

Debugging Schema Mappings with Routes*

Laura Chiticariu
UC Santa Cruz
laura@cs.ucsc.edu

Wang-Chiew Tan
UC Santa Cruz
wctan@cs.ucsc.edu

ABSTRACT

A schema mapping is a high-level declarative specification of the relationship between two schemas; it specifies how data structured under one schema, called the source schema, is to be converted into data structured under a possibly different schema, called the target schema. Schema mappings are fundamental components for both data exchange and data integration. To date, a language for specifying (or programming) schema mappings exists. However, developmental support for programming schema mappings is still lacking. In particular, a tool for *debugging* schema mappings has not yet been developed. In this paper, we propose to build a debugger for understanding and exploring schema mappings. We present a primary feature of our debugger, called *routes*, that describes the relationship between source and target data with the schema mapping. We present two algorithms for computing all routes or one route for selected target data. Both algorithms execute in polynomial time in the size of the input. In computing all routes, our algorithm produces a concise representation that factors common steps in the routes. Furthermore, every *minimal* route for the selected data can, essentially, be found in this representation. Our second algorithm is able to produce one route fast, if there is one, and alternative routes as needed. We demonstrate the feasibility of our route algorithms through a set of experimental results on both synthetic and real datasets.

1. INTRODUCTION

A schema mapping is a high-level declarative specification of the relationship between two schemas; it specifies how data structured under one schema, called the source schema, is to be converted into data structured under a possibly different schema, called the target schema. Schema mappings are fundamental components for both data exchange and data integration [12, 13]. A widely used formalism for specifying relational-to-relational schema mappings is that of *tuple generating dependencies (tgds)* and *equality generating dependencies (egds)*. (In the terminology of data integration, tgds are equivalent to *global-and-local-as-view* assertions.) Using

* Supported in part by NSF CAREER Award IIS-0347065 and NSF grant IIS-0430994.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

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

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

a language that is based on tgds and egds for specifying (or programming) schema mappings has several advantages over “lower-level” languages, such as XSLT scripts or Java programs [14, 19], in that it is declarative and it has been widely used in the formal study of the semantics of data exchange and data integration. Indeed, the use of a higher-level declarative language for programming schema mappings is similar to the goal of model management [4, 15]. One of the goals in model management is to reduce programming effort by allowing a user to manipulate higher-level abstractions, called models and mappings between models. In this case, models and mappings between models are schemas and mappings between schemas. A recent example of a data exchange system that allows a user to program a schema mapping using a declarative language based on tgds and egds is Clío [11]. However, developmental support for programming schema mappings in this language is still lacking. In particular, to the best of our knowledge, a tool for *debugging* schema mappings has not yet been developed. It is for the same motivation as developing a debugger for a programming language that we wish to develop a debugger for the language of schema mappings.

In this paper, we present a primary feature of our debugger, called *routes*, that allows a user to explore and understand a schema mapping. Routes describe the relationships between source and target data with the schema mapping. A user is able to select target (or source) data and our algorithms are able to compute all routes or one route for the selected data. Our algorithms are based on the formalism of tgds and egds for specifying schema mappings and we have implemented these algorithms on top of Clío, with Clío’s language for programming schema mappings. We emphasize that even though our implementation is built around Clío’s language for schema mappings, it is not specific to the execution engine of Clío. In fact, our implementation requires *no* changes to the underlying execution engine of Clío. Hence, we believe that the algorithms we have developed in this paper can be easily adapted for other data exchange systems based on a similar formalism. Our debugger can also be used to understand the specification of a data integration system: In this case, we materialize (test) data under the target schema (often called the *global schema* in the terminology of data integration) for the purpose of debugging the schema mapping.

It is also worth mentioning that in Clío, schema mappings are often not programmed directly by the user but, rather, they are generated from the result of matching the source and target schemas (i.e., schema matching). However, it is often the case that the generated schema mapping needs to be further refined before it accurately reflects the user’s intention. Hence, even though schema mappings are not usually programmed directly in Clío, there is still a need for a debugger that would allow the user to understand the generated schema mapping.

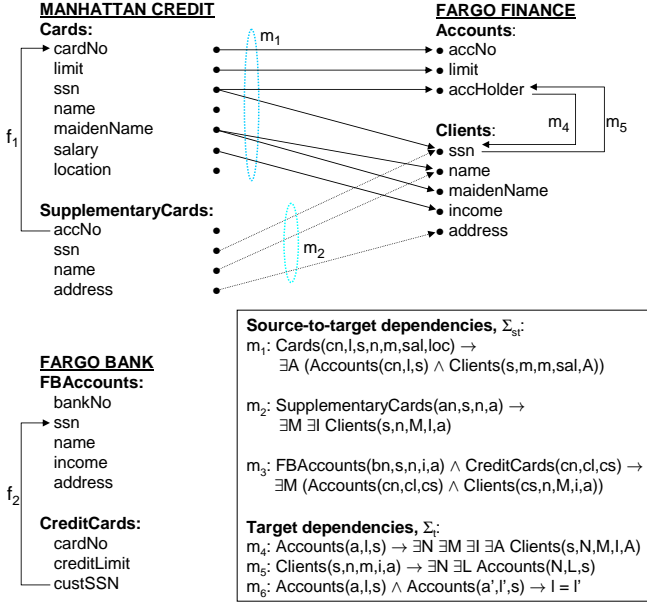


Figure 1: A relational-to-relational schema mapping.

Contributions We propose to build a debugger for understanding and exploring schema mappings. A description of some of the features in this prototype debugger can be found in [2]. In this paper, we make the following contributions.

- We propose the concept of a *route* that we use to drive a user’s understanding of a schema mapping. A route describes the relationship between source and target data with the schema mapping. Our concept of a route is not tied to any procedural semantics associated with the transformation of data from the source to the target according to the schema mapping.
- We describe an algorithm for constructing a concise representation of all routes for the selected target data with the following properties: (1) It runs in polynomial time in the size of the input. (2) Our representation factors common steps in the routes. (3) Every *minimal* route can, essentially, be found in this polynomial-size representation, even though there may be exponentially many minimal routes.
- We also describe an algorithm that computes one route fast for selected target data if there is one, and produces another route as needed. This algorithm executes in polynomial time in the size of the input.
- The route algorithms we present can be easily adapted to work for selected source (not target) data as well. Our implementation handles both relational/XML to relational/XML data exchange although we shall only describe our algorithms for the relational-to-relational case in this paper.
- Our experiments report on the feasibility of our algorithms. In particular, they show that computing one route can execute much faster than computing all routes. Hence, even though we can compute all routes, the ability to compute one route fast and exploit the “debugging-time” of the user to generate alternative routes, as needed, is valuable.

2. BACKGROUND

We introduce various concepts from the data exchange framework [8] that we will use in this paper and we briefly describe the

SOURCE INSTANCE I									
Cards									
	cardNo	limit	ssn	name	maidenName	salary	location		
s_1 :	6689	15K	434	J. Long	Smith	50K	Seattle		
SupplementaryCards									
	accNo	ssn	name	address					
s_2 :	6689	234	A. Long	California					
FBAccounts									
	bankNo	ssn	name	income	address				
s_3 :	1001	234	A. Long	30K	California				
s_4 :	4341	153	C. Don	900K	New York				
CreditCards									
	cardNo	creditLimit	custSSN						
s_5 :	2252	2K	234						
s_6 :	5539	40K	153						
SOLUTION J									
Accounts									
	accNo	limit	accHolder						
t_1 :	6689	15K	434						
t_2 :	N_1	2K	234						
t_3 :	2252	2K	234						
t_4 :	5539	40K	153						
Clients									
	ssn	name	maidenName	income	address				
t_5 :	434	Smith	Smith	50K	A_1				
t_6 :	234	A. Long	M_1	I_1	California				
t_7 :	153	A. Long	M_2	30K	California				
t_8 :	234	A. Long	M_3	30K	California				
t_9 :	153	C. Don	M_4	900K	New York				
t_{10} :	234	C. Don	M_5	900K	New York				

Figure 2: A source instance I and a solution J for I.

data exchange system Clio [11, 18].

Schema Mapping The specification of a data exchange is given by a *schema mapping* $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ [12], where \mathbf{S} is a source schema, \mathbf{T} is a target schema, Σ_{st} is a set of *source-to-target dependencies* (*s-t dependencies*) and Σ_t is a set of *target dependencies*. In the relational-to-relational data exchange framework [8], *s-t dependencies* is a finite set of *s-t tuple generating dependencies* (*tgds*) and the set of target dependencies is the union of a finite set of *target tgds* with a finite set of *target equality generating dependencies* (*egds*). A *s-t tgd* has the form $\forall \mathbf{x} \phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})$, where $\phi(\mathbf{x})$ is a conjunction of atomic formulas over \mathbf{S} and $\psi(\mathbf{x}, \mathbf{y})$ is a conjunction of atomic formulas over \mathbf{T} . A *target tgd* has a similar form, except that $\phi(\mathbf{x})$ is a conjunction of atomic formulas over \mathbf{T} . A *target egd* is of the form $\forall \mathbf{x} \phi(\mathbf{x}) \rightarrow x_1 = x_2$, where $\phi(\mathbf{x})$ is a conjunction of atomic formulas over \mathbf{T} , and x_1 and x_2 are variables that occur in \mathbf{x} .

Figure 1 shows a schema mapping \mathcal{M} , where \mathbf{S} consists of the relation schemas **Cards**, **SupplementaryCards**, **FBAccounts**, and **CreditCards** in Manhattan Credit and Fargo Bank. The target schema consists of the relation schemas **Accounts** and **Clients** in Fargo Finance. The set Σ_{st} consists of three s-t tgds, illustrated as m_1 , m_2 and m_3 (shown in the box). Only m_3 is not depicted as arrows in the figure. The set Σ_t consists of two target tgds m_4 and m_5 and a target egd m_6 (also shown in the box). For conciseness, we have omitted the universal quantifiers of the dependencies. In this example scenario, the goal is to migrate every cardholder and supplementary card holder of Manhattan Credit as a client of Fargo Finance. Also, every credit card holder of Fargo Bank is a client of Fargo Finance. Intuitively, m_1 and m_2 migrate data from Manhattan Credit to Fargo Finance, while m_3 migrates data from Fargo Bank to Fargo Finance. For example, m_1 projects every tuple (or fact) from the **Cards** relation into two tuples, in **Accounts** and **Clients** relations respectively. The target tgds m_4 and m_5 state that an account with **accHolder** value s exists in **Accounts** if and only if a client with **ssn** value s exists in **Clients**. Furthermore, the target egd m_6 states that there can only be one credit limit for an account holder. Although not part of the schema mapping, there is a constraint, depicted as f_1 in Figure 1, that states that for every supplementary card, there must be a sponsoring card in the **Cards** relation whose **cardNo** equals **accNo**. There is also a constraint f_2 from **CreditCards** to **FBAccounts** that states that every credit

card holder of Fargo Bank must have a bank account.

