Nested Mappings: Schema Mapping Reloaded

Ariel Fuxman*
University of Toronto
afuxman@cs.toronto.edu

Mauricio A. Hernandez
IBM Almaden Research Center
mauricio@almaden.ibm.com

Howard Ho
IBM Almaden Research Center
ho@almaden.ibm.com

Renee J. Miller
University of Toronto
miller@cs.toronto.edu

Paolo Papotti*
Università Roma Tre
papotti@dia.uniroma3.it

Lucian Popa
IBM Almaden Research Center
lucian@almaden.ibm.com

ABSTRACT

Many problems in information integration rely on specifications, called *schema mappings*, that model the relationships between schemas. Schema mappings for both relational and nested data are well-known. In this work, we present a new formalism for schema mapping that extends these existing formalisms in two significant ways. First, our *nested mappings* allow for nesting and correlation of mappings. This results in a natural programming paradigm that often yields more accurate specifications. In particular, we show that nested mappings can naturally preserve correlations among data that existing mapping formalisms cannot. We also show that using nested mappings for purposes of exchanging data from a source to a target will result in less redundancy in the target data. The second extension to the mapping formalism is the ability to express, in a declarative way, grouping and data merging semantics. This semantics can be easily changed and customized to the integration task at hand. We present a new algorithm for the automatic generation of nested mappings from schema matchings (that is, simple element-to-element correspondences between schemas). We have implemented this algorithm, along with algorithms for the generation of transformation queries (e.g., XQuery) based on the nested mapping specification. We show that the generation algorithms scale well to large, highly nested schemas. We also show that using nested mappings in data exchange can drastically reduce the execution cost of producing a target instance, particularly over large data sources, and can also dramatically improve the quality of the generated data.

1. INTRODUCTION

Many problems in information integration rely on specifications that model the relationships between schemas. These specifications, called *schema mappings*, play a central role in both data integration and in data exchange. We consider schema mappings over pairs of schemas that express a relation on the sets of instances of two schemas. The benefits of using declarative formalisms for schema mappings are well-known. Such formalisms have the promise of providing a high-level, natural programming paradigm for mappings, and can facilitate customization, evolution, and use in different integration tasks. Declarative schema mapping formalisms have been used to provide formal semantics for data exchange [9, 1], data integration [14], peer data management [12, 5], and model management operators [18] such as composition [15, 8, 21] and inversion [7].

We start by examining the most widely used formalisms for schema mappings. For relational schemas, these are based on *source-to-target tuple-generating dependencies* (source-to-target tgds) [9] or, equivalently, *GLAV (global-and-local-as-view) assertions* [10, 14]. For schemas containing nested data (including XML schemas), direct extensions have been proposed [24, 27]. We consider the expressiveness of these mappings to understand what semantics they can, and more importantly cannot, capture. In addition, we study to what extent these formalisms meet the goal of providing a natural programming paradigm for mappings. In particular, we identify several issues that can lead to inaccurate or underspecified mappings. Furthermore, we show how existing mapping specifications may be fragmented into many small, overlapping formulas where the overlap may lead to redundant computation, may hinder human understanding of the mappings and, ultimately, may limit the effectiveness of mapping tools.

To alleviate these issues, we propose a new mapping formalism, *nested mappings*, that allows for nesting and correlation of mappings. Nested mappings permit the expression of powerful grouping and data merging semantics declaratively within the mapping. We show that nested mappings yield more accurate specifications, and when used in data exchange can improve the quality of the exchanged data.

1.1 Current Schema Mapping Formalisms

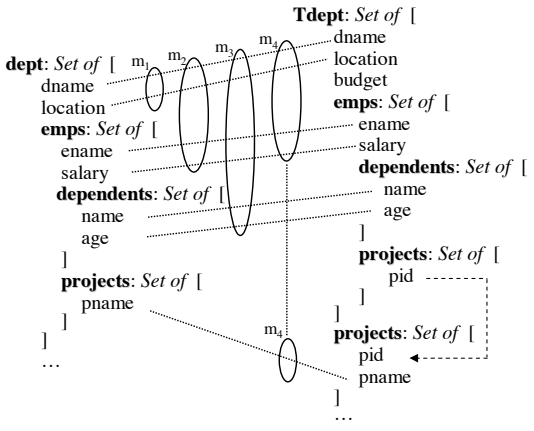
Source-to-target tgds and GLAV assertions are constraints between relational schemas. They are expressive enough to represent, in a declarative way, many of the relational schema mappings of interest. In this work, we start by examining an extension of source-to-target tgds designed for schemas with nested data that is based on path-conjunctive constraints [23], and that have been used in systems for data exchange [24], data integration [27], and schema evolution [26, 28]. We refer to such mappings as *basic mappings*. They form the basic building blocks for our subsequent nested mappings.[1]

To illustrate the use of basic mappings, consider the mapping example shown in Figure 1. The source schema, illustrated on the left, is a nested schema describing departments with their employees and projects. There is a top-level set of department records, and each department record has a (nested) set of employee records. There is additional nesting in that each employee has a set of dependents and a set of projects. Each set can be empty, in general. The target schema, shown on the right, is a slight variation of the source schema.

*Work done while at IBM Almaden Research Center

VLDB '06, September 12-15, 2006, Seoul, Korea.

[1] In the literature these *basic mappings* have sometimes been referred to as nested constraints or dependencies since they are constraints on nested data. However, the mappings themselves have no structure or nesting. Hence, we will use the term *basic* to distinguish them from the more structured *nested mappings* that we are proposing.



m_1: "for every department element, map department info":
for d in dept ⇒ exists d′ in Tdept where (d′.dname=d.dname ∧ d′.location=d.location)

m_2: "for every department element **with** employees, map department and employee info":
for d in dept, e in d.emps ⇒ exists d′ in Tdept, e′ in d′.emps where ...

m_3: "for every department element **with** employees **with** dependents, …"

m_4: "for every department element **with** employees **with** projects, …"

Figure 1: Multiple "small" mappings.

The formulas that are sketched below the schemas are examples of *basic mappings*. They are constraints that describe, in a declarative way, the mapping requirements. These formulas may be generated by a tool such as Clio [24] from the lines (or, *correspondences*) between schema elements, or may be written by a human expert and interpreted by a model management tool such as Moda [18] or other integration tools such as Piazza [12]. (We will give a precise semantics for the schema and basic mapping notation in Section 2. The exact details are not essential for this introductory discussion.)

Each formula (that is, each m_i) deals with one possible "case" in the source data (where each case is expressed by a conjunction of navigation paths joined in certain ways). In order to cover all possible cases of interest, we need many such formulas. However, many of the cases overlap (i.e., have common navigation paths). Hence, common mapping behavior must be repeated in many formulas.

For example, the formula m_2 must repeat the mapping behavior that m_1 already specifies for department data (although m_2 does it in a more specialized context). Otherwise, if we specify in m_2 only the mapping behavior for employees, we lose in the target the *association* that exists in the source between employees and their departments (since there is no correlation between m_1 and m_2). At the same time, m_1 cannot be eliminated from the specification, since it deals with departments in general (that are not required to have employees). Also, in the example, m_3 and m_4 contain a common mapping behavior for employees and departments (but they differ in that they map different components of employees: dependents and projects).

Such formulas are (relatively) easy to generate and reason about. This is, partly, why they have been widely used in research. However, the number of formulas quickly increases with large schemas, leading to an explosion in the size of the specification. This explosion as well as the overlap in behavior causes significant usability problems for human experts and for tools using these specifications in practice.

Inefficiency in execution In a naive use of basic mappings, each mapping formula may be interpreted separately. Optimization of these mappings requires sophisticated techniques that deduce the correlations and common subexpressions within the mappings.

Redundancy in the specification When using basic mappings in data exchange, the same piece of data may be generated multiple times in the target due to the multiple formulas. In addition to possible run-time inefficiency, this puts additional burden on methods for duplicate elimination or data merging. For the above example, an employee may be generated three times in the target: once for m_2 (with an empty set of dependents and an empty set of projects), once for m_3 (with a non-empty set of dependents) and once for m_4 (with a non-empty set of projects). Merging of the three employee records into one is more than just duplicate elimination: it requires merging of two nested sets as well. Furthermore, this raises the question of when to merge in general (since this is not expressed in any way by the mapping formulas of Figure 1). This brings us to the next point.

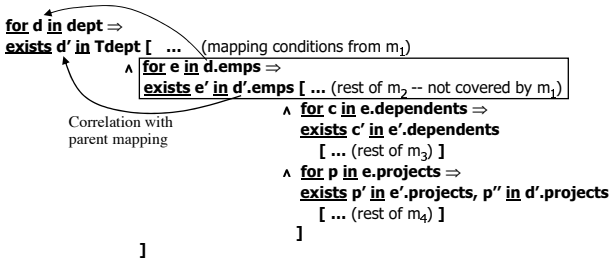


Figure 2: Nested mapping.

Underspecified grouping semantics The formula m_2 requires that for every department and for every employee record in the source there must exist, in the target, a "copy" of the department record with a "copy" of the employee record nested underneath. However, it is left unspecified whether to group multiple employees that are common for a given department name (`dname`), or whether to group by other fields, or whether not to group at all. Again, one of the reasons for this lack of expressive power is the simplicity of these basic mapping formulas. In an early version of Clio [24], a default grouping behavior is used based on partitioned normal form (PNF) which always groups nested sets of elements by all the atomic elements at the upper levels. For example, under the PNF semantics, employees will be grouped by `dname` and `location` (assuming that `budget` is not mapped and its value is null). In effect, the semantics of the transformation is specified in two parts: first the mapping formulas, and then the implicit (PNF-based) grouping semantics. An important limitation of this approach is that the default grouping semantics is not specified declaratively, and it cannot be easily changed or customized when it is not the desired semantics.

