TRAMP: Understanding the Behavior of Schema Mappings through Provenance

Boris Glavic
University of Zurich
glavic@ifi.uzh.ch

Gustavo Alonso
ETH Zurich
alonso@inf.ethz.ch

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

Laura M. Haas
IBM Almaden Research Center
laura@almaden.ibm.com

ABSTRACT

Though partially automated, developing schema mappings remains a complex and potentially error-prone task. In this paper, we present *TRAMP* (TRAnsformation Mapping Provenance), an extensive suite of tools supporting the debugging and tracing of schema mappings and transformation queries. TRAMP combines and extends *data* provenance with two novel notions, *transformation* provenance and *mapping* provenance, to explain the relationship between transformed data and those transformations and mappings that produced that data. In addition we provide query support for transformations, data, and all forms of provenance. We formally define transformation and mapping provenance, present an efficient implementation of both forms of provenance, and evaluate the resulting system through extensive experiments.

1. INTRODUCTION

Schema mappings, declarative constraints that model relationships between schemas, are the main enabler of data integration and data exchange. They are used to translate queries over a *target schema* into queries over a *source schema* (data integration) or to generate executable *transformations* that produce a target instance from a source instance (data exchange). These *transformations* are generated from the logical specification of the *mappings*.

Schema mappings are often generated semi-automatically using tools like *Clio* [20, 10], BEA AquaLogic [5], and others [4]. The generation of executable transformations from mappings is also largely automated. The complexity of large schemas, lack of schema documentation, and the iterative, semi-automatic process of mapping and transformation generation are common sources of errors. These issues are compounded by limitations and idiosyncrasies of mapping tools (which can produce wildly different transformations for the same input schemas [4]). Understanding and debugging an integration, its mappings, and transformations is far from trivial and is often a time consuming, expensive procedure. In addition, schema mapping is often done over data sources that are themselves dirty or inconsistent. Errors caused by faulty data cannot be neatly and cleanly separated from errors caused by an incorrect mapping or transformation.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

As a result, a number of research efforts have emerged to support users in debugging and understanding schema mappings and mapping alternatives (these include SPIDER[8] and MXL[25], along with mapping understanding-by-example systems such as the Clio data-viewer [26] and MUSE [3]). Such systems focus on fixing or refining the mapping specification. In contrast, *TRAMP* (TRansformation MApping Provenance), the system we present in this paper, is a more holistic approach that aims at providing a robust data integration debugging tool for tracing errors, no matter what their source (the data, inconsistencies between data sources, the schemas, schema constraints, the mappings, or the transformations). We argue that a robust tool for understanding the behaviour of complex schema mappings has to make all elements of a mapping scenario (schemas, schema constraints, mappings, transformations, and data) and their inter-relationships query-able. Building such a system involves non-trivial challenges from a conceptual and a system design point of view: **(1)** integrate this functionality in a single system that exploits a common approach to tracing errors - rather than providing a loose collection of tools, and **(2)** provide a uniform query interface for all elements of a mapping scenario and their inter-relationships. The main contributions of the paper are:

•Two new types of provenance information, *transformation*-provenance and *mapping*-provenance.

•A query facility for mapping scenario information and provenance; with a focus on querying of mapping and transformation provenance. This facility is fully integrated in SQL, thus, access to all types of data, provenance, and the transformations themselves can be combined in a single query.

•A full implementation and evaluation of TRAMP. TRAMP efficiently computes provenance for relational data, mappings specified as source-to-target tuple-generating-dependencies, and transformations implemented in SQL.

The paper is organized as follows. In Section 2, we introduce schema mappings and provenance background. Afterwards, we illustrate common errors and the provenance and query functionality needed to understand these errors (Section 3). In Section 4, we formally introduce several notions of provenance. Building upon this theoretical treatment, we discuss the implementation of these ideas in the TRAMP system (Section 5). Section 6 presents performance measurements. We discuss a debugging example in Section 7, review related work in Section 8, and conclude in Section 9.