Solutions and homomorphisms Figure 2 illustrates a source instance I , as well as a solution J for I under \mathcal{M} of Figure 1. We say that J is a solution for I under \mathcal{M} if J is a finite target instance such that (I, J) satisfies $\Sigma_{st} \cup \Sigma_t$. In other words, (I, J) satisfies the schema mapping \mathcal{M} . The solution J may contain *labeled nulls*. In Figure 2, $N_1, M_1, \dots, M_5, I_1$, and A_1 are labeled nulls. Distinct labeled nulls are used to denote possibly different unknown values in the target instance.

Let K and K' be two instances. We say that h is a *homomorphism from K to K'* , denoted as $h : K \rightarrow K'$, if h maps the constants and labeled nulls of K to the constants and labeled nulls of K' such that $h(c) = c$ for every constant c and for every tuple (or fact) $R(t)$ of K , we have that $R(h(t))$ is a tuple of K' . We also say that h is a *homomorphism from a formula $\phi(\mathbf{x})$ to an instance K* , denoted as $h : \phi(\mathbf{x}) \rightarrow K$, if h maps the variables of \mathbf{x} to constants or labeled nulls in K such that for every relational atom $R(\mathbf{y})$ that occurs in $\phi(\mathbf{x})$, we have that $R(h(\mathbf{y}))$ is a tuple in K .

In general, there are many possible solutions for I under \mathcal{M} . A *universal solution J* for I under \mathcal{M} has the property that it is a solution and it is the most general in that there is a homomorphism from J to every solution for I under \mathcal{M} . It was shown in [8] that the result of *chasing* I with $\Sigma_{st} \cup \Sigma_t$ is a universal solution.

Clio Clio is a prototype data exchange system developed at IBM Almaden Research Center [11, 18]. The schema mapping language of Clio is a nested relational extension of tgds and egds that also handles XML data. In Clio, a user gets to make associations between source and target schema elements by specifying value correspondences, or Clio may suggest possible value correspondences. Value correspondences are illustrated as arrows as in Figure 1. Clio then interprets these value correspondences into s-t (nested) tgds. From these s-t tgds, executables such as XSLT scripts are generated. Given a source instance I , a solution J is created by applying the generated script on I . We note that Clio does not compute a target instance based on the chase procedure [8] that has been defined for data exchange. Also, the current Clio implementation does not handle target egds, although the general framework does not impose this restriction.

2.1 Example debugging scenarios

Next we illustrate some usage scenarios with our debugger. We assume that Alice, a banking specialist, is interested to debug \mathcal{M} of Figure 1. We expect that in most cases, Alice would debug \mathcal{M} by providing her own (small) test data for the source. In this case, she uses the source instance I and solution J shown in Figure 2.

Scenario 1: Incomplete and incorrect associations between source and target schema elements

When Alice browses through J , she discovers that the `address` value of the tuple t_5 in `Clients` contains a null A_1 . Knowing that neither Fargo Bank nor Manhattan Credit would allow any of its customers to open an account without providing an address, she probes t_5 . Our debugger shows a route from the source that is a witness for t_5 in the target, depicted as $s_1 \xrightarrow{m_1, h} t_1, t_5$. The route consists of the source tuple s_1 in `Cards`, the tgd m_1 , as well as an assignment h of variables of m_1 : $\{cn \mapsto 6689, l \mapsto 15K, s \mapsto 434, n \mapsto \text{J. Long}, m \mapsto \text{Smith}, sal \mapsto 50K, loc \mapsto \text{Seattle}, A \mapsto A_1\}$. Under this assignment, the right-hand-side (RHS) of m_1 is t_1 and t_5 . Hence, m_1 asserts the presence of t_5 with s_1 and h . With this route, Alice discovers that the address of J. Long (i.e., the value “Seattle”) was not copied over by the tgd. Indeed, Figure 1 shows that there is no value correspondence between any schema element of `Cards` and `address` of `Clients`. Suppose Alice also noticed that in t_5 , the `name` value is the same as its `maidenName` value (i.e., Smith). With the same

route $s_1 \xrightarrow{m_1, h} t_1, t_5$, Alice discovers that `maidenName` of `Cards` has been incorrectly mapped to `name` of `Clients`. She therefore corrects m_1 to the following tgd m'_1 which (1) adds the missing value correspondence between `location` of `Cards` and `address` of `Clients` and (2) retrieves the `name` of `Clients` from the `name` of `Cards`:

$$m'_1: \text{Cards}(cn, l, s, n, m, sal, loc) \rightarrow \\ \text{Accounts}(cn, l, s) \wedge \text{Clients}(s, \boxed{n}, m, sal, \boxed{loc})$$

In this scenario, our debugger has helped Alice discover an incomplete, as well as an incorrect association between source and target schema elements. Ideally, we would also like to be able to simultaneously demonstrate how the modification of m_1 to m'_1 affects tuples in J . This is one of our future work.

Scenario 2: Incomplete associations between source schema elements

When browsing through J , Alice discovered that A. Long (tuple t_7) who has an income of 30K has a credit limit of 40K (tuple t_4). Knowing that it is very unlikely for an account holder to have a credit limit that is higher than her income, Alice probes t_4 . Our debugger explains that t_4 was created due to the s_4 tuple in `FBAccounts` and the s_6 tuple of `CreditCards` through m_3 . Suppose Alice could not find anything peculiar with this explanation. She now requests to view all routes for t_4 . Our debugger reports only one other route that uses the first tuple in `FBAccounts` (s_3) and the tuple s_6 through m_3 . Since the `ssn` values of these two source tuples are different, Alice realizes that m_3 has missed the join condition on `ssn` in the source relations. She corrects the s-t tgd to:

$$m'_3: \text{FBAccounts}(bn, \boxed{cs}, n, i, a) \wedge \text{CreditCards}(cn, cl, cs) \rightarrow \\ \exists M(\text{Accounts}(cn, cl, cs) \wedge \text{Clients}(cs, n, M, i, a))$$

Alice may also decide to enforce `ssn` as a key of the relation `Clients`, which can be expressed as egds. Here, our debugger has helped Alice discover a missing join condition, as well as realize an additional dependency that may need to be added to the target.

Scenario 3: Incomplete associations between relations

As Alice browses through the target instance further, she sees that the `accNo` of the account holder 234 is unspecified (N_1 of tuple t_2). As it is not likely that there is no account number for an account holder, Alice probes N_1 of t_2 . Our debugger shows that t_2 was created through the target tgd m_5 with the tuple t_6 . (Note that with this explanation, the existentially-quantified variable L of m_5 is assumed to map to the value 2K of t_2 .) Furthermore, our debugger shows that t_6 was created through the tgd m_2 with the source tuple s_2 . With this information, Alice discovers that m_2 is in fact missing an association with the source relation `Cards`. Indeed, every supplementary card holder must have a sponsoring card holder in `Cards` and they share the same credit limit. So Alice corrects m_2 by adding the association between `SupplementaryCards` and `Cards`, as indicated by the constraint f_1 . Furthermore, the target now includes an `Accounts` relation that is used to hold the account number and `ssn` of the supplementary card holder, as well as the credit limit of the sponsoring card holder. The new tgd m'_2 , with the changes highlighted, is now specified as follows:

$$m'_2: \boxed{\text{Cards}(cn, l, s_1, n_1, m, sal, loc)} \wedge \\ \boxed{\text{SupplementaryCards}(cn, s_2, n_2, a)} \rightarrow \\ \exists M \exists I (\text{Clients}(s_2, n_2, M, I, a) \wedge \boxed{\text{Accounts}(cn, l, s_2)})$$

In this scenario, our debugger has helped Alice discover an incomplete tgd that misses out on some associations between relations in the source schema, as well as between relations in the target schema. Alice may also choose to remove the tgd m_2 completely

Algorithm ComputeAllRoutes $\mathcal{M}(I, J, J_s)$

Input: A source instance I , a solution J for I under \mathcal{M} and a set of tuples $J_s \subseteq J$.

Output: A route forest for J_s .

Global data structures:

- A set of ACTIVETUPLES that contains tuples for which the algorithm has attempted to find all routes. Initially, this set is empty.

FindAllRoutes(J_s)

For every tuple t in J_s

If t is not in ACTIVETUPLES, then

1. Add t to ACTIVETUPLES.
2. For every s-t tgd σ and assignment h such that h is a possible assignment returned by findHom(I, J, t, σ)
 - (a) Add (σ, h) as a branch under t .
3. For every target tgd σ and assignment h such that h is a possible assignment returned by findHom(I, J, t, σ)
 - (a) Add (σ, h) as a branch under t .
 - (b) FindAllRoutes(LHS($h(\sigma)$)).

Return the constructed route forest for J_s .

Figure 3: An algorithm for computing all routes.

if she thinks it is incorrect, i.e., only primary card holders of Manhattan Credit are automatically customers of Fargo Finance.

Several remarks are in order now. First, debugging a schema mapping is not solely a matter of identifying the target schema elements that are left unmapped or that are mapped incorrectly from the source. Indeed, as scenarios 2 and 3 illustrate, the problems may also be due to missing associations between source schema elements, or missing relations. Second, observe that routes are always computed in its entirety even though only part of a route may demonstrate problems with the schema mapping (see for example, Scenario 3). Third, as illustrated in Scenario 2, observe that there are situations in which a computed route may not reveal any problems with the schema mapping. In scenario 2, Alice needs the knowledge of the second route for t_7 to discover the problem in the s-t tgd m_3 . Certainly, one may argue that if the second route for t_7 would have been computed before the first one, Alice would have been able to debug m_3 without the knowledge of additional routes for t_7 . It is conceivable, however, that t_7 is a tuple containing sensitive information and in this situation, the knowledge of *all* routes for t_7 would be crucial for the purpose of identifying tgds that export sensitive information. It is also worth mentioning that in our debugging scenarios, we have only illustrated the use of routes for understanding the schema mapping through anomalous tuples. We believe that routes for correct tuples are also useful for understanding the schema mapping in general.

