. As is common in the literature, we represent nested instances as trees. There is exactly one node in the graph for each distinct value in the instance. Nodes representing attributes of a complex type value are labeled with the attribute label and are connected to a node that represents that complex value through an edge. Similarly, all the nodes representing members of a set are labeled with the implicit label "*" and are connected through an edge to the node representing the set.

Each data source has an instance and a schema to which it conforms. In order to distinguish between the different sources (or *databases*) each has a unique name assigned.

Example 4.3 *Figure 2 indicates the graph representations of the* `EUdb` *and* `Pdb` *schemas in Figure 2. Figure 3 shows the graph representation of an instance of the* `Pdb` *data source. The node annotations should be ignored for now.*

4.2. Queries

Queries are formed by *select-from-where* clauses with path expressions extended with a special selection symbol "$\rightarrow$" to mark choice of attribute in a union type. A *path expression* (or *expression*) is a root element or a variable followed by a series of record projections and union choices. An expression exp is defined by the grammar $exp ::= S|x|exp.l|exp{\rightarrow}l$ where x is a variable, S a root element, l a label, $exp.l$ a record projection, and $exp{\rightarrow}l$ a union choice. A *valuation* of an expression over an instance $\mathcal{I}$ is the value described by the expression after an instantiation of the variables of the query to actual values of the instance such that the structure and the conditions in the query are satisfied. Each expression *refers to* a specific schema element. The *type* of an expression is the type of the schema element to which the expression refers. Every valuation of an expression is a value from the interpretation of the schema element to which the expression refers.

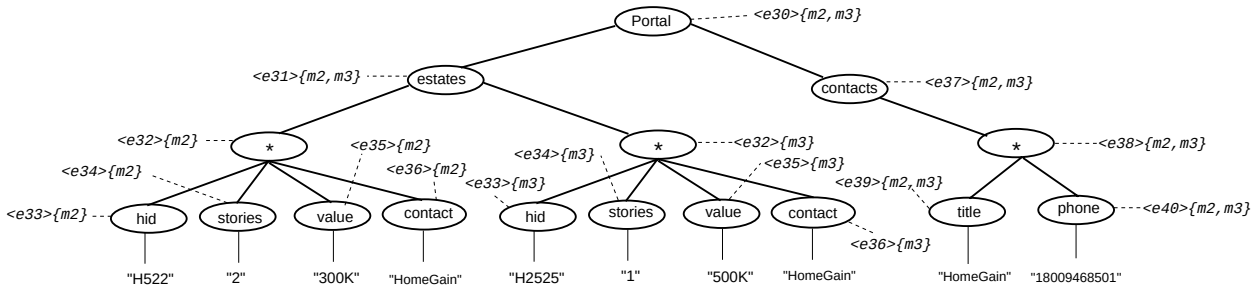
Example 4.4 *Expression* `US.agents` refers *to element* `agents` *in* `USdb` *database and is of set type. If variable* x *is bound to a record type in that set,* x`.title` *refers to the element* `title` *and is of union type. Then, expression* x`.title`$\rightarrow$`name` *refers to element* `name` *of that union and is of* **string** *type.*

A *well-formed* query is a query of the form:

select $exp_0, exp_1, ..., exp_m$
from $P_0\ x_0, P_1\ x_1, ..., P_n\ x_n,$
where $c_0\theta_0 c_0'$ and $c_1\theta_1 c_1'$ and ... and $c_k\theta_k c_k'$

where each P_i is an expression that either is of type set or is a union type choice of the form $exp{\rightarrow}l$. Expression P_i can use only variables that have been previously defined, i.e., expression P_i can use a variable x_j only if $j{<}i$. Every expression c_i, c_i', and exp_i is either an atomic type expression that uses at most one of the variables $x_0 \ldots x_n$ or it is a constant atomic value. Each operator θ_i is one of $<$, $>$, $\leq$, $\geq$ or $=$. If an expression P_i is of type Set, then variable x_i binds to individual elements of the set. If not, then it will bind to the element to which expression P_i refers. For instance, if P_i is of type Set of τ then the variable x_i will bind to the elements of type τ in that set. If, on the other hand, P_i is of the form $exp{\rightarrow}l$ (where exp is an expression of type Choice$[\ldots, l{:}\tau_l, \ldots]$), then variable x_i will bind to the element of type τ_l under the choice l. Nesting and grouping are supported through the capability of the P_i expressions to use previously bound variables, thus, nested conjunctive queries [21], SQL, and FLWR expressions with no wildcards can be expressed. Note that the functionality of wildcards is not a real requirement in our queries, since, having the schema available, we can always perform explicit query unfolding and replace them. The query language covers the core queries that have been considered in the literature for use in data exchange and data integration mapping expressions [22, 16, 21, 10], and can also be extended to include aggregation functions, negation and order. However, since our goal is not to provide and study an expressive query language but rather a framework for representing and querying meta-data (schemas and mappings) and in order to keep the presentation simple, the subsequent sections will not consider these features.

Queries may also contain function calls. A function call accepts as arguments one or more values (described as expressions) and returns a single value or a set of values. Function calls returning a set can be used in the from clause as one of the P_i expression.