1.2 Nested Mappings

In order to address the above issues, we propose an extension to basic mappings that is based on arbitrary nesting of mapping formulas within other mapping formulas. We shall call this formalism the language of *nested mappings*. As a first observation, it can be argued that nested mappings offer a more natural programming paradigm for mapping tasks, since human users tend to design a mapping from top to bottom, component-wise: define first how the top components of a schema relate, then define, recursively, via nested submappings, how the subcomponents relate, and so on. For our earlier example, the corresponding nested mapping is illustrated in Figure 2. The nested mapping relates, at the top-level, source departments with target departments; it then continues, in this context of a department-to-department mapping, with a submapping relating the corresponding employees, which then continues with submappings for dependents and projects. At each level, there are correlations between the current submapping and the upper-level mappings. In particular, nothing is repeated from the upper level, but instead reused.

Advantages of nested mappings Nested mappings overcome (to a large extent) the previous shortcomings of basic mappings. First, we need fewer formulas and overall produce a more natural and accurate specification. For our example, one nested mapping replaces four ba-

sic mappings. In general, we may still need multiple nested mappings (one common situation is when we have multiple data sources). Second, by using nested mappings, we are able to produce more efficient data exchange queries. This is because nested mappings factor out common subexpressions, so we can more easily optimize the number of passes over the same input data. For our example, department records can be scanned only once, and the entire work involving the subelements can be done in the same pass (by the submappings). The execution will also generate much less redundancy in the target data. An employee is generated once, and all dependents and projects are added together (by the two corresponding submappings).

Nested mappings also have a natural, built-in, grouping behavior, that follows the grouping of data in the source. For example, the above nested mapping requires that all the employees in the target are grouped in the same way as they are in the source. This grouping behavior is ideal for mappings between two similar schemas (which is common in the important case of schema evolution) where much of the data should be mapped using the identity (or mostly-identity) mapping. For more complex restructuring tasks, additional grouping behavior may need to be specified. We use a simple, but powerful, mechanism for adding such grouping behavior by using explicit grouping functions (a restricted form of Skolem functions).

Summary of Contributions

• We propose a nested mapping formalism for representing the relationship between schemas for relational or nested data (Section 2).

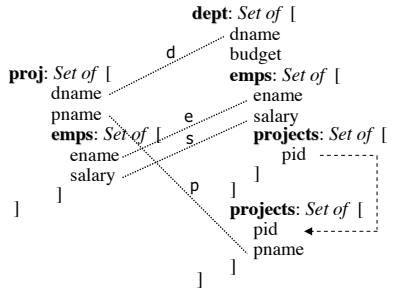
• We propose an algorithm for generating nested mappings from matchings, or correspondences, between schema elements. The nested nature of the mappings makes this generation task more challenging than in the case of basic mappings (Section 3).

• We give an algorithm for the generation of data transformation queries that implement data exchange based on nested mapping specifications. Notably our algorithm can handle all nested mappings, including those generated by our mapping algorithm as well as arbitrary customizations of these mappings, which may be made, for example, by a user to capture specialized grouping semantics (Section 4).

• We show experimentally that the use of nested mappings in data exchange can drastically reduce the execution cost of producing a target instance, and can also dramatically improve the quality of the generated data. We show examples of important grouping semantics that cannot be captured by basic mappings, and we empirically show that underspecified basic mappings may lead to significant redundancy in data exchange (Section 5).

Related Work Schema mappings are so important in information integration that many mapping formalisms have been proposed for different tasks. Here we mention only a few. The important role of Skolem functions for merging data has been recognized in a number of approaches [13, 22] and Skolem functions appear explicitly as part of the XML-QL query language [6]. Work on model management has used embedded dependencies (similar to our basic mappings) which may be augmented with Skolem functions [3, 18]. HePToX [5] uses a datalog-like language that supports nested data and allows Skolem functions, but mappings cannot be nested or correlated. Grammars have been used to specify mappings for (recursive) DTDs [2]. While most of these formalisms support nested data, to the best of our knowledge, none of the existing declarative formalisms support the expression of nesting between mappings.

Our nested mappings are strictly more expressive than basic mappings. At the same time, they are less expressive than languages used for composition [8, 21]. In particular, if we restrict ourselves to the relational model (for comparison purposes), nested mappings are a strict sublanguage of the second-order tgds (SO tgds) introduced in [8]: every nested mapping can be rewritten, via Skolemization, into an equivalent SO tgd (but not vice-versa). However, this rewriting would erase the nesting structure of the mapping, and it is not clear



m_1 for p in proj ⇒
exists d′ in dept, p′ in d′.projects
where d′.dname=p.dname ∧ p′.pname=p.pname

m_2 for p in proj, e in p.emps ⇒
exists d′ in dept, p′ in d′.projects, e′ in d′.emps, p″ in e′.projects
where p″.pid=p′.pid ∧
d′.dname=p.dname ∧ p′.pname=p.pname ∧
e′.ename=e.ename ∧ e′.salary=e.salary

Figure 3: A mapping scenario with two basic mappings.

to what extent such nesting could be extracted from an arbitrary SO tgd. The language of SO tgds does not allow nesting of formulas, but instead allows correlation of formulas via arbitrary Skolem functions, a more powerful but arguably less user-friendly programming concept. We also note that there is no known algorithm for generating SO tgds; the algorithm for generating nested mappings that we propose here can, however, be seen as a step in that direction, since nested mappings correspond to a form of SO tgds.

Many industry tools such as BizTalk Mapper, IBM WebSphere Data Stage TX, and Stylus Studio's XML Mapper support the development (by a programmer) of mappings. Some support nested mappings, though in more procedural languages. However, most, if not all, of the work done to express such mappings is manual. Generation of mappings (with no nesting) has been considered in the TranSem system [20], Clio [19, 24] and HePToX [5]. Also, Bohannon et al. [4] consider the generation of information preserving mappings (based on path mappings). Our work extends the Clio mapping generation algorithm to produce nested mappings.

As part of our generation algorithm, we identify common expressions within mappings. Our goal is to identify possible correlations between mappings that can be exploited to produce more accurate mapping specifications. Our techniques are in the same spirit of work on identifying common expressions within complex queries for use in query optimization [25]. However, unlike query optimization which must necessarily preserve query equivalence, our techniques lead to mappings with better semantics, and so do not preserve equivalence.

Notably the generation of efficient queries for data exchange is not considered in work like Piazza [12] and HePToX [5] which instead focus on query generation for data integration. In model management [18, 3], query or code generation for data exchange has been considered for embedded dependencies. Clio [24] generates XQuery, XSLT, SQL/XML, and SQL queries for basic mappings.

2. MAPPINGS WITHIN MAPPINGS

In this section, we fix the notation and terminology for schemas and mappings based on our previous work [24, 28]. Furthermore, we take a closer look at the qualitative differences between basic mappings and nested mappings.

2.1 Basic Mappings

Consider the mapping scenario illustrated in Figure 3. The two schemas in the figure (source and target) are shown in a nested relational representation that can be used as a common abstraction for

relational and XML schemas (and other hierarchical set-oriented data formats). This representation is based on sets and records that can be arbitrarily nested. In the source schema, `proj` is a set of records with two atomic components, `dname` (department name) and `pname` (project name), and a set-valued component, `emps`, that represents a (nested) set of employee records. The target schema is a reorganization of the source: at the top-level we have a set of department records, with two nested sets of employee and project records. Moreover, each employee can have its own set of project ids (pids), which must appear at the department level (this is required by the foreign key shown in the figure with an arrow).

Formally, a *schema* is a set of labels (also called roots), each with an associated *type* τ, defined by: $\tau ::= \mathsf{Str} \mid \mathsf{Int} \mid \mathsf{SetOf}\ \tau \mid [\, l_1 : \tau_1, \ldots, l_n : \tau_n]$, where $l_1, \ldots, l_n$ are labels.[2] We point out that this is only a simplified abstraction: in the system that we implemented, we also deal with choice types, optional elements, nullable elements, etc. However, the presence of these additional features does not essentially change the formalism.

In Figure 3, we also show two basic mappings that can be used to describe the relationship between the source and the target schemas. The first one, m_1, is a constraint that maps department and project names in the source (independently of whether there exist any employees in `emps`) to corresponding elements in the target. The second one, m_2, is a constraint that maps department and project names *and* their employees (whenever such employees exist).

In the figure, we use a "query-like" notation, with variables bound to set-type elements. Each variable can be a record and hence contain multiple components. Correspondences between schema elements (e.g., `dname` to `dname`) are captured by equalities between such components (e.g., $d'.dname = p.dname$). These equalities are grouped in the where clause that follows the exists clause of a mapping. Moreover, equalities can also be used to express join conditions (or other predicates) in the source or in the target. For example, see the requirement on *pid* in m_2 that appears in the same where clause.

Logic-based notation Alternatively, we will use a "logic-based" notation for mappings that quantifies each individual component in a record as a variable. In particular, nested sets are explicitly identified by variables. Each mapping is an implication between a set of atomic formulas over the source schema and a set of atomic formulas over the target schema. Each atomic formula is of the form $e(x_1, \ldots, x_n)$ where e denotes a set, and $x_1, \ldots, x_n$ are variables.[3] The main difference from the traditional relational atomic formulas is that e may be a top-level set (e.g., `proj`), or it may be a variable (in order to denote sets that are nested inside other sets). We will write the atomic variables in lower-case and the set variables in upper-case. The formulas corresponding to the mappings m_1 and m_2 of Figure 3 are:

$$
\begin{array}{ll}
m_1: & \texttt{proj}(d, p, E_s) \rightarrow \texttt{dept}(d, ?b, ?E, ?P) \wedge P(?x, p) \\
m_2: & \texttt{proj}(d, p, E_s) \wedge E_s(e, s) \\
& \rightarrow \texttt{dept}(d, ?b, ?E, ?P) \wedge E(e, s, ?P') \wedge P'(?x) \wedge P(x, p)
\end{array}
$$

For each formula, the variables on the left of the implication are assumed to be universally quantified. The variables on the right that do not appear on the left are assumed to be existentially quantified. For clarity, we omit the quantifiers and use a question mark in front of the first occurrence of an existentially-quantified variable.