In the next section, we describe the algorithms behind computing all routes or one route for selected target data for relational-to-relational schema mappings. We have extended our algorithms to handle selected source data, as well as the schema mapping language of [18, 11] to handle relational/XML-to-relational/XML schema mappings. We report experimental results for the XML case, but we limit our discussion in this paper to algorithms for selected target data with relational-to-relational schema mappings.

3. ROUTE ALGORITHMS

In this section, we formalize the notion of a *route* and describe algorithms, as well as properties of our algorithms for computing all routes or one route for a selected set of target tuples. A route illustrates the relationship between source and target data with the schema mapping. As an example, consider again the route for t_2 described in Scenario 3 of Section 2.1. The route for t_2 shows

findHom(I, J, t, σ)

In the following, let K denote I if σ is a s-t tgd, and K denotes J if σ is a target tgd.

Input: A source instance I , a solution J for I under \mathcal{M} , a tuple $t \in J$ of the form $R(\mathbf{a})$, and a tgd σ in $\Sigma_{st} \cup \Sigma_t$ of the form $\forall \mathbf{x} \phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})$.

Output: An assignment h such that $h(\phi(\mathbf{x})) \subseteq K$, $h(\psi(\mathbf{x}, \mathbf{y})) \subseteq J$ and $t \in h(\psi(\mathbf{x}, \mathbf{y}))$.

1. Let $R(\mathbf{z})$ be a relational atom of $\psi(\mathbf{x}, \mathbf{y})$. If no such relational atom can be found, return failure. Otherwise, let v_1 be a mapping that assigns the i th variable of \mathbf{z} to the i th value of \mathbf{a} in $R(\mathbf{a})$. If v_1 assigns a variable z to two different values under this mapping scheme, repeat step 1 with a different relational atom $R(\mathbf{z})$ of $\psi(\mathbf{x}, \mathbf{y})$. If no other relational atom $R(\mathbf{z})$ of $\psi(\mathbf{x}, \mathbf{y})$ can be found, return failure.
 2. Let v_2 be an assignment of variables in $v_1(\phi(\mathbf{x}))$ to values in K so that $v_2(v_1(\phi(\mathbf{x}))) \subseteq K$.
 3. Let v_3 be an assignment of variables in $v_2(v_1(\psi(\mathbf{x}, \mathbf{y})))$ to values in J so that $v_3(v_2(v_1(\psi(\mathbf{x}, \mathbf{y})))) \subseteq J$.
 4. Return $v_1 \cup v_2 \cup v_3$.
-

Figure 4: The findHom procedure.

the relationship $s_2 \xrightarrow{m_2} t_6 \xrightarrow{m_5} t_2$. In particular, it shows that s_2 and t_6 satisfy the tgd m_2 , and t_6 and t_2 satisfy the tgd m_5 . More specifically, a route is a sequence of *satisfaction steps*, which we define next.

DEFINITION 3.1. (Satisfaction step) Let σ be a tgd $\forall \mathbf{x} \phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})$. Let K and K_1 be instances such that K contains K_1 and K satisfies σ . Let h be a homomorphism from $\phi(\mathbf{x}) \wedge \psi(\mathbf{x}, \mathbf{y})$ to K such that h is also a homomorphism from $\phi(\mathbf{x})$ to K_1 . We say that σ can be satisfied on K_1 with homomorphism h and solution K , or simply σ can be satisfied on K_1 with homomorphism h , if K is understood from the context. The result of satisfying σ on K_1 with homomorphism h is K_2 , where $K_2 = K_1 \cup h(\psi(\mathbf{x}, \mathbf{y}))$ and $h(\psi(\mathbf{x}, \mathbf{y})) = \{R(h(\mathbf{z})) | R(\mathbf{z}) \text{ is a relation atom in } \psi(\mathbf{x}, \mathbf{y})\}$. We denote this step as $K_1 \xrightarrow{\sigma, h} K_2$.

EXAMPLE 3.2. In the example described earlier with $s_2 \xrightarrow{m_2} t_6 \xrightarrow{m_5} t_2$, the first satisfaction step is $(\{s_2\}, \emptyset) \xrightarrow{m_2, h_1} (\{s_2\}, \{t_6\})$, where $h_1 = \{an \mapsto 6689, s \mapsto 234, n \mapsto \text{A.Long}, a \mapsto \text{California}, M \mapsto M_1, I \mapsto I_1\}$. The result of satisfying m_2 on the instance $(\{s_2\}, \emptyset)$ with homomorphism h_1 and solution J of Figure 2 is $(\{s_2\}, \{t_6\})$. \square

We note that the instances K and K_1 in Definition 3.1 are instances over the schema $\langle \mathbf{S}, \mathbf{T} \rangle$ in our context, not necessarily satisfying the source or target constraints. We describe next a few technical differences between a satisfaction step, a chase step [8], and a solution-aware chase step [9]. First, unlike the definition of a chase step or solution-aware chase step for a tgd $\forall \mathbf{x} \phi(\mathbf{x}) \rightarrow \psi(\mathbf{x}, \mathbf{y})$ where h is defined only for \mathbf{x} , the homomorphism h in a satisfaction step is defined for variables in both \mathbf{x} and \mathbf{y} . In other words, $\phi(\mathbf{x}) \wedge \psi(\mathbf{x}, \mathbf{y})$ is completely defined under h . Second, when σ is satisfied on K_1 with homomorphism h , it may be that σ is already satisfied on K_1 with some other homomorphism h' where $h(x) = h'(x)$, for every $x \in \mathbf{x}$. For example, suppose σ is $S(x) \rightarrow \exists y T(x, y)$, $I = \{S(a)\}$, and $J = \{T(a, b), T(a, c)\}$. Let $h_1 = \{x \mapsto a, y \mapsto b\}$ and let $h_2 = \{x \mapsto a, y \mapsto c\}$. Clearly, $h_1(x) = h_2(x)$ and $(I, \emptyset) \xrightarrow{\sigma, h_1} (I, \{T(a, b)\}) \xrightarrow{\sigma, h_2} (I, J)$ is a route for J . After the first step of the route, the tgd σ is already satisfied with $(I, \{T(a, b)\})$, and therefore, σ cannot be applied on $(I, \{T(a, b)\})$ during a chase or a solution-aware chase. In contrast, we allow σ to be used again, together with a different homomorphism (e.g., h_2), to witness the existence of some other tuple

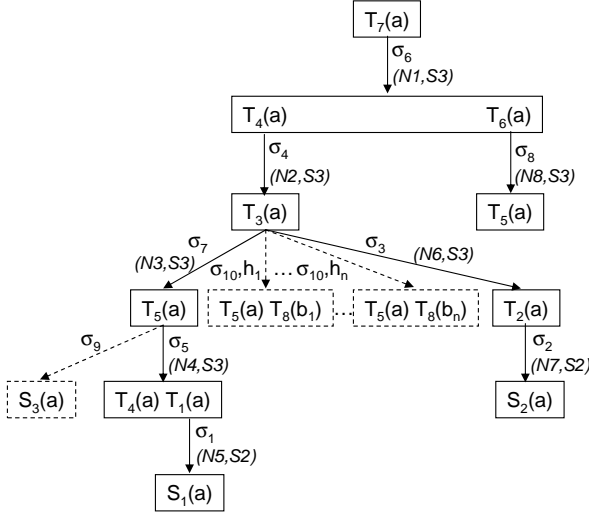


Figure 5: A route tree for $T_7(a)$.

(e.g., $T(a, c)$) not in $(I, \{T(a, b)\})$. Third, there is no corresponding definition for egds. This is because if K already satisfies an egd σ , then K_1 must also satisfy σ since it is contained in K . Hence there is no need to consider “egd satisfaction steps” in routes.

DEFINITION 3.3. (Route) Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ be a schema mapping, I be a source instance and J be a solution of I under \mathcal{M} . Let $J_s \subseteq J$. A route for J_s with \mathcal{M} , I and J (in short, a route for J_s) is a finite non-empty sequence of satisfaction steps $(I, \emptyset) \xrightarrow{m_1, h_1} (I, J_1) \dots (I, J_{n-1}) \xrightarrow{m_n, h_n} (I, J_n)$, where (a) $J_i \subseteq J$, $1 \leq i \leq n$, (b) m_i , $1 \leq i \leq n$, are among $\Sigma_{st} \cup \Sigma_t$, and (c) $J_s \subseteq J_n$. We say that the set of tuples produced by this route is J_n .

EXAMPLE 3.4. Referring to Example 3.2 and the source instance I and solution J of Figure 2, $(I, \emptyset) \xrightarrow{m_2, h_1} (I, \{t_6\})$ is a route for t_6 . The following is also a route for t_6 : $(I, \emptyset) \xrightarrow{m_2, h_1} (I, \{t_6\}) \xrightarrow{m_5, h_2} (I, \{t_6, t_2\})$, where $h_2 = \{s \mapsto 234, n \mapsto \text{A.Long}, m \mapsto M_1, i \mapsto I_1, a \mapsto \text{California}, N \mapsto N_1, L \mapsto 2K\}$. Note that the target instance produced by the second route for t_6 is $\{t_2, t_6\}$ and the last satisfaction step is redundant, in the sense that the first satisfaction step is sufficient as a route for t_6 . \square

3.1 Computing all routes

We show next an algorithm for computing all routes for a given set of tuples $J_s \subseteq J$, where J is any solution for a source instance I under the schema mapping \mathcal{M} . Note that our algorithm works for any solution and so, we are not limited to the solutions that are generated by Clío. Our algorithm constructs a route forest, in polynomial time in the size of I , J and J_s , that concisely represents all routes. We characterize what “all routes” means. More specifically, we show that every minimal route for a set of target tuples is, essentially, represented in this route forest. Intuitively, a minimal route for a set of target tuples is a route where none of its satisfaction steps can be removed and the result still forms a route for the set of target tuples. The algorithm for computing all routes is shown in Figure 3. It makes use of a `findHom` procedure shown in Figure 4. Intuitively, for every tuple t encountered during the construction of the route forest, our algorithm considers every

possible σ and h such that h is a possible assignment returned by `findHom`(I, J, t, σ). Conceptually, this corresponds to all possible (σ, h) pairs under the tuple t in the route forest. In other words, our algorithm explores all possibilities of witnessing t . We first examine `findHom` with an example and `ComputeAllRoutes` next.