4.3. Mappings

Queries are used to describe mappings. In keeping with the data integration and data exchange literature [16], we define a mapping between two schemas S_s and S_t as an expression of the form **foreach** Q_s **exists** Q_t where Q_s is a query on the first schema and Q_t a query on the second. Schemas S_s and S_t will be referred to as *source* and *target* schema, respectively. Mappings of this form are referred to as *GLAV* [12] mappings, and they naturally include the *GAV* [16] and *LAV* [17] mappings that have been used extensively in integration systems. *GLAV* mappings have also been considered in model management [1], virtual data transformation [25], peer-to-peer systems [18] and data exchange [10]. Queries Q_s and Q_t should have the same number of type compatible expressions in their select clauses. Given a pair of instances $\mathcal{I}_s$ of schema S_s and $\mathcal{I}_t$ of schema S_t, the mapping is satisfied if $\forall t \in Q_s(\mathcal{I}_s) \Rightarrow t \in Q_t(\mathcal{I}_t)$ where $Q_s(\mathcal{I}_s)$ and $Q_t(\mathcal{I}_t)$ represent the results of executing queries Q_s and Q_t on instances $\mathcal{I}_s$ and $\mathcal{I}_t$, respectively, and t a tuple of atomic values. A mapping can be seen as a form of inter-schema constraint specifying what data from the first schema corresponds to data of the second [16]. Given a number of such mappings, there may be many transformation functions $f: I_1 \rightarrow I_2$ satisfying the mappings. We have already studied how to find these functions in [21] where it was also shown that although the data units associated by the mappings are tuples of atomic values, complex structures can also be exchanged. Furthermore, more than one expression in the select clause can be combined together through a function to form new complex expressions. This allows mappings that can map more than one elements of one schema to an element of a second schema.

Example 4.5 *Mapping m_1 of Figure 1 asserts that every tuple consisting of* `hid`, `stories`, `price`, `name` *and* `phone` *values generated by joining on* `aid` *the* `houses` *and the* `agents` *of the* `USdb` *data source must be in the result of the query that joins* `estates` *and* `contacts` *of the* `Pdb` *data source on* `title`. *The correspondences between the attributes of the tuples in the two queries is determined by the positions of the expressions in the* select *clauses. For example, the* `price` *corresponds to* `value` *since expressions* h.`price` *and* c.`value` *are both in the third position of the* select *clauses.*

A mapping from a schema $<\mathcal{E}_1, f_1^p>$ to schema $<\mathcal{E}_2, f_2^p>$ can be modeled as a triple $<\mathcal{E}_s, \mathcal{E}_t, W_c>$ where $\mathcal{E}_s \subseteq \mathcal{E}_1$, $\mathcal{E}_t \subseteq \mathcal{E}_2$, and W_c is a set of atomic type element pairs from $(\mathcal{E}_s \cup \mathcal{E}_t) \times (\mathcal{E}_s \cup \mathcal{E}_t)$. Intuitively, the sets $\mathcal{E}_s$ and $\mathcal{E}_t$ consist of all the schema elements that are referred to by the expressions in the **foreach** and **exists** clauses of the mapping, respectively. The set W_c consists of pairs of elements that are referred to either by two expressions in a binary predicate in a where clause (in which case both elements are from the same schema), or by two expressions in the same position of the two mapping select clauses, e.g., for mapping m_3 of Figure 1, elements `totalVal` and `value`. We will use this representation to model lineage information from mappings.

5. Supporting meta-data querying

This section provides the formal basis for realizing schema mapping scenarios where data, schema elements, and transformations are available for querying. It builds on the data model of the previous section by directly extending it to make schemas and mappings first-class citizens of the model. It then extends the query language with special operators to utilize this meta-data information. This extended query language will be referred to as MXQL, that stands for *meta-data extended query language*.

Definition 5.1 *A* mapping setting *is a triple $<\boldsymbol{\mathcal{S}}_s, S_t, \mathcal{M}>$ where $\boldsymbol{\mathcal{S}}_s$ is a set of source schemas, S_t a target schema and $\mathcal{M}$ a set of mappings, each from a schema $S \in \boldsymbol{\mathcal{S}}_s$ to S_t.*

Note that the interpretation of an element e of S_t for an instance I of S_t, i.e., the set $I[e]$, contains values of the instance I that may have been generated by different mappings. We will use the notation $I[e]^m$ to refer to the subset of $I[e]$ that was generated through mapping m. A detailed discussion of the semantics for $I[e]^m$ and an implementation of one semantics was presented elsewhere [10, 21]. To allow schemas and mappings to be considered as data and allow them to be queried, we introduce the notion of a tagged instance.

Definition 5.2 *Given a mapping setting $<\boldsymbol{\mathcal{S}}_s, S_t, \mathcal{M}>$, an instance of each schema of $\boldsymbol{\mathcal{S}}_s$ and an instance $\mathcal{I}_t$ of schema S_t satisfying the mappings in $\mathcal{M}$, if $\mathcal{E}_t$ represents the set of elements of schema S_t, a* tagged instance *$\mathcal{I}^M$ is a 6-tuple $<\mathcal{I}_t, \boldsymbol{\mathcal{S}}_s, S_t, \mathcal{M}, f_{mp}, f_{el}>$ where f_{mp} and f_{el} are total functions defined as:*

$f_{el} : \mathcal{I}_t \rightarrow \mathcal{E}_t$ *s.t.* $f_{el}(v) = e$, *where* $v \in I[e]$

$f_{mp} : \mathcal{I}_t \rightarrow \mathcal{M}$ *s.t.* $f_{mp}(v) = \{m \mid m \in \mathcal{M} \wedge v \in I[f_{el}(v)]^m\}$

Intuitively, function f_{el} associates each value in the target instance $\mathcal{I}_t$ of the mapping setting with the schema element in the interpretation to which it belongs. (Recall that

in our model each value is a label-value pair.) This is a functionality that in most database systems is implicit in the storage mechanism. SchemaSQL [15] is an effort to overcome this limitation and allow queries on both instance values and schema elements. We go further, by incorporating into the model not only the schema elements but also the mappings that perform the data transformation. This is achieved through the function f_{mp}, which associates with each data value in the instance $\mathcal{I}_t$ the mappings that generated it by explicitly modeling the structure of the transformations. Note that a data value in the instance $\mathcal{I}_t$ may be generated by multiple mappings, hence, function f_{mp} returns a set. In the graph representation of an instance, the information from functions f_{el} and f_{mp} is represented through annotations on the graph nodes.