To illustrate, in m_2, the variable E_s denotes the nested set of employee records (inside a tuple in the top-level set `proj`). The variables E, P, and P' are also set variables, but existentially quantified. The variables b (for budget) and x (project id) are existentially quantified as well (but atomic). The meaning of m_2 is: for every source tuple (d, p, E_s) in `proj`, and for every tuple (e, s) in the set E_s, there must exist four tuples in the target as follows. First, we must have a tuple (d, b, E, P) in `dept`, where b is some "unknown" budget, E identifies a set of employee records, and P identifies a set of project records. Then, there must exist a tuple (e, s, P') in E, where P' identifies a set of project ids. Furthermore, there must exist a tuple (x) in P', where x is an "unknown" project id. Finally, there must exist a tuple (x, p) in the previously mentioned set P, where x is the same project id used in P'. Notice that all data required to be in the target by the mapping satisfies the foreign key for the projects.

[2] In Figure 3, we do not show any of the atomic types.

[3] For simplicity of presentation, we assume strict alternation of set and record types in a schema.

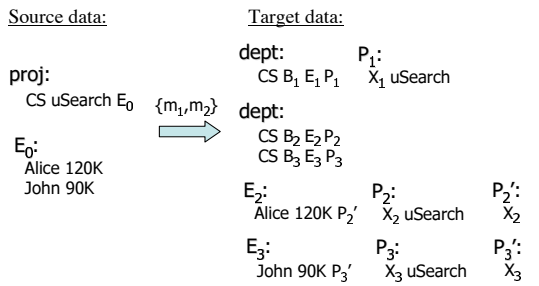


Figure 4: Source and target instances satisfying $\{m_1, m_2\}$.

2.2 Correlating Mappings via Nesting

We now take a look at actual data in order to understand the semantics of basic mappings, and to see why such specification language is not entirely satisfactory. In Figure 4, we show source and target instances that satisfy the constraints m_1 and m_2. In the source, E_0 is a "name", or *set id*, for the nested set of employee records corresponding to the tuple given in `proj`. We assume that every nested set has such an id. Similarly, $E_1, P_1, E_2, \ldots, P_3'$ are set ids in the target instance. The top two target tuples, for `dept` and P_1, respectively, ensure that m_1 is satisfied; the rest are used to satisfy m_2.

In general, for a given source instance, there may be several target instances satisfying the constraints imposed by the mapping specification. Given the specification $\{m_1, m_2\}$, the target instance shown in Figure 4 can be considered to be the most general that can be produced (a *universal solution* [9]), because it is the one that makes the least assumptions. For example, it does not assume that E_1 and E_2 are equal (since this is not required by the specification). However, this target instance may not be satisfactory for a number of reasons. First, there is redundancy in the output: there are three `dept` tuples generated for "CS", for different instantiations of the left-hand sides of m_1 and m_2. Also, there are three project tuples for "uSearch" (although in different sets). Second, there is no grouping of data in the target: E_2 and E_3 are different singleton sets, generated for different instantiations of the left-hand side of m_2 (same for P_2 and P_3). This does not violate the constraints, however, since the mapping specification does not require E_2 and E_3 to be equal.

In Figure 5, we show a target instance that is more "desirable". This instance has no redundant departments or projects, and it maintains the grouping of employees that exists in the source. While this instance satisfies the constraints m_1 and m_2, for the given source instance, it is not *required* by these mappings. In particular, the specification given by $\{m_1, m_2\}$ does not rule out the undesired target instance of Figure 4.

We would like to have a specification that "enforces" correlations such as the ones that appear in the more "desirable" target instance (e.g., that the two source employees appear in the *same* set in the target). In particular, we would like to correlate the mapping m_2 with m_1 so that it *reuses* the set id E for employees that is already asserted by m_1 (along with other existentially-quantified elements in

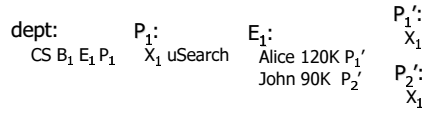


Figure 5: Target data required by the nested mapping n.

m_1), without repeating the common part, which is m_1 itself. This can be done using the following *nested mapping*:

$$\begin{aligned} n: \ & \texttt{proj}(d, p, E_s) \rightarrow \\ & [\ \texttt{dept}(d, ?b, ?E, ?P) \wedge P(?x, p) \\ & \wedge [\ E_s(e, s) \rightarrow E(e, s, ?P') \wedge P'(x)\ ]\ ] \end{aligned}$$

The inner implication in n (the third line) is a *submapping*. We refer to the rest of n as the *outer mapping*. The submapping is correlated to the outer mapping because it reuses the existential variables E and x. In particular, the submapping requires that for every employee tuple in the set E_s (where E_s is bound by the outer mapping), there must exist an employee tuple in the set E, which is also bound by the outer mapping. Also, there must exist a project tuple in the set P' associated to this employee, and the project id must be precisely the one (x) already required by the outer mapping. Note that P' is now existentially quantified and bound in the inner mapping.

A fundamental observation about the nested mapping n is that the "undesirable" target instance of Figure 4 does not satisfy its requirements. For example, when we apply the outer mapping of n to $proj(CS, uSearch, E_0)$, we require $dept(CS, B_1, E_1, P_1)$ to be in the target. Now, when we apply the submapping to $E_0(Alice, 120K)$ and $E_0(John, 90K)$, we *must* have tuples for $Alice$ and $John$ within the same set E_1. The nested mapping explicitly rules out the target instance of Figure 4, and is a tighter specification for the desired schema mapping.

Another important observation is that there is no set of basic mappings that is equivalent to the above nested mapping. (It is not hard to show this and we leave the details for a larger version of this paper.) Thus, the language of nested mappings is strictly more expressive than that of basic mappings.

Finally, we show below the nested mapping in query-like notation. Notice that the variables p, d' and p' from the outer level are being reused in the inner level.

n: for p in proj $\Rightarrow$
exists d' in dept, p' in d'.projects
where d'.dname=p.dname $\wedge$ p'.pname=p.pname $\wedge$
(for e in p.emps $\Rightarrow$
exists e' in d'.emps, p'' in e'.projects
where p''.pid=p'.pid $\wedge$
e'.ename=e.ename $\wedge$ e'.salary=e.salary)

2.3 Grouping and Skolem Functions

As seen in the above example, nested mappings can take advantage of the grouping that exists in the source, and require the target data to have a similar grouping. In the example, all the employees that are nested inside one source tuple are required to be nested inside the corresponding target tuple. In this section, we show how a restricted form of *Skolem functions* can be used to model groupings of data that may not be present in the source.

To illustrate, consider again the source schema in Figure 3. In Figure 6, we show source and target data for this schema. On the left, we show a source instance that extends the one of Figure 4. In particular, the "CS" department is associated with two different projects instead of one. On the right, we show a desired target instance, where projects are grouped by department name. This target instance is not required by the nested mapping n, which allows target instances where we may have multiple department tuples (with the same `dname` value), each with a singleton set containing one project. In other words, the source data is flat and, consequently, the target data is flat (as far as the relationship between departments and projects goes). Furthermore, the above nested mapping does not merge sets of employees that appear in different source tuples with the same department name, in contrast with the target instance shown in Figure 6.

Suppose now that we *do* want to group into one set all the projects of a department, and also all the projects for each employee in a department. Also, we want to merge all the employees for a given department. To generate such new groupings of data, we need to add something else to the specification, since nesting of mappings alone is not flexible enough to describe such groupings. The mechanism that we add is that of Skolem functions for *set* elements. Intuitively, such functions can express that certain sets in the target must be functions of certain values from the source. For our example, to express the desired grouping, we enrich the nested mapping with three Skolem functions for the three nested set types in the target, as follows:

n': for p in proj $\Rightarrow$
exists d' in dept, p' in d'.projects
where d'.dname=p.dname $\wedge$ p'.pname=p.pname $\wedge$
d'**.emps=E[**p**.dname]** $\wedge$ d'**.projects=P[**p**.dname]** $\wedge$
(for e in p.emps $\Rightarrow$
exists e' in d'.emps, p'' in e'.projects
where p''.pid=p'.pid $\wedge$
e'.ename=e.ename $\wedge$ e'.salary=e.salary $\wedge$
e'**.projects=P'[**p**.dname,**e**.ename]**)

The new mapping constrains the target set of projects to be a function of only department name: P[p.dname]. Also, there must be only one set of employees per department name, E[p.dname], meaning that multiple sets of employees (for different source tuples with the same department name) must be merged into one. Similarly, all projects of an employee in a department must be merged into one set.

More concretely, for the source tuple $proj(CS, uSearch, E_0)$ of Figure 6, the outer mapping of n' requires that the target contains $dept(CS, B_1, E_1, P)$. In addition, E["CS"] (the result of applying the Skolem function E to the value "CS") corresponds to E_1. Due to the inner mapping, the two employees of E_0 ("Alice" and "John") must be in E_1. Now consider the source tuple $(CS, iMap, E_0')$. The mapping n' requires the employees working on the "iMap" project (Bob and Alice) to also be within the set E_1. The reason for this is that, according to n', the employees of "iMap" must also be in E["CS"], which is E_1.

Due to lack of space, we omit the precise definition of nested mappings, which is straightforward and follows the examples and intuition given above. We do point out the following natural restriction. The for clause of a submapping can use a correlation variable (i.e., bound in an upper-level mapping) only if that variable is bound in a for clause of the upper-level mapping. (A similar restriction holds for the usage of correlation variables in exists clauses.)