Suppose we invoke `findHom`(I, J, t_1, m_1), where I, J, t_1 and m_1 are from Figure 1 and Figure 2. By “matching” t_1 with the atom `Accounts`(cn, l, s) of m_1 , step 1 of `findHom` defines v_1 as $\{cn \mapsto 6689, l \mapsto 15K, s \mapsto 434\}$. When v_1 is applied to the left-hand-side (LHS) of m_1 , we obtain the partially instantiated relational atom `Cards`(6689, 15K, 434, n, m, sal, loc). Hence, the assignment v_2 (step 2) is $\{n \mapsto \text{J. Long}, m \mapsto \text{Smith}, sal \mapsto 50K, loc \mapsto \text{Seattle}\}$. With $v_1 \cup v_2$, the LHS of m_1 corresponds to the tuple s_1 in the `Cards` relation, and the RHS of m_1 is the conjunction of the tuples `Accounts`(6689, 15K, 434) and `Clients`(434, Smith, Smith, 50K, A). Hence, step 3 of `findHom` returns v_3 as $\{A \mapsto A_1\}$. The algorithm then returns $v_1 \cup v_2 \cup v_3$ (step 4). In general, there are many possible assignments for v_1, v_2 and v_3 for a tgd σ . The algorithm looks for one combination of v_1, v_2 and v_3 that works. In our implementation (Section 3.3), we push the evaluation for v_2 and v_3 to the database. Hence, v_2 and v_3 can be derived efficiently in general.

EXAMPLE 3.5. Let \mathcal{M} be a schema mapping where Σ_{st} and Σ_t consists of the following tgds.

$$\begin{array}{ll} \Sigma_{st} : & S_1(x) \rightarrow T_1(x) \quad \sigma_1 \\ & S_2(x) \rightarrow T_2(x) \quad \sigma_2 \\ \Sigma_t : & T_2(x) \rightarrow T_3(x) \quad \sigma_3 \\ & T_3(x) \rightarrow T_4(x) \quad \sigma_4 \\ & T_4(x) \wedge T_1(x) \rightarrow T_5(x) \quad \sigma_5 \\ & T_4(x) \wedge T_6(x) \rightarrow T_7(x) \quad \sigma_6 \\ & T_5(x) \rightarrow T_3(x) \quad \sigma_7 \\ & T_5(x) \rightarrow T_6(x) \quad \sigma_8 \end{array}$$

Let the source instance I consists of two tuples $S_1(a)$ and $S_2(a)$ and a solution J for I under \mathcal{M} consists of tuples $T_1(a), \dots, T_7(a)$. Suppose we wish to compute all routes for $T_7(a)$. That is, we invoke `ComputeAllRoutes`($\mathcal{M}, I, J, \{T_7(a)\}$). The route forest (in this case, a tree) that is constructed by this algorithm is shown in Figure 5. (Please disregard the dotted branches for this example.) The order at which the branches are added to the forest and the steps involved in `ComputeAllRoutes` are labeled as a pair beside the branches in the tree. For example, $(N2, S3)$ for the branch with σ_4 denotes that this is the second branch added in the construction and it is added by step 3 of `ComputeAllRoutes`. In this example, h is always $\{x \mapsto a\}$ and so, we have omitted h from the figure.

In the process of constructing a route tree for $T_7(a)$, step 2 of `ComputeAllRoutes` fails to add any branches to $T_7(a)$. However, step 3 adds the σ_6 branch to $T_7(a)$ and continues the construction of the tree with `FindAllRoutes`($\{T_4(a), T_6(a)\}$). Finding a route for $T_4(a)$ leads to the tuple $T_3(a)$. There are two branches, σ_7 and σ_3 , for $T_3(a)$. In this computation, the σ_7 branch was explored before σ_3 and eventually, the σ_7 branch will cause `FindAllRoutes`($\{T_4(a), T_1(a)\}$) to be invoked. However, since $T_4(a)$ belongs to `ACTIVETUPLES` at this point, the branches for $T_4(a)$ are not explored at this node. Similarly, there are no branches under the tuple $T_5(a)$ under the σ_8 branch because $T_5(a)$ is an active tuple at this point. Intuitively, the branches for an active tuple t are added only when t is first encountered during the construction of the forest. \square

Obviously, the resulting forest that is constructed is not unique. For instance, if $T_6(a)$ was selected to be explored before $T_4(a)$ in `FindAllRoutes`($\{T_4(a), T_6(a)\}$), the constructed tree will be different from Figure 5. It is also easy to see that `ComputeAllRoutes` terminates since for each tuple t , there are only finitely many branches to add under t in steps 2 and 3. Furthermore, due

NaivePrint_F(J_s)

We denote by F the route forest returned by $\text{ComputeAllRoutes}_{\mathcal{M}}(I, J, J_s)$, where I is a source instance, J is a solution for I under \mathcal{M} and $J_s \subseteq J$.

Input: A set of tuples J_s where every tuple in J_s occurs in F . We assume that ANCESTORS is a global stack which is initially empty.

Output: A set of all routes for J_s .

For every tuple t in J_s

1. Push t into ANCESTORS. Goto any t in F .
2. Let L_1 denote the set of all (σ, h) branches under t such that σ is a s-t tgd.
3. Let L_2 denote the set of all (σ, h) branches under t such that σ is a target tgd and every tuple in $\text{LHS}(h(\sigma))$ does not occur in ANCESTORS.
4. Let $L_3 = \emptyset$.
5. For every (σ, h) in L_2
 - (a) Let L' denote $\text{NaivePrint}_F(\text{LHS}(h(\sigma)))$.
 - (b) Append (σ, h) to every element in L' .
 - (c) $L_3 = L_3 \cup L'$.
6. Let $L(t)$ be $L_1 \cup L_3$.
7. Pop ANCESTORS.

Return $L(t_1) \times \dots \times L(t_k)$, where $J_s = \{t_1, \dots, t_k\}$.

Figure 6: An algorithm for printing all routes.

to ACTIVETUPLES, the branches of an active tuple are only explored at one place in the forest. Hence, ComputeAllRoutes runs in polynomial time in the size of I , J and J_s since there can only be polynomially many branches under each tuple.

PROPOSITION 3.6. *Let \mathcal{M} be a schema mapping. Let I be a source instance, J be a solution for I under \mathcal{M} and $J_s \subseteq J$. Then, $\text{ComputeAllRoutes}_{\mathcal{M}}(I, J, J_s)$ executes in polynomial time in the size of I , J and J_s .*

From the tree in Figure 5, it is easy to see that a route, R_1 , for $T_7(a)$ is:

$$R_1: I \xrightarrow{\sigma_2} I, T_2 \xrightarrow{\sigma_3} I, T_2, T_3 \xrightarrow{\sigma_4} I, T_2, T_3, T_4 \xrightarrow{\sigma_1} I, T_1, \dots, T_4 \\ \xrightarrow{\sigma_5} I, T_1, \dots, T_5 \xrightarrow{\sigma_8} I, T_1, \dots, T_6 \xrightarrow{\sigma_6} I, T_1, \dots, T_7$$

For conciseness, we have written T_i instead of $T_i(a)$ above. If there is another s-t tgd $\sigma_9 : S_3(x) \rightarrow T_5(x)$ and suppose I also contains the source tuple $S_3(a)$, then there would be another branch under the tuple $T_5(a)$ under σ_7 . (See the leftmost dotted branch in Figure 5.) This would mean there is another route for $T_7(a)$:

$$R_2: I \xrightarrow{\sigma_9} I, T_5 \xrightarrow{\sigma_7} I, T_5, T_3 \xrightarrow{\sigma_4} I, T_5, T_3, T_4 \\ \xrightarrow{\sigma_8} I, T_5, T_3, T_4, T_6 \xrightarrow{\sigma_6} I, T_5, T_3, T_4, T_6, T_7$$

Observe that in this route, we have bypassed $T_1(a)$ since we can now witness $T_5(a)$ directly with the s-t tgd σ_9 .

Completeness of the route forest in representing all routes We show next that the route forest generated by ComputeAllRoutes is complete in the sense that every minimal route for J_s can, essentially, be found in this route forest. More specifically, we show that every minimal route for J_s is represented by one of the routes we naively generate from this forest. Our procedure for naively generating routes for J_s is shown in Figure 6. It finds the set of all routes for every tuple t in J_s and takes a cartesian product of these sets of routes in the last step. Observe that in step 1, we allow the search for routes to start from any occurrence of t in F . Even though there may be many occurrences of t in F , we assume that every other occurrence of t has a link to the first t in F where the branches of t are explored. For example, $T_4(a)$ under the σ_5 branch has a reference to $T_4(a)$ under the σ_6 branch. NaivePrint on $T_7(a)$ will produce a route $\sigma_2 \rightarrow \sigma_3 \rightarrow \sigma_4$ for $T_4(a)$ and a

route $\sigma_2 \rightarrow \sigma_3 \rightarrow \sigma_4 \rightarrow \sigma_1 \rightarrow \sigma_5 \rightarrow \sigma_8$ for $T_6(a)$. For conciseness, we have listed only the tgds involved. Hence the route produced for $T_7(a)$ is

$$R_3: \sigma_2, \sigma_3, \sigma_4, \sigma_2, \sigma_3, \sigma_4, \sigma_1, \sigma_5, \sigma_8, \sigma_6,$$

Obviously, this route contains some redundant satisfaction steps. A minimal route for $T_7(a)$ is in fact R_1 , where none of its satisfaction steps are redundant. In other words, it does not contain redundant satisfaction steps. Although R_3 is not a minimal route, it has the same set of satisfaction steps as the route R_1 . To compare routes based on the satisfaction steps they use, rather than the order in which the satisfaction steps are used, we introduce an interesting concept called the *stratified interpretation* of routes. To stratify a route, we make use of the concept of the *rank of a tuple* in a route. Intuitively, every tuple is associated with a unique rank in a route. Source tuples have rank 0 and a tuple t has rank k in a route if for some satisfaction step in the route that involves σ and h , (1) t occurs in $\text{RHS}(h(\sigma))$, (2) the maximum rank of tuples in $\text{LHS}(h(\sigma))$ is $k - 1$, and (3) t is not of a lower rank. The *stratified interpretation* of a route R , denoted as $\text{strat}(R)$, partitions the pairs (σ, h) in R into blocks. We say that (σ, h) of a route belongs to rank 1 if $\text{LHS}(h(\sigma))$ consists of only source tuples and it belongs to rank k if the maximum rank of tuples in $\text{LHS}(h(\sigma))$ is $k - 1$. The *rank of a route* is the number of blocks in the stratified interpretation of the route. For example, R_1 and R_3 have the same stratified interpretation shown below and they both have rank 6.