2. BACKGROUND AND NOTATION

In this section, we introduce the background and notation for the rest of the paper. Commonly, a schema mapping is modeled as a tuple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_s, \Sigma_t)$ where $\mathbf{S}$ is a *source* schema, $\mathbf{T}$ is a *target* schema, and Σ_s (and Σ_t) are sets of *constraints* over $\mathbf{S}$ (resp. $\mathbf{T}$) [11, 17]. The constraints Σ_{st} are the mapping(s) rep-

resenting the relationship between **S** and **T**. In our implementation, the source and target constraints (Σ_s and Σ_t) may be any SQL constraint, including primary keys, unique constraints and foreign keys. Our framework and definitions are general enough to include any constraint, including the commonly studied *tuple-* and *equality-generating dependencies* [1]. Our implementation assumes the mapping, Σ_{st}, is a set of source-to-target *tuple-generating dependencies* (*s-t tgds*): $\forall \mathbf{x}\phi(\mathbf{x}) \Rightarrow \exists \mathbf{y}\psi(\mathbf{x},\mathbf{y})$ where $\phi(\mathbf{x})$ is an SPJ query over **S** and $\psi(\mathbf{x},\mathbf{y})$ is an SPJ query over **T**. Schema mappings of this form are quite general and are the basis for most work in data exchange and data integration.

For a given instance $\mathcal{I}$ of the source schema, an instance $\mathcal{J}$ of the target schema is called a *solution* of a schema mapping $\mathcal{M}$ if $\mathcal{J}$ satisfies all the constraints in Σ_t and $\mathcal{I}, \mathcal{J}$ satisfy all the constraints in Σ_{st}. In general, there can be many such solutions for a given schema mapping. One of these solutions will be produced through *transformations* that implement the schema mapping. A single transformation may implement several mappings, or a single mapping may be implemented by several transformations. In addition to the many possible sources of errors, the tools used in data exchange or data integration that are based on schema mappings often generate *transformations* in a way that leads to unexpected results, because *ad hoc* choices are made when several solutions are possible (in data exchange systems) or different strategies for query rewriting are used (in data integration systems).

To capture all the aspects that influence the result of a schema mapping, we model a *mapping scenario* by explicitly including the source and target instance, and the transformations $\mathcal{T}$ that implement the schema mappings Σ_{st} along with the relationship between them. In this paper, we focus on transformations implemented in SQL, but the ideas should be generalizable to other implementation languages such as XSLT or Java.

Several notions of provenance have been introduced in the literature to describe the relationship between the input and output data of a transformation. The most common semantics is to define the provenance of an output data item of a transformation to contain all the input data items that *contributed* to its existence. Therefore, we call such provenance *data* provenance. The different types of *data* provenance differ in how provenance information is represented and their definition of the term *contribution* (called *contribution semantics* or *CS*) [14]. Examples are *Why*-provenance, aka *influence-CS* (*I-CS*), and *Where*-provenance, aka *copy-CS* (*C-CS*) [7]. *Data* provenance relates output and input data, but does not provide any information about which parts of a transformation were used to derive an output tuple. E.g., consider a transformation that uses the SQL *union all* operator. Each output tuple of the union is produced by exactly one of the queries that are input to the union. We refer to this type of provenance as *transformation* provenance. *Transformation* provenance is similar in motivation to *how*-provenance [15], but unlike how-provenance, it is operator-centric, describing the contribution of each operator in a transformation.[1] As mentioned above, in a mapping scenario each transformation implements one or more mappings and vice versa. To be able to understand such a mapping scenario it is crucial to know which parts of a transformation correspond to which mapping(s) (as mentioned before we include this information in the mapping scenario) and to be able to use this information in conjunction with *transformation* provenance to determine which mappings were used to derive some output data. We call this type of information *mapping provenance*. We model *transformation* and *mapping* provenance as annotated parts of the structure of a transformation. To be able to benefit from such data it is necessary to be able to ask queries that access the structure and/or runtime properties of queries. This type of functionality is often referred to as *Meta-querying* [24].

[1]We discuss the relationship between how- and transformation-provenance in more detail in Section 8.

3. ERROR TYPES

We now present common types of errors and discuss what information (especially provenance) can be used to trace them. We distinguish between three error categories based on the origin of an error: *Mapping errors*, *Transformation errors*, and *Instance Errors*.