Every nested mapping (with no explicit Skolem functions) is equivalent to one in which default Skolem functions are assigned to all the existentially-quantified set variables (using here the logic-based notation). The default arguments to such Skolem functions are all the universally quantified variables that appear before the set variable.

As an example, the previous nested mapping n is equivalent to one in which the target set of projects nested under each `dept` tuple is determined by a Skolem function of all three components of the input `proj` tuple (i.e, `dname`, `pname`, and `emps`). In other words, there must be a set of target projects for each input `proj` tuple. Of course, this does not require any grouping of projects by departments. However, once we expose them to a user, the Skolem functions can be customized, in order to achieve different grouping behavior (such as the one seen with the earlier mapping n'). This is the approach that we follow in our system: we first generate nested mappings (with no Skolem functions), then apply default Skolemization, which can then be altered in a GUI by a user.

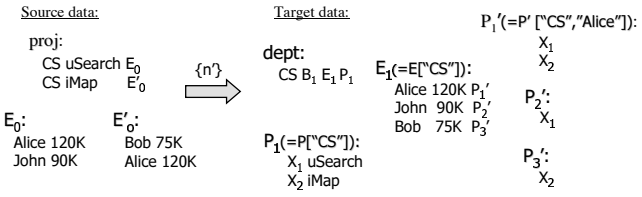


Figure 6: Example data for the nested mapping n'.

Skolem functions and data merging Our example illustrates how one occurrence of a Skolem function permits data to be accumulated into the same set. Furthermore, the same Skolem function may be used in multiple places of a mapping or even across multiple mappings. Thus, different mappings (correlated via Skolem functions) may contribute to the same target sets, effectively achieving data merging. This is a typical requirement in data integration. Hence, Skolem functions are a declarative representation of a powerful array of data merging semantics.

As an interesting example of a set being shared from multiple places consider the case when "Alice" has different salaries ($120K$ and $130K$) in the two tuples in the source of Figure 6. Then our mapping n' requires that there be two different "Alice" tuples in the target (both in the set E_1 = E["CS"]). Moreover, the same set of projects will be constructed for the two Alice tuples since the (projects) set id is a Skolem function (P′) of "CS" and "Alice" (and does not take into account `salary`). This showcases an interesting feature of the mapping language, which is the ability to merge several components of a piece of data while still keeping other components separated (perhaps until further resolution).

3. GENERATION OF NESTED MAPPINGS

In this section, we describe our algorithm for the generation of nested mappings. Given two schemas, a source and a target, and a set of correspondences between atomic elements in the schemas, the algorithm generates a set of nested mappings that "best" reflect the given schemas and correspondences. The first two steps in the algorithm (Section 3.1) follow the generation of basic mappings that we introduced in our previous work [24]. We then describe (Section 3.2) an additional step in which unlikely basic mappings are pruned. This significantly reduces the number of basic mappings. We define when a basic mapping can be nested under another basic mapping in Section 3.3. The pruned basic mappings are then input to the final step in the algorithm to generate nested mappings (Section 3.4).

3.1 Basic Mapping Generation

We now review the generation algorithm for basic mappings [24]. The main concept is that of a *tableau*. Intuitively, tableaux are a way of describing all the "basic" concepts and relationships that exist in a schema. There is a set of tableaux for the source schema and a set of tableaux for the target schema. Each tableau is primarily an encoding of one concept of a schema (here, concept is synonymous to a set type). In addition, each tableau includes all *related* concepts, that is, concepts that must exist together according to the referential constraints of the schema or the parent-child relationships in the schema. This will allow the subsequent generation of mappings that preserve the basic relationships between concepts. Such preservation is one of the main properties of our previous algorithm [24], and will continue to apply for the new algorithm as well.

Step 1. Computation of tableaux Given the two schemas, the sets of tableaux are generated as follows. For each set type T in a schema, we first create a *primary path* that spells out the navigation path from the root to elements of T. For each intermediate set, there is a variable to denote elements of the intermediate set. To illustrate, recall the

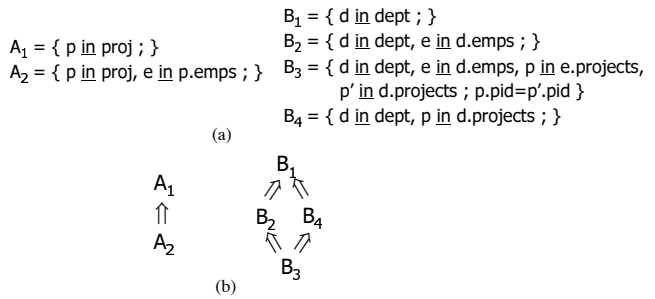


Figure 7: (a) Source and target tableaux (b) Tableaux hierarchies

earlier schemas in Figure 3. In Figure 7(a), A_1 and A_2 are primary paths corresponding to the two set types associated with `proj` and `emps` in the source schema. Note that in A_2, the parent set `proj` is also included, since it is needed in order to refer to an instance of `emps`. Similarly, B_1, B_2, and B_4 are primary paths in the target.

In addition to the structural constraints (parent-child) that are part of the primary paths, the computation of tableaux also takes into account the integrity constraints that may exist in schemas. For our example, the target schema includes the following constraint (similar to a keyref in an XML Schema): every project id of an employee within a department must appear as the id of a project listed under the department. This constraint is explicitly enforced in the tableau B_3 in Figure 7(a). The tableau is constructed by enhancing, via the *chase* [16, 23] with constraints, the primary path B_3' that corresponds to the set type `projects` under `emps`:

$$B_3' = \{\ d \text{ in dept},\ e \text{ in } d.\text{emps},\ p \text{ in } e.\text{projects};\ \}$$

The tableau B_3 encodes, intuitively, that the concept of a project-of-an-employee-of-a-department requires the following concepts to exist: the concept of an employee-of-a-department, the concept of a department, and the concept of a project-of-a-department.

For each schema, the set of its tableaux is obtained by replacing each primary path with the result of its chase (with all the applicable integrity constraints). For our example, only one primary path is changed by the chase (into B_3). The rest remain unchanged (since no constraints are applicable). For each tableau, for mapping purposes, we will consider all the atomic type elements that can be referred to from the variables in the tableau. For example, B_3 includes `dname`, `budget`, `ename`, `salary`, `pid`,[4] `pname`. We say that such elements are *covered* by the tableau. Let us call *generators* the variable bindings that appear in a tableau. Thus, a tableau consists of a sequence of generators and a conjunction of conditions.

Step 2. Generation of basic mappings In the second step of the algorithm, basic mappings are generated by pairing in all possible ways the source and the target tableaux that were generated in the first step. For each pair (A, B) of tableaux, let V be the set of all correspondences for which the source element is covered by A and for which the target element is covered by B. For our example, if we consider the pair (A_1, B_1) then V consists of one correspondence: `dname` to `dname`, identified by d in the earlier Figure 3. If we consider the pair (A_1, B_4) then there is one more correspondence covered: `pname` to `pname` (or p).

Every triple (A, B, V) encodes a possible basic mapping: the for and the associated where clause are given by the generators and the conditions in A, the exists clause is given by the generators in B, and the subsequent where clause includes all the conditions in B along with conditions that encode the correspondences (i.e., for every v in V, there is an equality between the source element of v and

[4]We include only one `pid`, since p.pid is equal to p'.pid.

the target element of v). We may write the basic mapping represented by (A, B, V) as $\forall A \rightarrow \exists B.V$, with the meaning described above. For our example, the basic mapping $\forall A_1 \rightarrow \exists B_4.\{d, p\}$ is precisely the mapping m_1 of Figure 3. Also, the basic mapping $\forall A_2 \rightarrow \exists B_3.\{d, p, e, s\}$ is the mapping m_2 of the same figure.

Among all the possible triples (A, B, V), not all of them generate actual mappings. We generate a basic mapping only if it is not *subsumed* and not *implied* by other basic mappings. This optimization procedure is described in the next subsection.

3.2 Subtableaux and Optimization

The following concept of subtableau plays an important role in reasoning about basic mappings, and in particular in pruning out unlikely mappings during generation (see the following Step 3). The same concept also turns out to be very useful in the subsequent generation of nested mappings.

DEFINITION 3.1. A tableau A is a *subtableau* of a tableau A' (notation $A \leq A'$) if the generators in A form a superset of the generators in A' (possibly after some renaming of variables) and also the conditions in A are a superset of those in A' or they imply them (modulo the renaming of variables). We say that A is a *strict subtableau* of A' ($A < A'$) if $A \leq A'$ and the generators in A form a strict superset of those in A'.

For each schema, the subtableau relationship induces a directed acyclic graph of tableaux, with an edge from A to A' whenever $A \leq A'$. Such a graph can be seen as a hierarchy where the tableaux that are smaller in size are at the top. Intuitively, the tableaux at the top correspond to the more general concepts in the schema, while those at the bottom correspond to the more specific ones. Although the subtableau relationship is reflexive and transitive, most of the time we are concerned with the "direct" subtableau edges. For our example, the two hierarchies (with no transitive edges) are shown in Figure 7(b).

Step 3. Pruning of basic mappings We now complete the algorithm for generation of basic mappings with an additional step that prunes unlikely mappings. This step is especially important because it reduces the number of candidate mappings that the nesting algorithm will have to explore.

A basic mapping $\forall A \rightarrow \exists B.V$ is *subsumed* [11] by a basic mapping $\forall A' \rightarrow \exists B'.V'$ if A and B are respective subtableaux of A' and B', with at least one being strict, and $V = V'$. Note that if A and B are respective subtableaux of A' and B', then necessarily V includes V' (since A and B cover all the atomic elements that are covered by A' and B', and possibly more). The subsumption condition says that we should not consider (A, B, V) since it covers the same set of correspondences that are covered by the smaller (and more general) tableaux A' and B'. For our example, $\forall A_1 \rightarrow \exists B_2.\{d\}$ is subsumed by $\forall A_1 \rightarrow \exists B_1.\{d\}$.