Rank 1	2	3	4	5	6
σ_1, σ_2	σ_3	σ_4	σ_5	σ_8	σ_6

We say that two routes R and R' have the same stratified interpretations, denoted as $\text{strat}(R) = \text{strat}(R')$, if for every block of rank i in $\text{strat}(R)$, every (σ, h) in this block can be found in the corresponding i th block of $\text{strat}(R')$ and vice versa. We show that for any $J_s \subseteq J$, the forest F that is constructed by $\text{ComputeAllRoutes}_{\mathcal{M}}(I, J, J_s)$ contains all routes, in the sense that every minimal route for J_s has the same stratified interpretation as one of the routes produced by $\text{NaivePrint}_F(J_s)$. Note that when we say two routes R and R' are equivalent if they have the same stratified interpretation, this is in fact the same as saying that R and R' have the same set of satisfaction steps. The stratified interpretation of routes, however, comes naturally in the proof of our completeness result. Furthermore, stratified routes are also more easily understandable, when compared to a sequence of satisfaction steps. We plan to include the ability to display stratified routes in our visual interface [2].

THEOREM 3.7. *Let \mathcal{M} be a schema mapping. Let I be a source instance and J be a solution for I under \mathcal{M} . Let $J_s \subseteq J$ and let F denote the route forest of $\text{ComputeAllRoutes}_{\mathcal{M}}(I, J, J_s)$. If R is a minimal route for J_s , then there exists a route R' in the result of $\text{NaivePrint}_F(J_s)$ with the property that $\text{strat}(R) = \text{strat}(R')$.*

Observe that there could be exponentially many routes for J_s , but our route forest is a compact, polynomial-size representation of all routes. Our experimental results in Section 4 indicate that it may be expensive to construct the route forest in general. Hence, a natural question is whether we can produce one route fast and leverage the “debugging-time” of the user to produce other routes as needed.

3.2 Computing one route

In debugging, we believe it is also useful to have the alternate feature where we can derive and display one route fast and display other routes, as needed. Our experimental results in Section 4 justify that in most cases, it is much faster to compute one route than

Algorithm ComputeOneRoute $_{\mathcal{M}}(I, J, J_s)$

Input: A source instance I , a solution J for I under \mathcal{M} and a set of tuples $J_s \subseteq J$.

Output: A route for J_s .

Global data structures:

- A set of ACTIVETUPLES that contains tuples for which the algorithm has attempted to find a route. Initially, this set is empty.
- A set UNPROVEN that contains unproven triples, initially empty.
- Every tuple has a status proven or unproven.
- A sequence of pairs G used to contain the route, initially empty.

FindRoute(J_s)

For every tuple t in J_s

If t is not in ACTIVETUPLES, then

1. Add t to ACTIVETUPLES.
2. If findHom(I, J, t, σ) returns h for some s-t tgd σ , then
 - (a) Append (σ, h) to G .
 - (b) Infer($\{t\}$).
 - (c) Continue with the next iteration of For-Loop.
3. If findHom(I, J, t, σ) returns h for some target tgd σ , then
 - (a) If LHS($h(\sigma)$) consists of only proven tuples, then
 - (i) Append (σ, h) to G .
 - (ii) Infer($\{t\}$).
 - Else
 - (iii) Add (t, σ, h) to UNPROVEN.
 - (iv) FindRoute(LHS($h(\sigma)$)).
 - (v) If (t, σ, h) is UNPROVEN, continue with step 3.

Return G .

Figure 7: An algorithm for computing a route.

Infer(S)

Repeat until $S = \emptyset$

1. Mark all tuples in S as proven.
2. Let $S = \emptyset$.
3. For every triple (t', σ', h') in UNPROVEN
 - (a) If LHS($h'(\sigma')$) consists of only proven tuples, then
 - (i) Add t' to S .
 - (ii) Remove (t', σ', h') from UNPROVEN.
 - (iii) Append (σ, h) to G .

Figure 8: The Infer procedure used by ComputeOneRoute.

compute all routes. We believe that in general however, it is valuable to incorporate both features for a debugger. In some cases, the user may be satisfied with one route, which is faster to compute than computing all routes. It is also useful, however, to be able to determine all routes whenever desired.

We describe next our algorithm **ComputeOneRoute** which computes a route for a given set of tuples $J_s \subseteq J$, where J is any solution for I under the schema mapping \mathcal{M} . The algorithm for computing one route is shown in Figure 7. It uses two procedures **Infer** in Figure 8 and **findHom**, in Figure 4, which was described earlier. We examine the algorithm **ComputeOneRoute** in some detail with an example next and make a comparison with **ComputeAllRoutes** after this.

EXAMPLE 3.8. Let \mathcal{M} , I and J be the schema mapping, source and target instances given in Example 3.5. Suppose a route for $T_7(a)$ is sought. With **ComputeOneRoute** $_{\mathcal{M}}(I, J, \{T_7(a)\})$, we obtain the same route tree of Figure 5. (Please disregard the dotted branches.) The computation that occurs during the construction, however, is different. With the tuple $T_7(a)$, **ComputeOneRoute** fails to find a s-t tgd in step 2 for $T_7(a)$. Hence, it proceeds to Step 3 and succeeds in finding homomorphisms with $T_7(a)$ and σ_6 . (As before, we have omitted h as it is always $\{x \mapsto a\}$ in this example.)

Consequently, **FindRoute**($\{T_4(a), T_6(a)\}$) is invoked. Similarly, for $T_4(a)$, **findHom** succeeds with σ_4 . For $T_5(a)$, **findHom** succeeds with σ_5 . In the branch with σ_5 , **FindRoute**($\{T_4(a), T_1(a)\}$) is invoked. Since $T_4(a)$ occurs in ACTIVETUPLES, the for-loop for **FindRoute** for $T_4(a)$ is not executed. Instead, **FindRoute** continues with the tuple $T_1(a)$ and succeeds with a s-t tgd σ_1 . Since **findHom** does not succeed with other tgds on $T_5(a)$, the algorithm returns to $T_3(a)$. It happens that, so far, the computation resembles **ComputeAllRoutes**. Continuing from $T_3(a)$, the algorithm succeeds in witnessing $T_3(a)$ with σ_3 and σ_2 . At σ_2 , the set UNPROVEN is $\{\sigma_6, \sigma_4, \sigma_7, \sigma_5, \sigma_3\}$, and G is the sequence $[\sigma_1, \sigma_2]$. When **Infer**($\{T_2(a)\}$) is invoked (see step 2(b)), the algorithm will deduce that $T_3(a)$, $T_4(a)$, $T_5(a)$ and $T_3(a)$ are proven, in this order, and G is now $[\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_7]$. After this, the algorithm returns to the branch σ_6 and attempts to find a route for $T_6(a)$ next. It succeeds with σ_8 because $T_5(a)$ is already proven and it will infer that $T_7(a)$ is proven with **Infer**($\{T_6(a)\}$) (see step 3(a-ii) of **ComputeOneRoute**). The algorithm successfully terminates and returns $[\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_7, \sigma_8, \sigma_6]$. \square

Comparisons between compute all routes and one route In **ComputeOneRoute**, the algorithm searches for one successful branch under a tuple t to find a route for t . In **ComputeAllRoutes**, however, all branches are searched, regardless of whether a route for t has already been found. To make a better contrast, suppose Σ_{st} contains another s-t tgd $\sigma_9 : S_3(x) \rightarrow T_5(x)$ and additionally, we have the source tuple $S_3(a)$. Then, for the tuple $T_5(a)$ that sits under the branch σ_7 in Figure 5, only the branch σ_9 will be considered. This is because the algorithm considers s-t tgds before target tgds (step 2 of **ComputeOneRoute**). Since a route for $T_5(a)$ can be found with the s-t tgd σ_9 , the branch with σ_5 will not be explored.

The second difference is that the result returned by **ComputeOneRoute** is a sequence of (σ, h) pairs that represents the route that is found for J_s , even though a route forest is constructed during the computation. These (σ, h) pairs are collected during the construction of the forest in **ComputeOneRoute**.

The third difference is that in **ComputeOneRoute**, we now have an **Infer** procedure to infer proven tuples as we construct the forest. This procedure is needed for the correctness of the algorithm. To see this, consider Example 3.8 again. Now consider the execution of **ComputeOneRoute** for $T_7(a)$ without **Infer**. At the σ_2 branch, no inference will be made. Although we can still conclude that $T_3(a)$ and $T_4(a)$ are proven as we return along the branches σ_3 and σ_4 respectively, observe that we cannot conclude that $T_5(a)$ is proven. Hence, without **Infer**, the status of $T_5(a)$ is unknown at the point when σ_8 is used and because $T_5(a)$ is in ACTIVETUPLES, the branches under $T_5(a)$ are not explored. The algorithm therefore terminates with a partial route for $T_7(a)$, which is incorrect. One might argue that we should remove the ACTIVETUPLES restriction so as to allow the branches under $T_5(a)$ to be explored again. In this case, since both $T_4(a)$ and $T_1(a)$ are proven by the time we explore the branches of $T_5(a)$ under σ_8 , we conclude that $T_5(a)$ is proven and so the algorithm terminates with a route for $T_7(a)$. However, without the ACTIVETUPLES restriction, there might be many unnecessary explorations. To see this, suppose the schema mapping of Example 3.8 has an additional target tgd $\sigma_{10} : T_5(x) \wedge T_8(y) \rightarrow T_3(x)$ and the target instance J consists of n additional tuples $T_8(b_1), \dots, T_8(b_n)$. Observe that J is a solution for I under \mathcal{M} with these n additional tuples. When $T_3(a)$ is encountered during the construction of a route for $T_7(a)$, it may happen that the σ_3 branch is explored last. (Refer to $T_3(a)$ of Figure 5.) Hence, the branches of $T_5(a)$ are repeatedly explored along the branches $\sigma_7, (\sigma_{10}, h_1), \dots, (\sigma_{10}, h_n)$,

where $h_i = \{x \mapsto a, y \mapsto b_i\}$. The repeated exploration of the branches of $T_5(a)$ in this case is unnecessary. We also remark that the ACTIVETUPLES restriction makes the running time analysis simpler. As in `ComputeAllRoutes`, since every (σ, h) pairs for a tuple t occurs at most once in the route forest, there are at most polynomially many branches in the forest constructed by `ComputeOneRoute`. In `ComputeOneRoute`, however, one also has to reason about the running time of `Infer`.