Example 5.3 *The annotations in Figure 3 are indicated with dotted lines. Those in angle brackets represent the element information (provided by function f_{el}), while those in curly brackets represent the mapping information (provided by function f_{mp}). For instance, the annotation of the* `title` *node indicates that it belongs to the interpretation of element $e39$ (Figure 2), and both mappings m_2 and m_3 have generated it.*

In order to be able to use databases, schema elements and mappings in queries and be able to return them in results, we extend our data model with three new atomic types: Database, Mapping and Element with domains the set of databases, mappings and elements, respectively. To access the schema element and mapping information of a value, two new operators are also introduced: @elem and @map. The first returns a value of type Element and the second a set of values of type Mapping. In particular, given an expression exp referring to an element e of schema S_t over a tagged instance $<\mathcal{I}_t, \boldsymbol{\mathcal{S}}_s, S_t, \mathcal{M}, f_{mp}, f_{el}>$ and a valuation v of exp over this instance, the interpretation of the two operators is: $I[v@\text{elem}]=f_{el}(v)$ (that is, v@elem is the schema element to which the value v refers), and $I[v@\text{map}]=f_{mp}(v)$ (that is, v@map is the set of mappings through which the value v was derived.)

Example 5.4 *Assume that a user is interested in retrieving all the prices of the estates from the portal. To better realize the meaning of a price, i.e., if it includes tax, if it is in its original currency or has been converted, etc., the user may need to know for each price, through what transformation it was generated. This can be provided by the MXQL query:*

```
select x.estate.hid, x.estate.value, m
from   Portal.estates x, x.value@map m
```

Note that operation @map *is used in the* from *clause as a variable binding expression, since it returns a set of values.*

One of our main goals is to not only query the mappings that generated the values in the integrated instance, but also trace back in order to find the schema elements of the sources from which these values originate or the elements on which they depend. For that, we introduce the *mapping predicate*, $<S_s{:}e_s \rightarrow m \rightarrow S_t{:}e_t>$. This boolean mapping predicate can be used in the where clauses of the queries. It specifies certain conditions regarding the origin, the transformation and the mappings that transform the data. When these conditions are satisfied, the mapping predicate evaluates to true. Given a tagged instance $<\mathcal{I}_t, \boldsymbol{\mathcal{S}}_s, S_t, \mathcal{M}, f_{mp}, f_{el}>$, the interpretation of the mapping predicate is defined by:

$I[<S_s{:}e_s \rightarrow m \rightarrow S_T{:}e_t>] = \exists m{:}<\mathcal{E}'_s, \mathcal{E}'_t, W_c> \in \mathcal{M}$ such that
$S_T{:}<\mathcal{E}_t, f^p_t> = S_t \wedge \exists S_s{:}<\mathcal{E}_s, f^p_s> \in \boldsymbol{\mathcal{S}}_s$ where $\mathcal{E}'_s \subseteq \mathcal{E}_s \wedge$
$\mathcal{E}'_t \subseteq \mathcal{E}_t \wedge \exists e_s \in \mathcal{E}'_s \wedge \exists e_t \in \mathcal{E}'_t \wedge (e_s = e_t) \in W_c$

Intuitively, the predicate evaluates to true if (i) there is a mapping m that uses in the select clause of its **foreach** part an expression that refers to the element e_s of one of the schemas in $\boldsymbol{\mathcal{S}}_s$, and (ii) in the select clause of its **exists** part, it uses an expression that refers to the element e_t of the target schema S_t, in a way that the values with which the mapping populates schema element e_t are the retrieved values of element e_s.

Example 5.5 *In the mapping setting of Figure 1, if e_s is the* `price` *element, S_s is schema* `USdb`*, m is mapping m_1, S_t is schema* `Pdb`*, and s_t is the* `value` *element, then the predicate $<S_s{:}e_s \rightarrow m \rightarrow S_t{:}e_t>$ evaluates to true since mapping m_1 generates values for the* `value` *element by retrieving* `price` *values from the* `USdb` *database.*

The mapping predicate can be combined with the @map *and* @elem *operators to form interesting queries. Assume, for example, that a user is interested in finding estates in the* `Pdb` *database originating from the* `USdb` *data source, and specifically, those having a firm as a contact. For each such estate the user needs to know the transformation (mapping) that generated its price. Unless the data administrator who designed the portal schema had taken special care to include that information in the schema, which is not the case in our example, this query cannot be answered since firms and individual agents have been merged under the element* `contacts` *and cannot be distinguished. Using MXQL, this requirement can be expressed as follows:*

```
select s.hid, m
from   Portal.estates s, Portal.contacts c, c.title@map m
where  s.contact=c.title and e=c.title @elem
       and <'USdb':'US/agents/title/firm' → m → 'Pdb':e>
```

The constant values `'USdb'` *and* `'Pdb'` *in the above query are of type Database, and the value* `'US/agents/title/firm'` *is a constant of type Element. The* @map *operator makes the Mapping type variable m iterate over all the mappings that generate values for the element* `title` *of the portal schema, and operator* @elem *makes the Element type variable e be the schema element* `title`*. The mapping predicate restricts variable m even further to only mappings that use* `firm` *values from the* `USdb` *to populate the* `title` *element of the portal. Note that variables used in the mapping predicate need not be defined in the* from *clause, e.g., variable e. They are implicitly defined through their position in the mapping predicate. Executing the above query over the*

tagged instance of Figure 3 returns tuple $('H522','m_2')$*, where* $'H522'$ *is a string and* $'m_2'$ *a Mapping type value.*

MXQL queries can also be used to form queries on metadata exclusively in order to understand the complex structure of the schemas and the way data is transformed.