A basic mapping may be logically *implied* by another basic mapping. Testing logical implication of basic mappings can be done using the chase [16, 23], since basic mappings are tuple-generating dependencies (albeit extended over a hierarchical model). Although in our implementation we use the chase (for completeness), often a simpler test suffices: a basic mapping m is implied by a basic mapping m' whenever m is of the form $\forall A \rightarrow \exists B.V$ and m' is of the form $\forall A \rightarrow \exists B'.V'$ and B' is a subtableau of B. Intuitively, all the target components (with their equalities) that are asserted by m are asserted by m' as well (with the same equalities). As an example, $\forall A_1 \rightarrow \exists B_1.\{d\}$ is implied by $\forall A_1 \rightarrow \exists B_4.\{d, p\}$.

We note that subsumption also eliminates some of the implied mappings. In the earlier definition of subsumption, in the particular case when B and B' are the same tableaux then the subsumed mapping is also implied (by the other one). For example, $\forall A_2 \rightarrow \exists B_1.\{d\}$ is subsumed and implied by $\forall A_1 \rightarrow \exists B_1.\{d\}$.

The generation algorithm for basic mappings stops after eliminating all the subsumed and implied mappings.[5] For our example, we are left with only the two basic mappings, m_1 and m_2, from Figure 3.

3.3 When Can We Nest?

We now give a formal definition of the notion of a basic mapping being *nestable* under another basic mapping. This definition follows the intuition given in Section 2.2: we nest m_2 inside m_1 if m_1 is "part" of m_2; morever the nesting is done by factoring out the common part (m_1) and adding the "remainder" of m_2 as a submapping. Based on this definition, we will construct a graph (hierarchy) of basic mappings that will be used by the actual generation algorithm, which is described in Section 3.4.

DEFINITION 3.2. A basic mapping $\forall A_2 \rightarrow \exists B_2.V_2$ is *nestable* inside a basic mapping $\forall A_1 \rightarrow \exists B_1.V_1$ if the following hold:

(1) A_2 and B_2 are strict subtableaux of A_1 and B_1, respectively,

(2) V_2 is a strict superset of V_1, and

(3) there is no correspondence v in $V_2 - V_1$ whose target element is covered by B_1.

For our example, the basic mapping $m_2 = \forall A_2 \rightarrow \exists B_3.\{d, p, e, s\}$ is nestable inside $m_1 = \forall A_1 \rightarrow \exists B_4.\{d, p\}$. In particular, A_2 and B_3 are strict subtableaux of A_1 and B_4; also there are two correspondences in m_2 but not in m_1 (e and s) and their target elements are not covered by B_4.

DEFINITION 3.3. Let $m_2 = \forall A_2 \rightarrow \exists B_2.V_2$ be *nestable* inside $m_1 = \forall A_1 \rightarrow \exists B_1.V_1$. Without loss of generality assume that all variable renamings have been applied so that the generators in A_1 (B_1) are literally a subset of those in A_2 (B_2). The *result of nesting* m_2 inside m_1 is a nested mapping of the form:

$$\forall A_1 \rightarrow \exists B_1.\ [\quad V_1 \wedge \\ \forall (A_2 - A_1) \rightarrow \exists (B_2 - B_1).(V_2 - V_1)\ ]$$

where $\forall (A_2 - A_1) \rightarrow \exists (B_2 - B_1).(V_2 - V_1)$ is a shorthand for a submapping constructed as follows. The for clause contains the generators in A_2 that are not in A_1. The subsequent where clause (if needed) contains all the conditions in A_2 that are not among (and not implied by) the conditions of A_1. The exists clause and subsequent where clause satisfy similar properties with respect to B_2 and B_1. Finally, the last where clause also includes the equalities encoding the correspondences in $V_2 - V_1$.

It can easily be verified that, for our example, the result of nesting m_2 inside m_1 is precisely the nested mapping n. We next explain conditions (1) and (3) in Definition 3.2 (condition (2) is the more obvious one). Assume that m_2 and m_1 are as in Definition 3.2. The condition that A_2 is a strict subtableau of A_1 ensures that the for clause in the submapping that appears in the result of nesting m_2 inside m_1 is non-empty.

Assume now that B_2 is not a strict subtableau of B_1 and it is equal to B_1 (the case when there are additional conditions in B_2 does not affect the discussion). Then, the submapping that appears in the result of nesting of m_2 inside m_1 is a formula of the form: $\forall (A_2 - A_1) \rightarrow (V_2 - V_1)$ (i.e., the equalities on the right are implied by the left-hand side). There is at least one correspondence v in $V_2 - V_1$, and its source element is not covered by A_1 (otherwise it would be in V_1). Hence, in the right-hand side of the above implication, there is at least one equality asserting that a target element covered by B_1 is equal to a source element covered by $A_2 - A_1$. The problem with this is that there are *many* instances of such a source element for *one* instance of the target element (since B_1 is outside the scope of $\forall (A_2 - A_1)$). This constraint would effectively require that all such instances of the source element be equal (and equal to the one instance of the target

[5] Although our original algorithm did not include the elimination of implied mappings, this step can remove many unnecessary formulas.

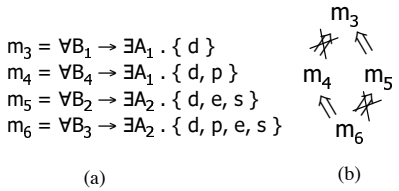


Figure 8: (a) Reverse basic mappings; (b) Nestable relation

element). Such a constraint is unlikely to be desired (even when it is satisfiable). Although condition (3) is a bit more subtle, a careful analysis yields a similar justification.

We illustrate this discussion by considering the reverse of the mapping scenario shown in Figure 3. The schema on the right of that figure is now the source schema, while the schema on the left is the target schema. The correspondences are the same. Also, the tableaux remain the same as in Figure 7, with the difference that B_1, B_2, B_3, B_4 are now source tableaux, and A_1 and A_2 are target tableaux.

There are four basic mappings (not implied and not subsumed) that are generated by the algorithm described in Section 3.1. These mappings are shown in Figure 8(a). We have that m_5 is nestable inside m_3 and m_6 is nestable inside m_4. However, m_4 is not nestable inside m_3 (because the target tableaux is the same). Similarly, m_6 is not nestable inside m_5. If we try to nest m_4 inside m_3, we would obtain the following nested mapping:

n_{34}: for d in `dept` ⇒
exists p' in `proj`
where p'.dname=d.dname ∧
(for p in d.projects ⇒ p'.pname=p.pname)

This constraint says that if there are multiple projects in one `dept` tuple (which is possible according to the source schema) then all these projects are required to have the same `pname` value (which must also equal the `pname` value in the corresponding target `proj` tuple). This puts a constraint on the *source* data that is unlikely to be satisfied. Our algorithm does not generate such mappings.

3.4 Nesting Algorithm

In the next step of the algorithm, we use the *nestable* relation of Definitions 3.2 and 3.3 to create a set of nested mappings. The input to this step is the set of basic mappings that result after Step 3.

Step 4. Generation of nested mappings In this step, the algorithm first constructs a DAG $G = (M, E)$ that represents all possible ways in which basic mappings can be nested under other basic mappings. Here, M is the set of basic mappings generated in Step 3, while E contains edges $m_i \rightarrow m_j$ with the property that m_i is nestable under m_j according to Definition 3.2. To create nested mappings out of G, the *root* mappings of G are identified and a *tree* of mappings is extracted from G for each root. Each such tree of mappings becomes a separate nested mapping.

To understand the shape of G and the issues involved in its construction, we examine the properties of the *nestable* relation of Definition 3.2. Given two basic mappings m_i and m_j, let us write $m_i \Rightarrow m_j$ if m_i is nestable inside m_j. We note that:

(1) The *nestable* relation is not reflexive and not symmetric. In fact, stronger statements hold: (a) for all m_i, $m_i \not\Rightarrow m_i$, and (b) if $m_i \Rightarrow m_j$, then $m_j \not\Rightarrow m_i$. This follows from the strict subtableaux requirement in condition (1) of Definition 3.2.

(2) The *nestable* relation is transitive: if $m_i \Rightarrow m_j$ and $m_j \Rightarrow m_k$ then $m_i \Rightarrow m_k$. This again follows from condition (1) of Definition 3.2 and, further, from conditions (2) and (3).

Because of (1) and (2) above, G is necessarily acyclic. If there is a path $m_i \rightsquigarrow m_j$ in G, then no path $m_j \rightsquigarrow m_i$ exists in G. Condition (2) tells us that a naive algorithm for creating G might add too many edges and hence form unnecessary nestings. Indeed, suppose that $m_i \Rightarrow m_j$ and $m_j \Rightarrow m_k$, which also implies that $m_i \Rightarrow m_k$. Then m_i can be nested under m_j which can be nested under m_k. At the same time, m_i can be nested directly under m_k. However, we prefer the former, deeper, nesting strategy because that interpretation preserves all source data together with its structure.

To illustrate this point, consider the mapping in Figure 1. There, we have that $m_3 \Rightarrow m_2 \Rightarrow m_1$, and also $m_3 \Rightarrow m_1$. We prefer the deepest nesting which results in a nested mapping with the following pattern: first map `dept` tuples, then map the `emps` tuples under the current `dept` tuple, and then map the `dependents` tuples of the current `emps` tuple. The other interpretation, obtained by nesting m_3 directly inside m_1, is not semantically equivalent to the first one. Indeed, this second interpretation maps all `dept` tuples but then, for each `dept` tuple, it maps the join of `emps` and `dependents` tuples (thus, `emps` tuples with no dependents are not mapped). In order not to lose data, we can "fix" this second interpretation by nesting both m_2 and m_3 directly inside m_1 (using the fact that $m_2 \Rightarrow m_1$ and $m_3 \Rightarrow m_1$). This would have the effect of mapping all tuples of `emps`. However, this choice still does not model any correlation between the two submappings m_2 and m_3. Hence, there is no merging of employee tuples and no grouping of dependents within employees. The first interpretation solves the issue by utilizing, intuitively, all the available nesting.