PROPOSITION 3.9. *Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$. Let I be a source instance, J be a solution for I under \mathcal{M} and $J_s \subseteq J$. Then, `ComputeOneRoute` $_{\mathcal{M}}(I, J, J_s)$ executes in polynomial time in the size of I, J and J_s .*

We note that our running time analysis is based on the size of I, J and J_s . The default behavior of our debugger, however, uses the solution J that is generated by `Clio` on a given source instance I under a schema mapping \mathcal{M} . Therefore, a natural question is whether the polynomial time results of Proposition 3.6 and Theorem 3.9 hold when analyzed against the size of the source instance I . It is easy to show that if J is polynomial in the size of I , then our route algorithms run in polynomial time in the size of I . Since `Clio` generates a solution J that is polynomial in the size of I under a relational-to-relational schema mapping [8], our route algorithms run in polynomial time in the size of I in this setting as well.

Next, we show that our algorithm is complete for finding one route. If there is a route for J_s , then `ComputeOneRoute` on J_s will produce a route for J_s .

THEOREM 3.10. *Let \mathcal{M} be a schema mapping, I be a source instance and J be a solution for I under \mathcal{M} . For every $J_s \subseteq J$, if there is a route for J_s , then `ComputeOneRoute` $_{\mathcal{M}}(I, J, J_s)$ will produce a route for J_s .*

3.3 Some implementation details

Although we have only described how one can compute routes for a set of tuples with relational-to-relational schema mappings, we have extended and implemented our route algorithms to handle schema mappings where the source or target schemas may be hierarchical. Our implementation uses the nested relational model as our underlying representation and the mapping language of `Clio` [18, 26] to represent schema mappings. The system is implemented in Java 1.5. Currently, we store relational instances using DB2 UDB Personal Edition release 8.1.10, while XML instances are stored as XML documents. We use DB2’s query engine and Saxon-SB 8.6 XSLT transformation engine to run SQL and respectively, XSLT queries over relational and respectively, XML instances.

In the `findHom` procedure (Figure 4), the required assignments v_1, v_2 and v_3 for a given tuple t are obtained as follows. First, we obtain v_1 by matching t against the RHS of the tgd σ . Second, we run the LHS of σ as a selection query (as indicated by v_1) against the instance K to obtain all the assignments v_2 that agree with v_1 . (Here, K is the source or target instance, depending on whether σ is a s-t or target tgd.) For each such v_2 , we obtain possible assignments v_3 that agree with v_2 by running the RHS of σ as an appropriate (based on v_2) selection query on the target instance. Note that all possible assignments of v_2 and v_3 could in fact be obtained by running a single selection query (the join of the LHS and RHS of σ) against the source and target instances. While this may be more efficient for relational schema mappings, it was a design choice to run two separate queries, in order to handle general situations in which for example, the source instance is relational, while the target is XML. We fetch the assignments one at a time, as needed, from the result of the selection queries. For this reason,

$$\begin{array}{l} 1 \text{ join} \\ S \bowtie_{\text{supkey}} L, O \bowtie_{\text{custkey}} C, PS \bowtie_{\text{partkey}} P, N \bowtie_{\text{nationkey}} R \\ 2 \text{ joins} \\ S \bowtie_{\text{supkey}} L \bowtie_{\text{orderkey}} O, S \bowtie_{\text{supkey}} PS \bowtie_{\text{partkey}} P, \\ C \bowtie_{\text{nationkey}} N \bowtie_{\text{nationkey}} R \\ 3 \text{ joins} \\ S \bowtie_{\text{supkey}} L \bowtie_{\text{partkey, supkey}} PS \bowtie_{\text{partkey}} P, \\ O \bowtie_{\text{custkey}} C \bowtie_{\text{nationkey}} N \bowtie_{\text{nationkey}} R \end{array}$$

Figure 9: Joins used in tgds in the relational and flat-hierarchy synthetic scenarios.

our implementation of `ComputeOneRoute` is scalable for relational instances. For XML instances, however, all the assignments are fetched at once, since the result produced by the Saxon engine is stored in memory.

Our implementation of `ComputeOneRoute` is an optimization of the algorithm in Figure 7. If the `findHom` step for tuple t is successful with some tgd σ in steps 2 or 3 of the algorithm, we conclude that *all* the target tuples produced by σ (and not only t) are proven. Hence, we may avoid performing redundant `findHom` steps with the rest of the tuples.

3.4 Some other features of our debugger

We briefly mention here some other features of our debugger. We refer the interested reader to [2] for a more detailed description on these features. Besides computing one or all routes for selected target tuple, our system is also capable of computing one or all routes for selected source tuples. We have also extended our algorithms for computing one route to generate alternative routes at the user’s request. Our system is also equipped with “standard” debugging features such as breakpoints on tgds, single-stepping the computation of routes and a “watch” window for visualizing how the target instance changes, as well as the assignments for variables used in a tgd at each step.

4. EXPERIMENTAL EVALUATION

We have experimentally evaluated our debugger on both real and synthetic datasets to assess its efficiency in computing one or all routes, under the effect of various parameters. We present our experimental results with both route algorithms. All our experiments were executed on a Pentium 4, 2.8GHz Windows machine with 2GB RAM. The DB2 buffer pool was set to 256MB. Each experiment was repeated three times and we report execution times averaged over the second and third runs.

4.1 The synthetic datasets

We designed three synthetic scenarios, called **relational**, **flat-hierarchy** and **deep-hierarchy**. The first two scenarios are designed with the goal of measuring the influence of various parameters, when the source and target schemas are relational and respectively, hierarchical. The parameters are: the size of source (and target) instances, the number of tuples selected by a user and the size and complexity of the schema mappings. The third scenario is designed to measure the influence of the depth of the selected elements of an XML document on the performance of our algorithms, where both the source and target schemas are deeply nested. In what follows, we describe the construction of each scenario and present our experimental results on both route algorithms. In summary, we have observed that `ComputeOneRoute` can be efficiently executed, while `ComputeAllRoutes` may perform orders of magnitude slower compared to `ComputeOneRoute`.

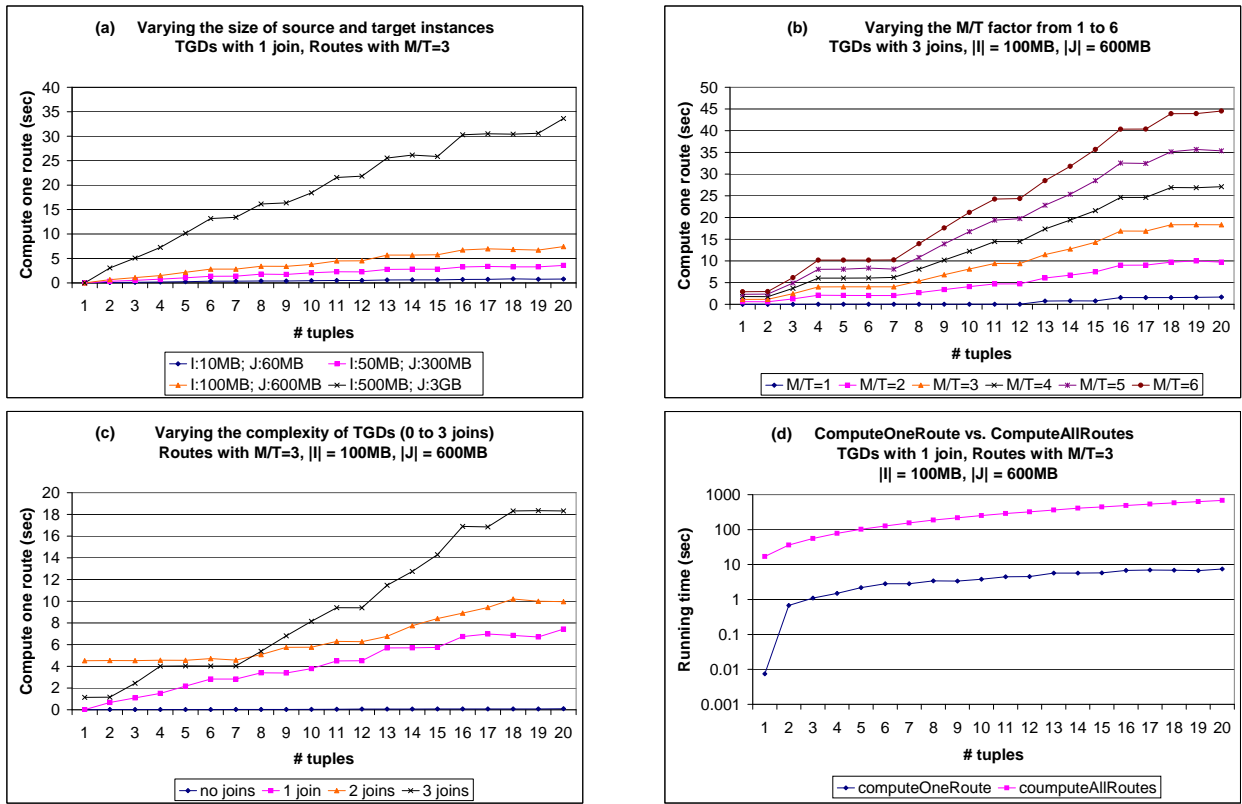


Figure 10: (a-c) Performance evaluation of ComputeOneRoute and (d) comparison in performance between ComputeOneRoute and ComputeAllRoutes in the relational synthetic scenarios.