Example 5.6 *Consider a user who does not completely understand what the meaning of the element* `stories` *is in the portal schema. In particular, she is wondering whether this is a comment about the number and type of floors in a property, or a comment about the history of the property. In order to find out, she asks the following query to identify from where its values originate:*

select e from
where $<db{:}e \rightarrow m \rightarrow 'Pdb':'$/Portal/estates/estate/stories$'>$

The query returns Element type values `floors` *and* `levels` *as an answer, making the user realize that the element* `stories` *describes how many stories the building has. Note that, in the above query, the* from *clause is empty, since the variables used in the query are implicitly defined by their use in the mapping predicate.*

In many cases, for a given value in the integration, users are interested in knowing not only from where it originates, but also what other parts of the schema contain values that affect its appearance in the integration [5]. To be able to answer such queries, we introduce a new form of the mapping predicate, having similar syntax but using double arrows instead of single. Its interpretation is defined by:

$I[<S_s{:}e_s \Rightarrow m \Rightarrow S_T{:}e_t>] = \exists m{:}<\mathcal{E}'_s,\mathcal{E}'_t,W_c> \in \mathcal{M}$ such that
$S_T{:}<\mathcal{E}_t,f^p_t>=S_t \ \wedge \ \exists S_s{:}<\mathcal{E}_s,f^p_s> \in \boldsymbol{\mathcal{S}}_s$ where $\mathcal{E}'_s \subseteq \mathcal{E}_s \ \wedge$
$\mathcal{E}'_t \subseteq \mathcal{E}_t \ \wedge \ \exists e_s \in \mathcal{E}'_s \ \wedge \ \exists e'_s \in \mathcal{E}'_s \ \wedge \ \exists e''_s \in \mathcal{E}'_s \ \wedge \ \exists e_t \in \mathcal{E}'_t \ \wedge$
$(e_s = e'_s) \in W_c \ \wedge \ (e''_s = e_t) \in W_c$

Intuitively, the predicate evaluates to true if (i) there is a mapping m that uses in the select clause of its **exists** part the element e_t of the target schema S_T, i.e., it generates values for e_t, and (2) it uses element e_s of one of the schemas $S_s \in \boldsymbol{\mathcal{S}}_s$ of the mapping setting in its where clause. This means that, although the values of e_s are not populating element e_t, they affect its population through their participation in the where clause conditions.

Example 5.7 *In the mapping setting of Figure 1, the values of the element* `aid` *are not used by any of the mappings to populate an element of the portal, since the portal schema has no* `aid` *element. However, its values play an important role in the population of the target schema, since they are used to form the join between the* `house` *and* `agent` *values. Thus, element* `aid` *will be in the answer set of the query:*

select c.`title`, e_s
from `Portal.estates` s, `Portal.contacts` c, c.`title`@map m
where s.`contact`=c.`title` and e=c.`title` @elem
and $<'$`USdb`$':e_s \Rightarrow m \Rightarrow '$`Pdb`$':e>$

This information may be useful to a designer wishing to understand why certain agents appear associated with specific houses.

6. Characterizing MXQL Queries

This section provides a more specific characterization of the mapping predicates. It shows that mapping predicates describe conditions on the schema elements of the *data provenance* [8, 5].

Consider two data sources db_s and db_t with schemas S_s and S_t, respectively, and a mapping m generating an instance $\mathcal{I}_t$ of S_t from instance $\mathcal{I}_s$ of S_s. Let query

q_f: select $exp_1, exp_2, ..., exp_m$
from $P_0\ x_0, ..., P_n\ x_n$ where $cond_1$ and ... and $cond_l$

be the **foreach** clause of mapping m and a value $v \in I[e_t]^m$. Since $v \in I[e_t]^m$, there is an expression exp_{e_t} in the select clause of q_f through which value v was retrieved from the source instance $\mathcal{I}_s$. Let query q'_f be one that has the same from and where clause as q_f but a select clause with all the possible valid expressions that can exist in the query. According to Buneman et al. [5], although they had considered a different data model, query q'_f, with the additional condition $exp_{e_t}=v$ in its where clause, returns the *witness basis* $W_{q_f,\mathcal{I}_s}(v)$, i.e., the *why-provenance* of value v.

Query q_f with only expression exp_{e_t} in its select clause and the additional condition $exp_{e_t}=v$ in the where clause returns the *derivation basis* $\Gamma_{q_f,\mathcal{I}_s}(v)$, i.e., the *where-provenance* of value v.

We need to note here that the result of a query is not considered a simple set of values, but a set of facts, i.e., values of the instance associated with their specific positions.

Assume that the mapping predicate $<db_s{:}e_s \rightarrow m \rightarrow db_t{:}e_t>$ is satisfied (evaluates to true) for some elements e_s and e_t. This means that, for a value $v \in I[e_t]^m$, there is an expression exp_{e_t} in the select clause of q_f that generated value v, and this expression refers to element e_s. For the mappings we consider here, query q_f with the additional condition $exp_{e_t}=v$ in its where clause and only expression exp_{e_t} in its select clause represents $\Gamma_{q_f,\mathcal{I}_s}(v)$, which means that $\Gamma_{q_f,\mathcal{I}_s}(v) \in I[e_s]$.