3.1 Mapping Errors

Missing Mappings: A mapping missing from a mapping scenario may lead to empty target relations or incomplete target relations. Missing mappings may be caused by missing correspondences (matches), missing schema constraints, and misinterpreting the semantics of mappings and schemas. Correspondences can be missed through errors by a matching tool [21]. **Incomplete Mappings**: Incomplete mappings (i.e., mappings that are missing relations or missing conditions) may also arise due to missing correspondences or missing schema constraints. **Oversized Mappings**: Oversized mappings (that is, mappings with too many relations) may be caused by using an association in the source with different semantics than the corresponding association in the target. Thus, an oversized mapping associates relations that should not be associated according to the desired mapping semantics. **Incorrect Association Paths**: Schema constraints, query logs, or even connections in the data may be used by mapping systems to figure out how to join source or target relations in a mapping. In a schema there might be several ways to reach one relation from another via connecting constraints, but not all of them represent a semantically correct way of associating relations.

3.2 Transformation Errors

The most commonly used mapping languages today (including the s-t tgds we consider) are constraints that specify properties that must be true of the transformation. However, they do not determine a unique target instance, and hence do not determine a unique transformation. New types of errors arise due to the fact that a mapping can be handled in different ways by the transformations that implement the mapping. **Incorrect Handling of Atomic Values**: This type of error arises if either a non-atomic attribute is handled as an atomic one or if incorrect functions are applied to split a non-atomic attribute value. An example of this type of error is a transformation that literally copies the values of a source attribute *name* that stores the full names of persons to a *firstname* attribute in the target. **Redundant Data**: A transformation that implements multiple, overlapping mappings may produce redundant data for a target relation. This was one of the motivations to introduce *nested mappings* [12], *core mappings* [18], and *laconic mappings* [23]. However, such mappings are not considered by most mapping tools.

3.3 Instance Errors

Instance Data Errors: Incorrect source instance data can cause errors in the target instance. Instance data errors may confuse a user and lead her to conclude a correct mapping is incorrect.

3.4 How to Trace Errors

Data provenance helps in tracing erroneous target data back to erroneous source data and in understanding mapping errors that are caused by mapping data from the wrong sources. It can also be used to trace *Incorrect Handling of Atomic Values* errors and *Incomplete Mappings* by examining to which source relations the provenance

of a result belongs. For *Oversized Mappings* and *Incorrect Association Paths*, *data* provenance can be used to understand where the incorrect data is coming from. *Transformation* provenance is extremely useful in understanding how data is integrated through a mapping, because it allows us to understand which parts of a transformation (that is, which operators in the transformation) produced a data item. Naturally, *transformation* provenance is very useful to debug *transformation* errors (*Incorrect Handling of Atomic Values* and *Redundant Data*). Also *Oversized Mappings* and *Incorrect Association Paths* errors can be seen in the transformation that implements these mappings and can therefore be debugged using *transformation* provenance. *Mapping* provenance can be used to understand which of the mappings implemented by a transformation produced erroneous data, thus, limiting the scope of the error tracing process and can be used to trace the same types of errors as *transformation* provenance.

Meta-querying can be used to trace *mapping errors* if they are understandable without data and run-time information. In contrast, *transformation* errors are often dependent on run-time information and are therefore normally not trace-able by using solely *meta-querying*. Due to the fact that SQL is not well suited for accessing the hierarchical structure of transformations, *meta-querying* is needed to drill down into *transformation* provenance information. In principle we follow the approach for *meta-querying* presented by Van den Bussche et al. [24]. In their work, queries are represented as XML data, because the hierarchical structure of a query is reflected well in an XML representation. We provide a function $f_{SQL \rightarrow XML}$ that transforms an SQL query text into its XML representation (similar to the one presented by Van den Bussche et al. [24]). Meta-querying functionality is realized by using the built-in XPath and XSLT functionality of the underlying database system. For convenience, we provide predefined XSLT-functions that may be helpful in investigating transformation and mapping provenance (see Appendix C and Glavic [13] for examples).

4. PROVENANCE FOR MAPPINGS

In this section, we formally define *data* provenance and the novel concepts of *transformation* and *mapping* provenance for the relational data model.