Relational scenario We designed four schema mappings $\mathcal{M}_0, \dots, \mathcal{M}_3$ for our experiments. The subscripts denote the number of joins, which we shall explain shortly, used in the tgds of the schema mapping. In each schema mapping, the source schema conforms to the TPC Benchmark H (TPCH) Standard Specification Revision 2.1.0 [21] and consists of eight relations Customer (C), Lineitem (L), Nation (N), Orders (O), Part (P), Partsupp (PS), Region (R) and Supplier (S). The target schema consists of six “copies” of the source schema: for each relation R_0 in the source schema, there are six relations $R_i, i \in [1, 6]$, identical to R_0 in the target schema. Hence, the target schema can be viewed as having six groups of relations, where each group is a “copy” of the source schema. In the first schema mapping \mathcal{M}_0 , the s-t tgds populate relations in the first group by copying every R_0 to the corresponding relation R_1 in the target. The target tgds are such that every relation R_i in the i th group, $i \in [2, 6]$, is copied from the corresponding relation R_{i-1} . Finally, no target egds appear in our target dependencies, since egds do not influence the performance of our algorithms. The second schema mapping \mathcal{M}_1 is similar except that every tgd in this schema mapping has 1 join on both sides corresponding to the 1 join case illustrated in Figure 9. For example, one such s-t “copying” tgd in \mathcal{M}_1 is $S_0(sk, \dots) \wedge L_0(\dots, sk, \dots) \rightarrow S_1(sk, \dots) \wedge L_1(\dots, sk, \dots)$, where the variable sk corresponds to the supplier key attribute (*sup-key*) shown in Figure 9. We have omitted the rest of the variables for conciseness. The schema mappings \mathcal{M}_2 and \mathcal{M}_3 are similar except that the tgds have 2 and respectively, 3 joins on both sides as shown in Figure 9. We note that using only such “copying” tgds in these four schema mappings does not bias our empirical evaluation; our debugger separately operates with the LHS and RHS of

each tgd in findHom and we have varied the complexity of each side of the tgds in each schema mapping.

In Figure 10(a), we study the influence that the size of the input (i.e., the size of source and target instances, as well as the number of tuples for which a route needs to be computed) has on the performance of ComputeOneRoute for a fixed schema mapping. The sizes of (I, J) are (10MB, 60MB), (50MB, 300MB), (100MB, 500MB), (500MB, 3GB) respectively. The number of selected tuples is varied between 1 and 20. To keep the comparison meaningful, all tuples were selected at random from the same group of target relations so that the number of satisfaction steps in a route of each selected tuple (called M/T factor) is kept constant. For example, M/T = 2 for tuples of relations in group 2, since these tuples are witnessed with one s-t tgd and one target tgd. Figure 10(a) illustrates the time required to compute one route for tuples in group 3 in the schema mapping with 1 join tgds (\mathcal{M}_1). As expected, the running time increases as the number of selected tuples increases, since more findHom steps need to be taken, hence more queries are executed. The execution time also increases with the size of the source and target instances. Routes for 10 and 20 tuples are computed in under 4, and respectively, under 8 seconds on the datasets with 10MB, 50MB and 100MB source instances. However, the performance degrades to a larger extent on the dataset where the source and target instances are of size 500MB and respectively, 3GB. This is not unexpected, since the queries, with joins involved, are now executed on larger instances. We believe, however, that a user is unlikely to select as many as 20 tuples in the first place. We expect a user to be interested in a smaller number of tuples at any time, cases in which our system would still perform well.

In Figure 10(b), we analyze the influence of the M/T factor on the performance of `ComputeOneRoute`, by computing routes for target tuples in different groups. We performed six runs, each time selecting up to 20 tuples from the same group (i.e., the M/T factor varies between 1 and 6). The source instance is fixed at 100MB in the schema mapping with 3 joins tgds (\mathcal{M}_3). As expected, the running time increases with a higher M/T factor, since more intermediary tuples are discovered (and consequently, more invocations of `findHom` are made) along the route to source tuples, hence more queries are executed. For example, it takes 1.8, and respectively, 2.9 seconds to find a route for a tuple with an M/T factor of 3 and respectively, 6.

In Figure 10(c), we analyze the influence of the complexity of schema mappings on the performance of the system. This time, we vary the schema mapping with 0 to 3 joins tgds and we fix the M/T factor to 3 and the size of the source instance to 100MB. The running time of `ComputeOneRoute` increases with the number of joins in the tgds. This is, again, not unexpected, since the performance of executing queries degrades with the number of joins. Still, the system performs well, taking up to 4.5 seconds to compute routes for a set of 7 target tuples, in all four schema mappings.

We have performed a similar suite of experiments with `ComputeAllRoutes` and observed similar trends (graphs not shown). As expected however, `ComputeAllRoutes` performs slower compared to `ComputeOneRoute`. Figure 10(d) shows a comparison between the running times of the two algorithms for the schema mapping with 1 join, the source instance of 100MB and tuples with an M/T factor of 3. (Note the logarithmic scale). For 5 tuples, one route is found and printed in 2 seconds, while `ComputeAllRoutes` requires about 100 seconds to construct the route forest. The running time shown for `ComputeAllRoutes` does not include the time required to print all routes from the route forest (algorithm `NaivePrint`). The performance gap between the two algorithms will be even larger if we require all routes to be printed.

Flat-hierarchy scenario We have studied the influence that the size of source (and target) instances, as well as the number of elements for which a route needs to be computed have on the performance of `ComputeOneRoute` for the XML case. We also performed experiments to analyze the influence of the M/T factor, as well as the complexity of the schema mappings. (We have omitted the graphs here, for lack of space.) The source schema consists of a root record having eight sets of records nested underneath, each set corresponding to one TPCB relation. Similarly, the target schema consists of six “copies” of the source schema and the s-t and target tgds are similar to our relational scenario (i.e., they are “copying” tgds). Hence, in this scenario we deal only with elements nested immediately underneath the root (i.e., the depth is 1). For our experiments, the sizes of (I, J) we use are (500KB,3MB), (1MB,6MB) and (5MB,30MB) respectively. As expected, the running time of `ComputeOneRoute` increases with the size of the source and target instances, as well as the number of selected target elements that need to be justified. The system performs very well, requiring at most 5 seconds to compute one route for 20 elements, for all three pairs of source and target instances. We observed that the performance of the algorithm decreases with the increase of the M/T factor, as in the relational case. However, we noticed a more drastic decrease in performance with the increase in the number of joins in the tgds. This is not unexpected, since the free version of Saxon XSLT engine which we use in the `findHom` procedure does not perform join reordering and simply implements all for-each clauses as nested loops.

Deep-hierarchy scenario To analyze the effect of the depth of selected elements on the performance, we designed a schema map-

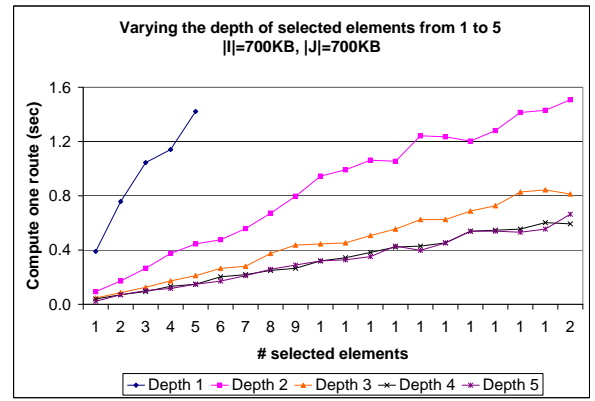


Figure 11: Performance evaluation of `ComputeOneRoute` in the deep hierarchy scenario.

ping where the source and target schemas are identical and have the nesting `Region/Nation/Customers/Orders/Lineitem`. In other words, the root consists in a set of regions, each region has nations nested underneath and so on. The set Σ_{st} consists of one s-t tgd that copies the source instance into an identical target instance and there are no target tgds. We tested the performance of `ComputeOneRoute` on elements found at different nesting levels in the target instance. For example, we picked `Customer` and respectively, `Lineitem` elements for our experiments with levels 3 and 5, respectively. The results are shown in Figure 11. The execution time *decreases* with the depth of the selected element. Intuitively, with a deeper selected element, more variables will be instantiated in the selection queries generated by `findHom`. Hence, the resulting selection queries will execute faster. We note that for elements of depth 1, we report the execution time for at most 5 selected region facts, since there are only 5 distinct regions in the TPCB instance.

4.2 The real datasets

We also evaluated our system using two real datasets (**DBLP** and **Mondial**) for which we created schema mappings in order to exchange bibliographical, and respectively, geographic information. For the DBLP scenario, we obtained two DBLP data sources. As a target schema, we used the first relational schema in the Amalgam integration test suite [16]. In the Mondial scenario, we used the relational and nested versions of the Mondial schema [17], as source, and respectively, target schemas. In both cases, we generated the s-t tgds and we used the foreign key constraints of the target schemas as target tgds. Some characteristics of the source and target schemas, the number of s-t and target tgds, as well as the size of source and resulting target instances used in our experiments are shown in Table 1. We used our debugger to compute (one or all) routes for one to ten randomly selected target tuples in both scenarios. The time required to find one route was under 3 seconds in all cases, while computing all routes took much longer. For example, a route was computed in under 1 second, while `ComputeAllRoutes` took about 18 seconds to construct the routes forest for a set of 10 target elements in the Mondial scenario.

5. RELATED WORK

A framework for understanding and refining schema mappings in Clío, where both the source and target schemas are relational, is proposed in [25]. The main focus of [25] is the selection of a good source and target instance that is illustrative of the behavior of the schema mappings. Our debugger differs from [25] in that: (1) Our

debugger works for relational or XML schema mappings. (2) We allow *any* source instance to be used for debugging the schema mapping. Since the instances are crafted in [25], routes are predetermined. In our case, we do not generate instances and routes are computed only “on demand”. We allow a user to create and use any source instance that she thinks is representative for debugging, and this is similar to creating test cases for testing the correctness of a program during a software development cycle. The work of [25] is thus complementary to our debugger. It would be desirable to incorporate the functionality of [25] into our debugger and investigate what are representative instances for debugging in general.