Theorem 6.1 *Given two data sources* db_s *and* db_t*, and a mapping* m*, the mapping predicate* $<db_s{:}e_s \rightarrow m \rightarrow db_t{:}e_t>$ *is satisfied if and only if there is a value* $v' \in I[e_s]$ *that is in the* where-provenance *of a value* $v \in I[e_t]^m$ *through* m*.*

Let query q^w_f be a query with the same from and where clause as query q_f, but with the select clause containing all the expressions that appear in the select or the where clause of query q_f. This query defines a new form of provenance which we refer to as the *what-provenance.*

Definition 6.2 (What-Provenance) *Consider two data sources schemas* S_s *and* S_t*, a mapping* $m : Q_s \subseteq Q_t$ *from* S_s *to* S_t*, two instances* $\mathcal{I}_s$ *and* $\mathcal{I}_t$ *of schemas* S_s *and* S_t*, respectively, and a value* v *of* $\mathcal{I}_t$*. Assuming that instance* $\mathcal{I}_t$ *has been generated through the mapping* m *from instance* $\mathcal{I}_s$*, if query* Q_s *is the query*

select $exp_1, exp_2, ..., exp_m$
from $P_0\ x_0, ..., P_n\ x_n$
where $cond_1$ and ... and $cond_l$

the what-provenance *of the value* v *is the query*

q_f: select U
from $P_0\ x_0, ..., P_n\ x_n$
where $cond_1$ and ... and $cond_l$ and $exp_{e_t} = v$

where U is the set of all the expressions in the select *clause and the expressions used in the conditions of the* where *clause of query Q_s, and exp_{e_t} is the expression in the* select *clause of query Q_s that generated value v.*

The *what-provenance* is value-based compared to the complex-structure-based (tuples or proof trees) *why-provenance*. It differs from why-provenance in that it does not consider the parts of the complex structures returned by the *why-provenance* that do not justify the appearance of a value in the generated instance. This difference becomes clear with the following relational example. Consider the two source schema relations $R(a, b, c)$ and $S(c, d, e)$ with contents $\{(3, 2, 3), (1, 2, 4)\}$ and $\{(4, 5, 6)\}$, respectively, and a target schema relation $T(a, b)$. Assume that the target schema relation is populated by the mapping

m: **foreach**
select $r.a, t.c$ from $R\ r, S\ s$ where $r.c{=}s.c$
exists
select $t.a, t.b$ from $T\ t$

As a result of a data exchange with mapping m, the target schema relation T will be populated with only the tuple: $(1, 2)$. The *where-provenance* of value 1 in this tuple is the value of attribute a of tuple $(1, 2, 4)$ since this is from where value 1 was derived. The *why-provenance* of the same value consists of the values $(1, 2, 4, 4, 5, 6)$ of relations R and S, respectively, since the join of those two tuples led to tuple $(1, 2)$. However, note that attributes d and e of relation S, whatever values they may have, will not affect the result of the query. Our *what-provenance* does not include them. Attribute c on the other hand is important since the join of R and S uses c. In the specific example, the what-provenance will be the tuple $(1, 2, 4)$.

In order to characterize the relationships among the *where*, *why* and *what* provenance, we introduce the notion of *element inclusion* between queries.

Definition 6.3 (Element Inclusion) *Let q_1, q_2 be two queries. Query q_1 is* element-included *in query q_2, noted as $q_1 \sqsubseteq q_2$, if there is a total injective renaming function h from the variables of q_1 to the variables of q_2 such that the* from *and* where *clauses of the queries $h(q_1)$ and q_2 are the same, and $K_1 \subseteq K_2$ where K_1, K_2 are the ordered sets of expressions in the* select *clauses of queries $h(q_1)$ and q_2, respectively.*

Intuitively, the above definition states that if $q_1 \sqsubseteq q_2$, then for every instance $\mathcal{I}$, $q_1(\mathcal{I}) = \pi_X(q_2(\mathcal{I}))$ where π_X means projection on some set of elements X.

In general, if queries q_{why}, q_{where} and q_{what} represent the *why*, *where* and *what-provenance*, respectively, it holds that $q_{where} \sqsubseteq q_{what} \sqsubseteq q_{why}$.

We will show next that the mapping predicate with double arrows relates to the *what-provenance*.

If the mapping predicate $<db_s{:}e_s \Rightarrow m \Rightarrow db_t{:}e_t>$ is satisfied for some elements e_s and e_t, then, for an instance value $v \in I[e_t]^m$, there is an expression exp_j in the select or the where clause of q_f that refers to the schema element e_s. By definition, expression exp_j will be in the what-provenance select clause, hence, the interpretation of element $[e_s]$ is in the what-provenance.

Db($\underline{name}$)
Element($\underline{eid}$, $name$, $type$, $parent$, db)
Query($\underline{qid}$)
Binding($\underline{bid}$, $\underline{qid}$, eid, $prev$)
Condition(qid, bid, eid, op, $bid2$, $eid2$)
Mapping($\underline{mid}$, $forQ$, $conQ$)
Correspondence(mid, $forBid$, $forEid$, $conBid$, $conEid$)

Figure 4. Meta-data storage schema.

Theorem 6.4 *Given two data sources db_s and db_t, and a mapping m, the mapping predicate $<db_s{:}e_s \Rightarrow m \Rightarrow db_t{:}e_t>$ is satisfied if and only if a value $v \in I[e_s]$ is in the* what-provenance *of a value $v' \in I[e_t]^m$ through mapping m.*

7. Implementing MXQL

This section describes our implementation of the MXQL query language using existing technology. Schemas and mappings, in order to be queried and returned in answer sets as regular data, need to be stored. Although there may be many different methodologies targeting different applications with different requirements, we propose one specific storage schema and we explain how MXQL queries can be executed by exploiting that storage schema.

7.1. Meta-data Physical Storage Schema

Figure 4 presents a number of Set of Rcd[...] types (represented as relations for notational simplicity) used to store meta-data information. Recall that a relation in our model is a schema root R of type Set of Rcd$[a_1{:}\tau_1, ..., a_n{:}\tau_n]$ where every type τ_k is an atomic type.