To implement the above nesting strategy, which performs the "deepest" nesting possible, our algorithm for constructing G makes sure not to include any transitively implied edges. More formally, the DAG $G = (M, E)$ of mappings is constructed so that its set of edges satisfies the following:

$$E = \{(m_i \rightarrow m_j) \mid \ m_i \Rightarrow m_j \wedge (\nexists m_k)(m_i \rightsquigarrow m_k \wedge m_k \Rightarrow m_j)\}$$

The creation of G proceeds in two steps. First, for all pairs (m_i, m_j) of mappings in M, we add an edge to G if $m_i \Rightarrow m_j$. Then, for every edge $m_i \rightarrow m_j$ in E, we try finding a *longer* path $m_i \rightsquigarrow m_j$. If such a path exists, we remove $m_i \rightarrow m_j$ from E. This is implemented using a variation of the all-pairs shortest-path algorithm (except that we are looking for the longest path) and its complexity is $O(|M|^3)$.

The next step is to extract *trees* of mappings from G. Each such tree becomes a nested mapping expression. These trees are computed in two simple steps. First, all *root* mappings R in G are identified: $R = \{m_r \mid m_r \in M \ \wedge (\nexists m')(m' \in M \ \wedge (m_r \rightarrow m') \in E)\}$. Then, for each mapping root $m_r \in R$, we do a depth-first traversal of G (following the reverse direction of the edges). Mappings collected during this visit become part of the tree rooted at m_r.

Constructing nested mappings from a tree of mappings raises several issues. First, in Definition 3.3 we explained the meaning of nesting *two* basic mappings, one under the other. But, in a tree, one mapping can have multiple children that each can be nested inside the parent. And also, we must apply the definition recursively. We omit the extensions to Definition 3.3 that are needed to define the result of nesting a tree of mappings as they are straightforward.

The second, more important issue is that, since these trees are extracted from a DAG, it is possible that they share mappings. In other words, a mapping can be nested under more than one mapping.

Consider, for example, a mapping scenario that involves three sets: `employees`, `worksOn`, and `projects`. The `worksOn` set contains references to `employees` and `projects` tuples, capturing an N:M relationship. Assume that m_e is a basic mapping for `employees`, m_p is a basic mapping for `projects`, and m_w is a basic mapping that maps `employees` and `projects` by joining them via `worksOn`. The resulting graph G of mappings contains two mapping trees (i.e., two nested mappings), which both yield valid interpretations: $T_1 = \{m_e \Leftarrow m_w\}$ and $T_2 = \{m_p \Leftarrow m_w\}$. Both trees share

m_w as a leaf. If we arbitrarily use only one tree and ignore the other, then source data can be lost: the nested mapping based on T_1 maps all the employees; however, it only maps projects that are associated with an employee via `worksOn` (the situation is reversed for T_2).

However, the inclusion of the shared subtrees in all their "parent" trees will create nested mappings that lead to redundancy in execution as well as in the generated data. To avoid this, we adopt a simple strategy to keep a shared subtree only in one of the parent trees and prune it from all the other. For our example, we can keep T_1 intact and cut the common subtree from T_2, yielding $T_2' = \{m_p\}$. In general, however, the algorithm should not make a choice of which trees to prune and which to keep intact. This is a semantic and application-dependent decision. The various choices lead to inequivalent mappings that do not lose data but give preference to certain correlations in the data (e.g., group projects by employees as opposed to grouping employees by projects). Furthermore, there can be differences in the performance of the subsequent execution of the data transformation.

Ideally, a human user could suggest which mapping to generate, if exposed to all the possible choices of mappings with shared submappings. We have implemented a strategy that selects one of the pruning choices whenever there is such choice, but in future versions of our prototype we will allow users to explore the space of such choices.

4. QUERY GENERATION

One of the main reasons for creating mappings is to be able to automatically create a query or program that transforms an instance of the source schema into an instance of the target schema. In [24, 11] we described how to generate queries from basic mapping specifications. Here we extend that work to cover nested mappings. Because they start from the more expressive nested mapping specification, the queries that we now generate often perform better, have more functionality in terms of grouping and restructuring, and at the same time are closer to the mapping specification (thus, easier to understand).

We first present in Section 4.1 a general query generation algorithm that works for nested mappings with arbitrary Skolem functions for the set elements (and hence for arbitrary regrouping and restructuring of the source data). In Section 4.2 we present an optimization that simplifies the query and can significantly improve performance in the case of nested mappings with default Skolemization, which are the mappings that we produce with our mapping generation algorithm. In particular, this optimization greatly impacts the scenarios where no complex restructuring of the source is needed (many schema evolution scenarios follow this pattern).

4.1 Two-Phase Query

The general algorithm for query generation produces queries that process source data in two phases. The first-phase query "shreds" source data into flat (or relational) views of the *target* schema. The definition of this query is based on the target schema and on the information encoded in the mappings. The second-phase query is a wrapping query that is independent of the actual mappings and uses the shape of the target schema to nest the data from the flat views in the actual target format.

First-phase query We now describe the construction of the flat views and of the first-phase query. For each target set type for which there is some mapping that asserts some tuple for it, there will be a view, with an associated schema and a query defining it. To illustrate, we will use the earlier schemas of Figure 3 and the earlier nested mapping n. The view schema for our example includes the following definitions:

`dept`(`dname`, `budget`, `empsID`, `projectsID`)
`emps`(`setID`, `ename`, `salary`, `projects1ID`)
`projects1`(`setID`, `pid`)
`projects`(`setID`, `pid`, `pname`)

As it can be seen, the view for each set type includes the atomic type elements that are directly under the set type. Additionally, it also includes setID columns for each of the set types that are directly nested under the given set type. Finally, for each set type that is not top-level (`dept` is the only top-level set type in this example), there is an additional column `setID`. The explanation for this column is the following (we use `emps` to illustrate). While in the target schema there is only one set type `emps`, in an actual instance there may be many sets of employee tuples, nested under the various `dept` tuples. However, the tuples of these nested sets will all be mapped into one single table (`emps`). In order to remember the association between employee tuples and the sets they belong to, we use the `setID` column to record the identity of the set for each employee tuple. This column will then later be used to join with the `empsID` column under the "parent" table `dept`, to construct the correct nesting.

We next describe the queries defining the views and how they are generated. The algorithm starts by Skolemizing each nested mapping and decoupling it into a set of single-headed constraints, each consisting of one implication and one atom in the right-hand side of the implication. For our example, the nested mapping n generates the following four constraints (one for each target atom in n):

$$
\begin{array}{ll}
r_1: & proj(d,p,E_0) \rightarrow dept(d, \texttt{null}, E[d,p,E_0], P[d,p,E_0]) \\
r_2: & proj(d,p,E_0) \rightarrow P[d,p,E_0]\ (X[d,p,E_0], p) \\
r_3: & proj(d,p,E_0) \wedge E_0(e,s) \rightarrow E[d,p,E_0]\ (e, s, P'[d,p,E_0,e,s]) \\
r_4: & proj(d,p,E_0) \wedge E_0(e,s) \rightarrow P'[d,p,E_0,e,s]\ (X[d,p,E_0])
\end{array}
$$

Skolemization replaces every existentially-quantified variable by a Skolem function that depends on all the universally-quantified variables that appear before the existential variable (in the original mapping). For example, the atomic variable $?x$ (along with all of its occurences) is replaced by $X[d,p,E_0]$, where X is a new Skolem function name. [6] Atomic variables that do not play an important role (e.g., not a key or a foreign key) can be replaced by `null` (see $?b$ above). Finally, all existential set variables are replaced by Skolem terms (if they are not already given by the mapping). Each of the above constraints can be seen as an assertion of "facts" that relate tuples and set ids. For example, r_3 above asserts a fact relating the tuple $(e, s, P'[d,p,E_0,e,s])$ and the set id $E[d,p,E_0]$.

Next, the queries defining the contents of the flat views have the role of "storing" the facts asserted by the above constraints into the corresponding flat views. For example, all the facts asserted by r_3 will be stored into `emps`, where the `setID` column is used to store the set ID (as explained earlier). The following is the set of query definitions for our four views:

```
let dept := for p in proj
            return [ dname = p.dname,
                     budget = null,
                     empsID = E[p.dname, p.pname, p.emps],
                     projectsID = P[p.dname, p.pname, p.emps]]
  emps := for p in proj, e in p.emps
          return [ setID = E[p.dname.p.pname,p.emps],
                   ename = e.ename,
                   salary = e.salary,
                   projects1ID = P'[p.dname, p.pname, p.emps,
                                    e.ename, e.salary]]
projects1 := for p in proj, e in p.emps
             return [ setID = P'[p.dname, p.pname, p.emps,
                                 e.ename, e.salary]],
                      pid = X[p.dname, p.pname, p.emps]],
projects := for p in proj
            return [ setID = P[p.dname,p.pname,p.emps],
                     pid = X[p.dname, p.pname, p.emps]],
                     pname = p.pname]
```

We note that if multiple mappings contribute tuples to a target set type, then each such mapping will contribute with a query expression

[6] We really mean here that E_0 is the set id and not the contents. Thus, the Skolem function does not depend on the actual values under E_0.

and the corresponding view is defined by the union of all these query expressions. In the case when the same Skolem function is used from multiple mappings to define the same set instance (as discussed in Section 2.3), then the union of queries defining the view will effectively accumulate all the tuples of this set instance within the view (moreover, all these tuples will have the same set id).