4.1 Algebra

We briefly introduce the relational algebra that we use to express queries. Relations are modeled as bags of tuples, thus, every tuple t in a relation R has a multiplicity m (t^m). We use t as a shorthand for t^1. The algebra is defined for bag-semantics, because SQL uses bag-semantics and the transformations implementing a mapping are implemented in SQL. We use q with subscripts to denote algebra expressions (queries) and Q to denote the relation that results from evaluating q. The schema of a relation R (or query result Q) is denoted by $\mathbf{R}$ (respectively, $\mathbf{Q}$). The operators of the algebra are standard relational algebra operators: selection σ_C on a condition C, projection Π_A on a list of expressions A over attributes, functions, constants, and renaming (denoted as $a \rightarrow b$), join operators, and set operations. Join operations include inner join ($\bowtie$) and outer joins (⟕, ⟖, ⟗). Aggregation $\alpha_{G,agg}$ groups its input on a list of group by attributes G and computes the aggregate functions in the list agg for each group. We use ε to refer to a *null*-value. The algebra includes the standard set operators union ($\cup$), intersection ($\cap$), and set difference ($-$).

4.2 Data Provenance

Data provenance describes the relationship between a result of a transformation and the inputs that contributed to it. In a relational setting, this is usually interpreted as the input tuples of a query q that contributed to an output tuple t of q. In *TRAMP* we use *PI-CS* (*Perm Influence Contribution Semantics*), a modified version of the *contribution semantics* [9] (which we will refer to as *Lineage-CS*). In contrast to *Lineage-CS*, *PI-CS* produces more precise provenance for outer joins and union, and uses a different representation of provenance information.

Lineage-CS models the provenance of a result tuple t of a query q as a list $\mathcal{W}(q,t) =< Q_1^*, \ldots, Q_n^* >$ of subsets Q_i^* of the inputs Q_i of the query (where the inputs could be base relations or the result of other queries) that *contribute* to t. Modeling provenance as independent sets of tuples has the disadvantage that the information about which input tuples were combined to produce a result tuple is not modeled. For instance, consider a query $q = \Pi_a(R \bowtie_{a=b} S)$. An output tuple t may be produced from several outputs of the join that are all projected on t. *Lineage-CS* would represent the provenance of t as two subsets of R and S containing the tuples from R and S that were joined by the query and projected on t. Which tuples contributed to t is apparent from this representation, but the information about which tuple from R was joined with which tuple from S is not modeled. To explicitly model which tuples were used together in the creation of an output tuple, we changed the provenance representation from a list of subsets of the input relations to a set of *witness lists*. A witness list w is an element from $(Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ with $Q_i^\varepsilon = Q_i \cup \perp$. Thus, a witness list w contains a tuple from each input of an operator or the special value $\perp$. The value $\perp$ indicates that no tuple from an input relation belongs to the witness list w (and, therefore, is useful in modeling outer joins and unions which are both important in integration). Each witness list represents one combination of input relation tuples that were used together to derive a tuple. We now present a declarative definition of *PI-CS* stating the conditions a set of witness lists has to fulfill to be the provenance of a tuple t.

DEFINITION 1 (PI-CS PROVENANCE). *For an algebra operator op with inputs $Q_1, \ldots, Q_n$, and a tuple $t \in op(Q_1, \ldots, Q_n)$ a set $\mathcal{P}(op,t) \subseteq \mathcal{W} = (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ where $Q_i^\varepsilon = Q_i \cup \perp$ is the* PI-CS provenance *of t if it fulfills the following conditions:*

$$op(\mathcal{P}(op,t)) = \{t\} \quad \textbf{(1)}$$

$$\forall w \in \mathcal{P}(op,t) : op(w) \neq \emptyset \quad \textbf{(2)}$$

$$\neg \exists \mathcal{P}' \subseteq \mathcal{W} : \mathcal{P}' \supset \mathcal{P}(op,t) \wedge \mathcal{P}' \models (1),(2),(4) \quad \textbf{(3)}$$

$$\forall w, w' \in \mathcal{P}(op,t) : w \prec w' \Rightarrow w \notin \mathcal{P}(op,t) \quad \textbf{(4)}$$