The $Element$ relation encodes the graph representation of a schema using the edge approach. Each tuple corresponds to a node, i.e., a member of the set $\mathcal{E}$ of schema $<\mathcal{E}, f^{parent}>$. Attribute *name* records its label and *type* specifies its type. Function f^{parent} of schema $<\mathcal{E}, f^{parent}>$ is implemented through the *parent* attribute that specifies the parent node. Finally, attribute db refers to the name of the data source.

Three relations are used to model queries: $Query$, $Binding$ and $Condition$. Each query gets a unique identifier that is recorded in relation $Query$. Relation $Binding$ records the from clause of each query as a list of bindings. Each tuple represents a binding and has a unique identifier (bid) within a query (qid). For the binding $P_i\ x_i$, variable x_i becomes the binding identifier. Expression P_i is represented by two parts: the variable or schema root with which it starts and the schema element to which it refers. The first is encoded in the attribute $prev$, referencing the corresponding binding. The second is encoded in the attribute eid, referencing the corresponding element identi-

fier in the $Element$ relation. Since queries have no bindings for schema roots, implicit bindings are introduced for each schema root used in the query. These bindings have the $prev$ attribute set to null. The where clause of a query is modeled in the $Condition$ relation, where each tuple represents a condition. Conditions are of the form $expr_1 \theta expr_2$. Each of the expressions $expr_1$ and $expr_2$ is encoded similarly to expression P_i, that is, by specifying the variable or schema root with which it starts and the schema element to which it refers. Attributes bid and eid encode expression $expr_1$, and attributes $bid2$ and $eid2$ encode $expr_2$. Attribute op designates the operator θ.

The $Mapping$ relation is used to encode mappings. Attribute mid is a unique identifier, while attributes $forQ$ and $conQ$ specify the queries in the **foreach** and **exists** part respectively. The expressions in the select clause of the two queries of the mapping are encoded in the $Correspondence$ relation in a way similar to relation $Conditions$, but with the op attribute taken by default to be "=". In general, the *Correspondence* and *Condition* relations are used to encode the W_c set of a mapping $<\mathcal{E}_s,\mathcal{E}_t,W_c>$, while the sets $\mathcal{E}_s$ and $\mathcal{E}_t$ are encoded in the relation *Binding* in conjunction with the relations *Query*, *Mapping* and *Element*.

Example 7.1 *Figure 5 shows an instance of the meta-data schema that encodes the* `Pdb` *and* `EUdb` *data source schemas and mapping m_3 of Figure 1.*

Since mappings are expressed as inter-schema constraints, one could use the same mapping storage mechanism to also store intra-schema constraints, e.g., foreign keys, and use them in queries.

7.2. Annotating the Instance Data

Annotations are not part of the nested relational model and are not supported directly by the nested relational queries. Hence, to use annotations in queries, we implement functions $getElAnnot(v)$ and $getMapAnnot(v)$, which return, respectively, the element annotation and the set of mapping annotations of a value v in the tagged instance. The advantage of using functions is implementation independence. In an XML data repository, for example, annotations may be implemented as attributes on elements, while in a relational repository as auxiliary tables.

We have already studied and described [21] the population of a schema through a set of GLAV mappings, but that work did not support annotations. To cope with this problem, we first rewrite each mapping by enhancing its select clauses in order to explicitly generate the annotations. In particular, given a mapping m, for every expression $expr$ referring to element e in the select clause of the **exists** part, expressions $getElAnnot(expr)$ and $getMapAnnot(expr)$ are also appended to this clause and constants $'e'$ and $'m'$ are appended to the select clause of the **foreach** part query. The rewritten mappings can then be executed as described in [21] to generate the annotated instance.

Element

eid	name	type	parent	db
e0	EU	Rcd	–	EUdb
e1	postings	Set	e0	EUdb
e2	*	Rcd	e1	EUdb
e3	hid	Str	e2	EUdb
e4	levels	Str	e2	EUdb
e5	totalVal	Str	e2	EUdb
e6	agents	Set	e2	EUdb
e7	*	Rcd	e6	EUdb
e8	agentName	Str	e7	EUdb
e9	agentPhone	Str	e7	EUdb
e30	Portal	Rcd	–	Pdb
e31	estates	Set	e30	Pdb
e32	*	Rcd	e31	Pdb
e33	hid	Str	e32	Pdb
e34	stories	Str	e32	Pdb
e35	value	Str	e32	Pdb
e36	contact	Str	e32	Pdb
e37	contacts	Set	e30	Pdb
e38	*	Rcd	e37	Pdb
e39	title	Str	e38	Pdb
e40	phone	Str	e38	Pdb

Query

qid
q0
q1

Mapping

mid	forQ	conQ
m3	q0	q1

Condition

qid	bid	eid	op	bid2	eid2
q1	e	e36	=	c	e39

Binding

bid	qid	eid	prev
r_1	q0	e0	–
p	q0	e1	r_1
a	q0	e6	p
r_2	q1	e30	–
e	q1	e31	r_2
c	q1	e37	r_2

Correspondence

mid	forBid	forEid	conBid	conEid
m3	p	e3	e	e33
m3	p	e4	e	e34
m3	p	e5	e	e35
m3	a	e8	c	e39
m3	a	e9	c	e40

Figure 5. Meta-data storage implementation.