Second-phase query Finally, the previously defined views are used within a query that combines and nests the data according to the shape of the target schema. Notice that the nesting of data on the target is controlled by the Skolem function values computed for the set id columns in the views.

```
(q) dept = for d in dept
           return [
              dname = d.dname,
              budget = d.budget,
              emps = for e in emps
                     where e.setID = d.empsID
                     return [
                        ename = e.ename,
                        salary = e.salary,
                        projects = for p in projects1
                                   where p.setID = e.projects1ID
                                   return [ pid = p.pid ]],
           projects = for p in projects
                      where p.setID = d.projectsID
                      return [ pid = p.pid,
                               pname = p.pname ] ]
```

4.2 Query Inlining for Default Skolemization

The two-phase algorithm is general in the sense that it can work for arbitrary restructuring of the data. However, it does require the data to be flattened before being re-nested in the target format. In cases where the source and target schemas have similar nesting shape and the grouping behavior given by the default Skolem functions is sufficient, the two-phase strategy can be inefficient.

For example, the nested mapping n used in Section 4.1 falls in this category of nested mappings with default Skolemization. Under default Skolemization, all the set ids that are created (by the first-phase query) depend on *entire* source tuples rather than individual pieces of these tuples. To illustrate, the default Skolem function $\mathbf{E}$ for `emps` depends on $p.dname$, $p.pname$ and $p.emps$, which is equivalent to say that E is a function of the source tuple p. Similarly, the Skolem function $\mathbf{P}$ for projects under departments depends on p. Also, the Skolem function $\mathbf{P'}$ for projects under employees depends on $p.dname$, $p.pname$, $p.emps$ *and* $e.ename$ and $e.salary$, which means that it is a function of the source tuples p and e. Under such scenario, we *inline* the views defined by the first-phase query into the places where they occur in the second-phase query. For our example (while taking care of renaming conflicting variable names), we obtain the following rewrite of q:

```
(q') dept = for p in proj
            return [
               dname = p.dname, budget = null,
               emps = for p' in proj, e in p'.emps
                      where E[p] = E[p' ]
                      return [
                         ename = e.ename, salary = e.salary,
                         projects = for p'' in proj, e' in p''.emps
                                    where P'[p',e] = P'[p'',e' ]
                                    return [
                                       pid = X[p''.dname, p''.pname, p''.emps] ] ],
               projects = for p' in proj
                          where P[p] = P[p' ]
                          return [ pid = X[p''.dname, p'.pname, p'.emps],
                                   pname = p'.pname ] ]
```

Since the Skolem functions are *one-to-one* id generators, we can now replace the equalities of the function terms with the equalities of the arguments. Thus we can replace $\mathbf{E}[p] = \mathbf{E}[p']$ with $p = p'$. We can also replace $\mathbf{P'}[p', e] = \mathbf{P'}[p'', e']$ with the conjunction of $p' = p''$ and $e = e'$, and also $\mathbf{P}[p] = \mathbf{P}[p']$ with $p = p'$. Hence, we obtain a rewriting where some of the inner loops are unnecessary. The boxes in q' above highlight the "redundant" parts. We can then rewrite q' by removing the declaration of p' and the self-join condition $p = p'$. If we do this at all levels where setID equalities are used, then all the highlighted parts of the query can be redacted. (In some cases, the loops are completely replaced by singleton set expressions; this happens for both `projects` sets in our example.) The final query is shown below. It tightly follows the expressions (and optimizations) encoded in the nested mapping n.

```
(q") dept = for p in proj
            return [
               dname = p.dname, budget = null,
               emps = for e in p.emps
                      return [
                         ename = e.ename, salary = e.salary,
                         projects = { [ pid = X[p.dname, p.pname, p.emps] ] } ],
               projects = { [
                  pid = X[p.dname, p.pname, p.emps],
                  pname = p.pname ] } ]
```

5. EXPERIMENTS

We conducted a number of experiments to understand the performance of (a) the nested mapping queries described in Section 4 and (b) the nested mapping creation algorithm of Section 3. Our nested mapping prototype is implemented in Java, on top of Clio [11]. The experiments were performed on a PC-compatible machine, with a single 2.0GHz P4 CPU and 1GB RAM, running Windows XP (SP1) and JRE 1.4.2. Each experiment was repeated three times, and the average of the three trials is reported.

5.1 Query Evaluation

We first compare the performance of queries generated using nested mappings with queries generated from basic mappings. We focus on a schema evolution scenario where nested mappings with default Skolemization suffice to express the desired transformation and inlining is applied to optimize the nested mapping query (as described in Section 4.2). We created a nested schema *authorDB*, based on the DBLP[7] structure, but with four levels of nesting. The first level contais an *author* set. Each *author* tuple has an attribute *name* and a nested set of *confJournal* tuples. Each *confJournal* tuple has an attribute *name* and a set of *year* tuples. Each *year* tuple contains a *yr* attribute and a set of *pub* elements, each with five attributes: *pubId*, *title*, *pages*, *cdrom*, *url*.

We ran the basic and nested mapping algorithms on four different settings to create four pairs of mappings (one basic and one nested). We used *authorDB* as the source and target schema and added different sets of correspondences to create the four different settings. In the first, m_1, we only mapped the top-level *author* set (this means we used only one correspondence between the *name* attributes of *author*). In the second mapping, we mapped the first and the second level of *authorDB* (i.e, *author* and *confJournal*). Since we mapped levels 1 and 2, we will refer to this mapping as m_{12}. We proceeded in the same fashion adding correspondences for the third and fourth levels *authorDB*, creating mappings m_{123} and m_{1234}, respectively.

For each mapping, we created two XQuery scripts: one generated using the basic mappings (using the original Clio query generation algorithm [24, 11]), and another generated from the nested mappings (as described in Sections 4.1 and 4.2). Figure 9 compares the generated queries for m_{12}. To simplify the experiment, we considered input instances where each *author* has at least one *confJournal* element under it, and similarly, each *confJournal* contains at least one

[7] http://www.informatik.uni-trier.de/ ley/db/

```
let $doc0 := fn:doc("instance.xml") return
<authorDB>
 { for $x0 in $doc0/authorDB/author,
       $x1 in $x0/confJournal
   return
    <author>
     <name> { $x0/name/text() } </name>
     { for $x0L1 in $doc0/authorDB/author,
           $x1L1 in $x0L1/confJournal
       where $x0/name/text()=$x0L1/name/text()
       return
        <confJournal>
         <name> { $x1L1/name/text() } </name>
        </confJournal> }
    </author> }
</authorDB>
```

```
let $doc0 := fn:doc("instance.xml") return
<authorDB>
 { for $x0 in $doc0/authorDB/author
   return
    <author>
     <name> { $x0/name/text() } </name>
     { for $x1 in $x0/confJournal
       return
        <confJournal>
         <name>{ $x1/name/text() }</name>
        </confJournal> }
    </author> }
</authorDB>
```

Figure 9: Basic (left) and nested (right) query for m_{12}.

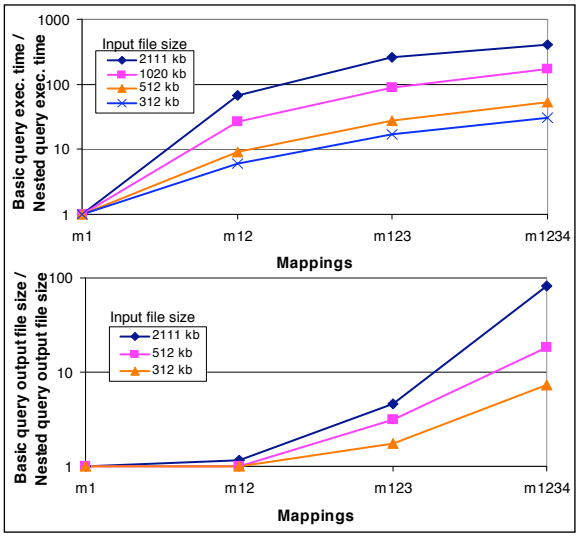


Figure 10: Execution time (upper) and output file size factors.

year subelement and each *year* contains at least one *pub* subelement. As a consequence, only one basic mapping is enough to map all the source data. Otherwise we would have to consider additional basic mappings (e.g., map *author* elements independently of the existence of *confJournal* subelements). This would only make the basic mapping query become more complex and have worse performance. On the other hand, even in the favorable case where one basic mapping is enough, we show that the nested mapping query is still much better.

We ran the queries using the Saxon XQuery processor[8] with increasingly larger input files. Figure 10 shows that the nested mapping queries consistently outperformed the basic mapping queries, both in time and in the size of the output instance generated.[9] The upper part of Figure 10 plots the execution speed-up for the nested mapping queries (i.e., the ratio of the execution time for the basic mapping query over the execution time for the query generated with the nested mapping). The lower chart shows the ratio of the output file size for the basic mapping over the output file size for the nested mapping. Both charts use a logarithmic scale in the y-axis.

A cursory inspection of the queries in Figure 9 reveals the reason for the better execution time of the nested mapping queries. Our basic mapping query generation strategy repeats the source tableau expression for each target set type. In the case of m_{12}, the basic query in-

[8]saxon.sourceforge.net

[9]Larger output files for the same mapping indicate more duplicate tuples in the result.