Commercial systems such as Altova’s MapForce [14] and Stylus Studio [19] ship with integrated debugging facilities for data exchange. These systems rely directly on “lower-level” languages such as XSLT or XQuery for specifying the exchange. Hence, their built-in debugging tools are simply XSLT or XQuery debuggers. Our debugger, however, debugs at the level of schema mappings.

The problem of computing a route is similar in spirit to the problem of computing the *provenance* (or *lineage*) of data. Our route algorithms also bear resemblance to top-down resolution techniques used in deductive databases. In the next two sections, we compare our work with related work in these areas in more detail.

5.1 Computing the provenance of data

Cui et al. [7] studied the problem of computing the provenance of relational data in a view in the context of data warehousing. The provenance of a tuple in a view is described as the tuples in the base tables that witness the existence of that tuple. Whenever the provenance of a view tuple t is sought, the approach of [7] is to generate a query to retrieve all combinations of base tuples that together with the view definition justify the existence of t . This type of provenance is also called *why-provenance* in [6].

There are several differences between our work and the approach of [7]. First, observe that part of the input to our route algorithms is (I, J) , where J is a *solution* for I under the schema mapping. Since J is any solution, there may exist tuples in J with *no* routes. In contrast, in the context of [7], the equivalent of J is the output of an SQL query executed over I . Consequently, the provenance of every tuple in J always exists. The second difference lies in the representation of provenance. Our route algorithms operate with schema mappings, and not with SQL queries as in [7]. In our case, a tuple in J may relate to several other intermediate tuples in I and J through possibly different tgds. Our route captures these “intermediate relationships” between tuples. In contrast, the provenance of a tuple t as defined in [7] is the set of source tuples, that will witness t according to the SQL query. (We will exemplify this point after we describe the next difference.) The third difference is that the language of schema mappings allows one to define recursive computations. In contrast, recursive views are not handled in [7]. Even if recursive views were handled in [7], the description of provenance only with source tuples can be unsatisfactory as the following example illustrates. Consider a schema mapping \mathcal{M} where Σ_{st} consists of one s-t tgd $\sigma_1 : S(x, y) \rightarrow T(x, y)$ and Σ_t consists of a target tgd $\sigma_2 : T(x, y) \wedge T(y, z) \rightarrow T(x, z)$ that defines the transitive closure of the binary relation T . Let the source instance $I = \{s_1 : S(1, 2), s_2 : S(2, 3)\}$ and the target instance $J = \{t_1 : T(1, 2), t_2 : T(2, 3), t_3 : T(1, 3)\}$ which is a solution for I under \mathcal{M} . Clearly, the two source tuples witness t_3 with \mathcal{M} , but this is not as informative as showing a route $s_1 \xrightarrow{\sigma_1} t_1, s_2 \xrightarrow{\sigma_1} t_2, \{t_1, t_2\} \xrightarrow{\sigma_2} t_3$ that describes all the intermediate relationships. In particular, the fact that t_3 is a consequence of t_1 and t_2 with σ_2 is captured in the route. (For simplicity, we have omitted the homomorphisms in each step and showed only the rel-

evant tuples in the source instance.) Consequently, the process of computing a route for t is more complex in our case, since it is no longer sufficient to pose a single query over the source instance as in [7]. Fourth, we have extended our approach for computing routes in the context of schema mappings where the source and target schemas can be nested, while the approach of [7] handles only relational views defined over relational sources. Lastly, [7] handles aggregates and negation. The language of schema mappings we consider cannot express aggregates or negation.

The approach of [7] is *lazy* in the sense that the SQL query that describes the transformation is not re-engineered and provenance is computed by (subsequently) examining the query, and the source and target databases. In contrast, several systems such as Explain [3], DBNotes [5], the MXQL system of [23], and recently Mondrian [10], adopt the *bookkeeping* or *eager* approach for computing provenance. In these systems, the transformation is re-engineered to keep extra information from the execution. Consequently, provenance can often be answered by examining only the target database and the extra information. Explain is an explanative module for the CORAL deductive database system. Explain records additional information during the execution of a rule-based program and uses this information for explaining how a certain conclusion is reached, as well as identifying consequences of a certain fact produced by the program. DBNotes has functionalities similar to Explain, where such additional information is a special form of *annotations* that can be propagated through a special query language. Mondrian extends DBNotes to allow annotations on sets of values.

In [23], the authors propose a concept of provenance at the level of schema mappings for data exchange. The underlying data exchange transformation engine is reengineered so that additional information about which source schema elements and mappings contributed to the creation of a target data is propagated along and stored with the target data. In particular, it modifies the way queries are generated in Clio in order to capture additional information during the exchange. This information can later be queried using a special query language called MXQL. Our debugger is similar to [23] in that it operates over relational or XML schema mappings. However, our approach is different from [23] in two aspects. First, our debugger can be used “as is” on data exchange systems based on similar formalisms for schema mappings. It does not require changes to the underlying engine. Second, we can *automatically* compute routes for any source or target data selected by a user and these routes contain information about schema-level, as well as data-level provenance. In contrast, the approach of [23] requires a user to be familiar with MXQL to query about schema-level provenance, while data-level provenance is not considered.

We emphasize that it is our design choice not to adopt the eager approach for computing routes, since this approach may involve re-engineering the underlying system. We want our debugger to *readily* work on top of Clio or (future) data exchange systems or data integration systems that are based on a similar formalism for schema mappings. It is worth commenting that there is existing research on both the lazy [6, 7] and eager [5, 10] approaches to computing provenance when the transformation is described as an SQL query. However, when the transformation is described as a schema mapping, only the eager approach for computing provenance has been studied [23]. Our work fills in the missing gap of using a lazy approach for computing the provenance of data when the transformation is specified by a schema mapping.

5.2 Approaches in deductive databases

Our route algorithms bear resemblance to top-down approaches such as the SLD resolution technique [1] for evaluating datalog. In

Schemas	Total elems	Atomic elems	Nest. depth	Inst. size	$ \Sigma_{st} / \Sigma_t $
S:	DBLP ₁ (XML)	65	57	1	640KB
	DBLP ₂ (XML)	20	12	4	850KB
T:	Amalgam ₁ (Rel)	117	100	1	1.1MB
S:	Mondial ₁ (Rel)	157	129	1	1MB
T:	Mondial ₂ (XML)	144	112	4	1.2MB

Table 1: Real datasets and schema mappings characteristics.

fact, sophisticated variants of SLD resolution such as query/subquery (QSQ) [24], rule/goal graphs [22] or OLDT [20] are even more closely related to our work. These approaches use memoization to avoid redundant computations and infinite loops. In our route algorithms, we also avoid redundant computations and infinite loops by not exploring any branches under repeated tuples. Explored branches are never discarded and they are memoized. However, a major difference between our route algorithms and these top-down approaches is that we make use of the target instance, which is available to us. In contrast, the result of a datalog program is not available during resolution. A consequence of this difference is that we may be able to detect if a tuple has no routes early in the computation. The top-down techniques, in contrast, will continue to perform resolution down to the source tuples before deciding whether a tuple belongs to the output of a datalog program executed against a source instance.

Another difference is that our approach is more scalable in general. We are able to exploit the database engine, since we push the computation to the database by using queries in each `findHom` step. In contrast, top-down resolution techniques have to perform nested loop joins in memory, since they expand one subgoal at a time and need to perform sideways information passing to propagate the new assignments to the unexplored subgoals.

6. CONCLUSION AND FUTURE WORK

We have presented two route algorithms for computing all routes or one route for selected target data. The former returns a compact, polynomial-size representation of all minimal routes, even though there may be exponentially many minimal routes for the selected data. The latter avoids the computation of all routes in general by producing one route fast, if there is one, and alternative routes as needed. We are currently developing a visual interface for our debugger and we refer the interested reader to [2] for more details. An interesting future extension for our debugger is to adapt the changes made to the target instance dynamically along with the changes to the schema mapping made by the user. Our concept of a route currently does not reflect how an `egd` is used in an exchange. We would like to explore definitions and algorithms for computing routes that take into account `egds`. Other questions include whether there is an efficient way of generating all minimal routes in a concise manner, optimization opportunities in `findHom`, as well as a systematic study on how our debugger provides developmental support to designers of schema mappings.

Acknowledgements. We thank Mauricio Hernández, Howard Ho, Haifeng Jiang, and Lucian Popa for helpful discussions. We also thank Phil Bernstein for numerous helpful suggestions.

7. REFERENCES

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.
- [2] B. Alexe, L. Chiticariu, and W. Tan. SPIDER: a Schema mapPIng DEbugger. In *VLDB Demonstration (To appear)*, 2006.
- [3] T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, and D. Srivastava. Explaining program execution in deductive systems. In *DOOD*, pages 101–119, 1993.
- [4] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, pages 209–220, 2003.
- [5] D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB*, pages 900–911, 2004.
- [6] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, pages 316–330, 2001.
- [7] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *TODS*, 25(2):179–227, 2000.
- [8] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.
- [9] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W. Tan. Peer data exchange. In *PODS*, pages 160–171, 2005.
- [10] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, page 82, 2006.
- [11] L. M. Haas, M. A. Hernandez, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, pages 805–810, 2005.
- [12] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.
- [13] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.
- [14] Altova MapForce. <http://www.altova.com>.
- [15] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Supporting executable mappings in model management. In *SIGMOD*, pages 167–178, 2005.
- [16] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam schema and data integration test suite. www.cs.toronto.edu/miller/amalgam, 2001.
- [17] The Mondial database. <http://www.dbis.informatik.uni-goettingen.de/Mondial/>.
- [18] L. Popa, Y. Velegarakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.
- [19] Stylus Studio. <http://www.stylusstudio.com>.
- [20] H. Tamaki and T. Sato. Old resolution with tabulation. In *ICLP*, pages 84–98, 1986.
- [21] TPC Transaction Processing Performance Council. <http://tpc.org>.
- [22] J. Ullman. Implementation of logical query languages for databases. In *TODS*, pages 289–321, 1985.
- [23] Y. Velegarakis, R. J. Miller, and J. Mylopoulos. Representing and querying data transformations. In *ICDE*, pages 81–92, 2005.
- [24] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *EDS*, pages 179–193, 1986.
- [25] L. Yan, R. Miller, L. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *SIGMOD*, pages 485–496, 2001.
- [26] C. Yu and L. Popa. Constraint-based xml query rewriting for data integration. In *SIGMOD*, pages 371–382, 2004.