Example 7.2 *Mapping m_2 of Figure 1 is rewritten to:*

foreach
select h.`hid`, h.`stories`, h.`price`, f, a.`phone`,
$'/Portal/estates/hid', 'm_2'$,
$'/Portal/estates/stories', 'm_2'$, ...
from `US.houses` h, `US.agents` a, a.`title`→`firm` f
where h.`aid`=a.`aid`
exists
select e.`hid`, e.`stories`, e.`value`, c.`title`, c.`phone`
$getElAnot(e.\texttt{hid})$, $getMapAnnot(e.\texttt{hid})$,
$getElAnot(e.\texttt{stories})$, $getMapAnnot(e.\texttt{stories})$, ...
from `Portal.estates` e, `Portal.contacts` c
where e.`contact`=c.`title`

7.3. Translating MXQL queries

This section describes how an MXQL query is translated into a query that can be executed over a tagged instance. The first step is to translate every e@map and e @elem operation in the query to a function call of $getMapAnnot(e)$ or $getElAnnot(e)$, respectively. Mapping predicates are processed next. If a mapping predicate contains constants, each is replaced by a new variable, and a condition is added in the where clause requiring that the variable be equal to the constant value.

Example 7.3 *Applying these steps to query in Example 5.7 gives:*

select s.`hid`, m
from `Portal.estates` s, `Portal.contacts` c,
$getMapAnnot(c.\texttt{title})$ m_v

where s.contact=c.title and $<db{:}e{\rightarrow}m{\rightarrow}db_2{:}e_2>$
and $m{=}m_v$ and $e{=}'$US/agents/title/firm$'$
and $e_2{=}getElAnnot(c.\texttt{title})$ and $db{=}'$US$'$
and $db_2{=}'$Pdb$'$

The bindings of the variables in the mapping predicates are generated next. In particular, for the predicate $<db{:}e{\rightarrow}m{\rightarrow}db_2{:}e_2>$, variables e and e_2 are bound to relation $Element$ and m to relation $Mapping$. Existing references to these variables in the select and the where clauses are then replaced by references to the identifier attribute of the corresponding table. For instance, since variable e is bound to $Element$, every expression involving variable e is replaced by $e.eid$. Variables db and db_2 are finally replaced by expression $e.db$ and $e_2.db$, respectively.

Example 7.4 *Applying the above steps to the query of Example 7.3 results in:*

select s.hid, $m.mid$
from Portal.estates s, Portal.contacts c, $Mapping\ m$,
$getMapAnnot(c.\texttt{title})\ v_n$, $Element\ e$, $Element\ e_2$
where s.contact=c.title and $<e.db{:}e{\rightarrow}m{\rightarrow}e_2.db{:}e_2>$
and $m_v{=}m.mid$ and $e.eid{=}'$US/agents/title/firm$'$
and $getElAnnot(c.\texttt{title}){=}e_2.eid$ and $e.db{=}'$US$'$
and $e_2.db{=}'$Pdb$'$

The last step is to specify how the variables of the mapping predicate relate to one another. The relationship between the element e and the mapping m depends on the mapping predicate. If it is a single arrow predicate, it means that the element e is used in the select clause of the **foreach** part of the mapping m. This translates to the conditions $e.eid{=}o.forEid$ and $o.mid{=}m.mid$ where o is a new variable that binds to $Correspondence$. If we have a double-arrow predicate, then the case is similar, but the requirement for e is to be used either in the select clause as before or in the where clause. In the latter case, this translates to the existence of a corresponding entry in the $Condition$ relation. The association between mapping m and element e_2 is analogous. The mapping predicates are finally removed.

Example 7.5 *The query of Example 7.4 becomes:*

select s.hid, $m.mid$
from Portal.estates s, Portal.contacts c,
$getMapAnnot(c.\texttt{title})\ m_v$, $Element\ e$
$Mapping\ m$, $Element\ e_2$, $Correspondence\ o$
where s.contact=c.title and $m_v{=}m.mid$
and $getElAnnot(c.\texttt{title}){=}e_2.eid$ and $e.db{=}'$US$'$
and $e.eid{=}'$US/agents/title/firm$'$ and $o.mid{=}m.mid$
and $o.forEid{=}e.eid$ and $o.conEid{=}e_2.eid$ and $e_2.db{=}'$Pdb$'$

The advantage of using MXQL queries over the rewritten form we have just described is that the user does not need to be aware of the details of the meta-data storage schema. Instead, she only has to declaratively specify her requirements in the query. Another advantage is that the storage method can be modified without altering the application queries.

8. Experience

As an application scenario for our framework, we used the generation of a real estate portal that integrates and materializes data from five popular real estate Web sites (Yahoo, NK Realtors, Winderemere, Westfall and Homeseekers) with an average schema size of 55 elements. The information extracted from the Web sites consists of a total of 14.3MB of XML data (10,000 real estate listings). This information was mapped through a number of nested relational mappings to an integrated schema having 135 elements. Execution of the mappings generated an integrated instance of 14.5MB, which is slightly larger than the total size of the instance data. It is larger because many pieces of information from the data sources were represented more than once in the portal instance. For example, the contact phone number from the Yahoo data source was mapped to both the `business` and the `home` phone in the integrated schema.

The mappings were given to a pre-processor that rewrote them in order to generate annotations (as described in Section 7.2). The re-written mappings were executed on the sources and generated the integrated instance. The new annotated instance was 3 MB larger than the instance without the annotations. This was expected since every XML element carries its annotations, which are represented as XML attributes. Exploiting the fact that the generation methodology [21] produces a data instance in Partition Normal Form, we were able to avoid storing mapping annotations on the children of a complex type value elements since they are the same as the annotations of their parents. This reduced the space overhead of the 3MB to only 0.8MB (i.e., 5.5% of the size of the integrated instance). The schemas and the mappings were encoded as described in Section 7.1, and were inserted in the annotated instance, increasing its size by 0.3MB. We performed a number of experiments with different sizes of source data, and the results showed that the increase in the space of the integrated instance, due to the annotations, was approximatelly 5.5% in all the cases. The increase in space caused by storing schemas and mappings was approximately 0.3MB. The real estate sites we used had very little overlap, i.e., few entries appeared in more than one data sources. We mapped parts of the data of Windermer to Westfall and Homeseekers, and parts of the Yahoo data to NK Realtors, so that different information about the same real estate entry would appear in different sources. We generated the integrated instance as before, and we noticed that the extra space needed by the annotations went down to 4.9%. This means that the space overhead in the annotated instance is less if the sources have overlapping information. Furthermore, we expect that the annotation space overhead should decrease even further if the number of nested sets in the integrated schemas increases.