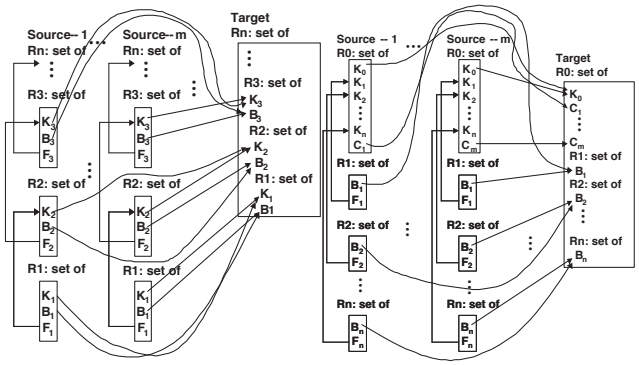


Figure 11: The chain (left) and authority (right) scenarios.

terates over every source *author* and *confJournal* once to create target *author* elements (variables $x0$ and $x1$ in the query). A second loop is used to compute the nested *confJournal* elements (variables $x0L1$ and $x1L1$). Further, since we only want to nest the *confJournal* elements for the current *author* tuple, the second loop is correlated to the outer one (the `where` clause in the query). That is, this query requires two passes over the input data plus a correlated nested subquery to correctly nest data. In contrast, the nested mapping query only does one pass over the source *author* and *confJournal* data and does not need any correlation condition since it takes advantage of existing nesting of the source data.

The basic mapping query strategy can also create a large number of duplicates in the output instance. To illustrate this problem, we create a mapping m_{14} that maps the *author* and *pub* levels of the schema. We ran the queries for m_{14} and m_{1234} using an input instance that contains 4173 *author* elements and a total of 6468 *pub* elements nested within those authors. The count of resulting *author* and *pub* elements in the output instance is shown in this table:

Mapping	B *author*	B *pub*	NM *author*	NM *pub*
m_{14}	6468	18826	4173	6468
m_{1234}	6468	157254	4173	6468

The nested mapping queries do not create duplicates for any of the two mappings and produce a copy of the input instance (which is the expected output instance in all these mappings). The basic mapping queries, on the other hand, create 2295 duplicate *author* elements. Intuitively, a duplicate is created whenever an author has more than one publication. Each author duplicate then carries the same set of duplicate publications causing an explosion of duplicate *pub* elements. The nested mapping query we *automatically* generate does not suffer from this common problem.

5.2 Algorithm Evaluation

We now study the performance and scalability of the nested mapping creation algorithm. We designed two synthetic scenarios (depicted in Figure 11), **chain** and **authority** [27]. The chain scenario simulates mappings between multiple inter-linked relational tables and an XML target with large number of nesting levels. The authority scenario simulates mappings between multiple relational tables referencing a central table and a shallow XML target with a large branching factor (large number of child tables). For each scenario, we used a schema generator to create schema definitions with variable degrees of complexity (e.g., number of elements, referential constraints, number of nesting levels). In addition, we also replicated each generated source schema a number of times in order to simulate the cases of multiple data sources mapping into one target.

For the chain scenario we increased the number of different sources (m) and the number of inter-linked relational tables ($depth$) ($1 \leq$

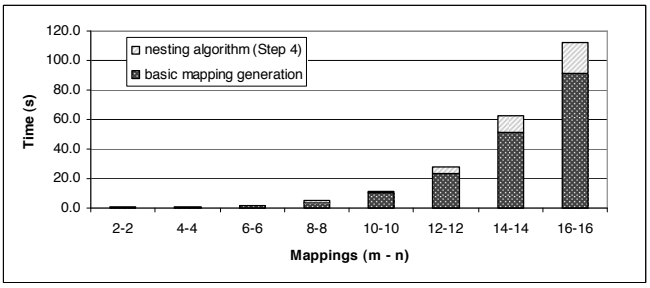


Figure 12: Algorithm's performance for the authority scenario.

$m \leq 20$ and $1 \leq depth \leq 3$). In the worst case, our prototype took 0.2 seconds to compute the nested mapping. For the authority scenario, we simultaneously increased the number of sources (m) and the branching factor (n) (the number of child tables) such that $m=n$ for each trial. Figure 12 shows the results for authority. For schemas of small to medium size (when m and n are less than 12), the nesting algorithm (Step 4 described in Section 3.4) finishes in a few seconds after the computation of the basic mappings (Steps 1, 2 and 3). But the time degrades exponentially as the mapping complexity increases. Note, however, that in the largest case we tried ($m = n = 20$), the nesting algorithm (Step 4) took only about 20 seconds.

Finally, we evaluated the algorithm performance with a mapping that uses the Mondial schema [17], a database of geographical data. Mondial has a relational representation with 28 relations and a maximum branching factor of 9. Its XML Schema counterpart has a maximum depth of 5 and a maximum branching factor of 9. We mapped from the relational into the XML representation and created 26 basic mappings in 1.2 seconds. The nesting algorithm then extracted 10 nested mappings in 2.8 seconds.

6. CONCLUSION

We have introduced a new, structured mapping formalism called *nested mappings* that provides a natural way to express correlations between schema mappings. We demonstrated benefits of this formalism including increased specification accuracy, and the ability to specify (and customize) grouping semantics declaratively. We provided an algorithm to generate nested mappings from standard schema matchings. We showed how to compile these mappings into transformation queries that can be much more efficient than their counterparts obtained from the earlier basic mappings. The new transformation queries also generate much cleaner data. Certainly nested mappings have important applications in schema evolution where the mapping must be able to ensure that the grouping of much of the data is not changed. Indeed our work here was largely inspired by the inability of existing mapping formalisms to faithfully represent the "identity mapping" for many schemas. We are currently evaluating the use of nested mappings in other tasks including (virtual) data integration over large schemas and large collections of schemas.

7. REFERENCES

[1] M. Arenas and L. Libkin. XML Data Exchange: Consistency and Query Answering. In *PODS*, pages 13–24, 2005.

[2] M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In *SIGMOD*, pages 277–288, 2003.

[3] P. Bernstein, S. Melnik, and P. Mork. Interactive Schema Translation with Instance-Level Mappings. In *VLDB*, pages 1283–1286, 2005.

[4] P. Bohannon, W. Fan, M. Flaster, and P. P. S. Narayan. Information preserving xml schema embedding. In *VLDB*, pages 85–96, 2005.

[5] A. Bonifati, E. Q. Chang, T. Ho, V. S. Lakshmanan, and R. Pottinger. HePToX: Marrying XML and Heterogeneity in Your P2P Databases. In *VLDB*, pages 1267–1270, 2005.

[6] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *WWW8*, pages 77–91, 1999.

[7] R. Fagin. Inverting Schema Mappings. In *PODS*, 2006.

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

[9] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[10] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational Plans For Data Integration. In *AAAI/IAAI*, pages 67–73, 1999.

[11] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *SIGMOD*, pages 805–810, 2005.

[12] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza Peer Data Management System. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.

[13] R. Hull and M. Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In *VLDB*, pages 455–468, 1990.

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

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

[16] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM TODS*, 4(4):455–469, 1979.

[17] W. May. Information Extraction and Integration with FLORID: The MONDIAL Case Study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999.

[18] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Applying Model Management to Executable Mappings. In *SIGMOD*, pages 167–178, 2005.

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

[20] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *VLDB*, pages 122–133, 1998.

[21] A. Nash, P. A. Bernstein, and S. Melnik. Composition of Mappings Given by Embedded Dependencies. In *PODS*, pages 172–183, 2005.

[22] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *VLDB*, pages 413–424, 1996.

[23] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *ICDT*, pages 39–57, 1999.

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

[25] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, pages 249–260, 2000.

[26] Y. Velegrakis, R. J. Miller, and L. Popa. Preserving Mapping Consistency under Schema Changes. *VLDB Journal*, 13(3):274–293, 2004.

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

[28] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. In *VLDB*, pages 1006–1017, 2005.

APPENDIX

A. DEFINITION OF NESTED MAPPINGS

We can now give a precise definition of nested mapping in terms of the query-like notation. An *expression* is defined by the grammar $e ::= S|x|e.l$, where S is a schema root, x is a variable, l is a label, and $e.l$ is record projection. A *nested mapping* has the following form

$$
\begin{aligned}
M ::= & \underline{\textsf{for}}\ x_1\ \underline{\textsf{in}}\ g_1, \ldots, x_n\ \underline{\textsf{in}}\ g_n \Rightarrow \\
& \underline{\textsf{where}}\ C_1 \\
& \underline{\textsf{exists}}\ y_1\ \underline{\textsf{in}}\ g'_1, \ldots, y_n\ \underline{\textsf{in}}\ g'_n \\
& \underline{\textsf{where}}\ (C_2 \wedge M_1 \wedge \cdots \wedge M_n)
\end{aligned}
$$

We say that $M_1, \ldots, M_n$ are *submappings* of M. Each submapping is itself a nested mapping. We will say that M is an ancestor of $M_1, \ldots, M_n$ and (recursively) of the submappings of $M_1, \ldots, M_n$.

Each $x_i\ \underline{\textsf{in}}\ g_i$ is a *source generator*. In a source generator, the head of expression g_i must be a source schema root; or it must be a variable defined in a source generator of M (in which case it must be some x_j with $j < i$) or in a source generator of an ancestor of M. Each $y_i\ \underline{\textsf{in}}\ g'_i$ is a *target generator*. In a target generator, the head of expression g'_i must be a target schema root; or it must be a variable defined in a target generator of M (in which case it must be some y_j with $j < i$), or in a target generator of an ancestor of M.

A *source expression* is an expression over a variable defined in a source generator of M or in a source generator of an ancestor of M. A *target expression* is an expression over a variable defined in a target generator of M or in a target generator of an ancestor of M. The expression C_1 consists of a conjunction of equalities between source expressions of atomic type. The expression C_2 has three kinds of equalities. First, it has target conditions: equalities between target expressions of atomic type. Second, it has source-to-target conditions: equalities between source and target expressions of atomic type. Finally, it has grouping conditions: equalities of the form $e = F[e_1, ..., e_m]$ where F is a Skolem function, e is a target expression of set type, and $e_1, ..., e_m$ are source expressions.