We executed a number of MXQL queries over the annotated instance, but we noticed no significant execution time increase. Furthermore, MXQL queries helped identify the meaning of some elements in the integrated schemas in a

way similar to the one described in Example 5.6. In addition, they helped detect ill-defined mappings. For example, we noticed that the sub-element `housesInNeighborhood` of a `house` element, in some cases, contained houses that were from different states. In investigating the problem, we executed the following MXQL query

select db, e from where $<db{:}e \Rightarrow m \Rightarrow' Portal' {:} e>$ and
$e='/Portal/house/housesInNeighborhood'$

For the Homeseekers data source, it returned two elements only: the `hid` and the `neighborhood`. Indeed, the Homeseekers mapping was computing the neighboring houses by performing a self-join only on the element `neighborhood`. Unfortunately, neighborhoods with the same name could appear in different states, thus generating the misleading data. When the mapping was updated to join on city, state, and neighborhood, the problem was corrected.

In addition, MXQL helped us judge the accuracy of the mappings. We noticed, for example, that for some houses the high, middle and elementary school districts were the same, although for other houses in the same area they were not. In querying the mappings of the latter values, we noticed that, for the mappings originating from the Realtors data source, all three elements were retrieving their values from a single element `schoolDistrict`, since the Realtors source was not separating elementary, middle and high school districts.

These experiments demonstrate the importance of detecting and managing the schema level origin of the data, as well as the transformations that have generated this data.

9. Conclusion and Future Work

In this work, we considered the problem of representing and querying data transformations. To achieve this, we elevated schemas and mappings to first-class citizens and we used annotations to associate the actual data with its metadata information. We proposed a language for querying this information that considers provenance at the schema level avoiding complexity issues inherent in data-level provenance. To the best of our knowledge, this is the first proposal to consider managing and querying of not only the origin of the data but also the declarative transformations through which the data has been derived.

In the current work, we assumed that mappings were used to materialize an integrated instance. However, that instance may also be virtual. It is a among our next steps to investigate this issue and study the semantics of query rewriting and query answering in such a setting.

Acknowledgments: We would like to thank Wang-Chiew Tan and Phil Bernstein for their useful comments.

References

[1] P. Bernstein, A. Levy, and R. Pottinger. A Vision for Management of Complex Models. *SIGMOD Record*, 29(4):55–63, 2000.

[2] D. Bhagwat, L. Chiticatiu, W. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB*, pages 900–911, 2004.

[3] S. Bowers, L. Delcambre, and D. Maier. Superimposed Schematics: Introducing E-R Structure for In-Situ Information Selections. In *ER*, pages 90–104, 2002.

[4] S. Bressan, C. H. Goh, K. Fynn, M. J. Jakobisiak, K. Hussein, H. B. Kon, T. Lee, S. E. Madnick, T. Pena, J. Qu, A. W. Shum, and M. Siegel. The COntext INterchange Mediator Prototype. In *SIGMOD*, pages 525–527, 1997.

[5] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, pages 316–330, 2001.

[6] P. Buneman, S. Khanna, and W. Tan. On Propagation and Deletion of Annotations Through Views. In *PODS*, pages 150–158, 2002.

[7] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *ICDE*, pages 4–19, 1998.

[8] Y. Cui and J. Widom. Practical Lineage Tracing in Data Warehouses. In *ICDE*, pages 367–378, 2000.

[9] R. Fagin, P. Kolaitis, L. Popa, and W. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *PODS*, pages 83–94, 2004.

[10] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *ICDT*, pages 207–224, 2003.

[11] H. Fan and A. Poulovassilis. Tracing Data Lineage Using Schema Transformation Pathways. In *KTSW*, pages 64–79, 2003.

[12] M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *AAAI*, pages 67–73, 1999.

[13] N. I. Hachem, K. Qiu, M. A. Gennert, and M. O. Ward. Managing Derived Data in the Gaea Scientific DBMS. In *VLDB*, pages 1–12, 1993.

[14] J. Kaha, M. Koivunen, E. Prud'Hommeaux, and R. R. Swick. Annotea: an open RDF infrastructure for shared Web annotations. In *WWW*, pages 623–632, May 2001.

[15] L. V. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL: A Language for Interoperability in Multiple Relational Databases. In *VLDB*, pages 239–250, 1996.

[16] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[17] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *PODS*, pages 95–104, 1995.

[18] J. Madhavan and A. Y. Halevy. Composing Mappings Among Data Sources. In *VLDB*, pages 572–583, 2003.

[19] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD*, pages 193–204, June 9–12 2003.

[20] R. J. Miller, L. M. Haas, and M. Hernandez. Schema Mapping as Query Discovery. In *VLDB*, pages 77–88, Sept. 2000.

[21] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.

[22] I. Tatarinov and A. Y. Halevy. Efficient Query Reformulation in Peer-Data Management Systems. In *SIGMOD*, pages 539–550, 2004.

[23] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation Under Evolving Schemas. In *VLDB*, pages 584–595, Sept. 2003.

[24] Y. R. Wang and S. E. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *VLDB*, pages 519–538, 1990.

[25] C. Yu and L. Popa. Constraint-Based XML Query Rewriting For Data Integration. In *SIGMOD*, pages 371–382, 2004.