The first condition **(1)** in Definition 1 checks that the provenance produces exactly t and nothing else by computing the result of the operator op over the provenance. The second condition **(2)** guarantees that each witness list w (combination of tuples) in $\mathcal{P}$ contributes something to t (removal of false positives). The third condition **(3)** checks that $\mathcal{P}$ is the maximal set with these properties, meaning that no witness lists that contribute to t are left out. The provenance of a query is computed by recursively applying Definition 1 to each operator of the query (for the exact transitivity definition see Glavic [13]). An advantage of *PI-CS* is that the focus on a single operator leads to a manageable evaluation strategy and the provenance of each operator can be studied independent of the provenance of other operators. Conditions 1 to 3 are the original *Lineage-CS* conditions adapted to the witness list representation. The fourth condition **(4)** is necessary to produce precise provenance for outer joins and set union. This condition states that we will exclude a witness list w from the provenance, if there is a "smaller" witness list w' in the provenance that subsumes w. A witness list w is subsumed by a witness list w' (denoted by $w \prec w'$) iff w' can be derived from w by replacing some input tuples from w with $\perp$.

$$\mathcal{P}(\sigma_C(q_1), t) = \{< t >\}$$
$$\mathcal{P}(\alpha_{G,agg}(q_1), t) = \{< t' > |\ t' \in Q_1 \wedge t.G = t'.G\}$$
$$\mathcal{P}(q_1 \bowtie_C q_2, t) = \{< t.\mathbf{Q_1}, t.\mathbf{Q_2} >\}$$

1(a) PI-CS

R

a	b
1	2
1	3
2	3
2	5

S

c
2
3

Q_a

a
1
2

Q_b

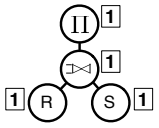
a	b	c
1	2	2
1	3	3
2	3	3
2	5	ε

Q_c

a
1
2

$$q_a = \Pi_a(R \bowtie_{b=c} S)$$
$$\mathcal{P}(q_a, (1)) = \{< (1,2), (2) >, < (1,3), (3) >\}$$
$$q_b = R \overrightarrow{\bowtie}_{b=c} S$$
$$\mathcal{P}(q_b, (2,5,\varepsilon)) = \{< (2,5), \bot >\}$$
$$q_c = \Pi_a(q_b)$$
$$\mathcal{P}(q_c, (2)) = \{< (2,3), (3) >, < (2,5), \bot >\}$$

1(b) Data Provenance $\mathcal{P}$



$$\mathcal{T}(q_c, (2)) : \theta_{<(2,3),(3)>}(op) = 1$$
$$\theta_{<(2,5),\bot>}(op) = \begin{cases} 0 \text{ if } op = \mathbf{S} \\ 1 \text{ else} \end{cases}$$

1(c) Transformation Provenance $\mathcal{T}$

Figure 1: Provenance Examples

As an example of the problems of *Lineage-CS* with operators that use some form of non-existence check, consider q_b from Fig. 1(b). According to *Lineage-CS*, the provenance of the result tuple $t = (2, 5, \varepsilon)$ would contain all tuples from relation S, but in fact none of them contributed to t. We believe a better semantics for the provenance of tuple t would be a witness list $< (2, 5), \bot >$. This indicates that $(2, 5)$ paired with no tuples from S influences t (rather than saying *every* tuple of S is in the provenance of this tuple). In addition to achieving this semantics, condition **(4)** changes the semantics for the $\cup$ operator from that of *Lineage-CS*. Without condition **(4)**, there would be a single witness list $< t_1, t_2 >$ if a result t of a union is generated from two tuples t_1 (left input) and t_2 (right input). We feel this is a bit misleading as it indicates that these two tuples *used together* influence t, when in fact each, independently, influences t. *PI-CS* provenance captures this intuition by defining the provenance as $\{< t_1, \bot >, < \bot, t_2 >\}$.

Fig. 1(a) presents the *PI-CS* provenance of some operators according to Definition 1. For example, the provenance of the output t of a selection is the single witness list containing t, because selection outputs unmodified input tuples. An output tuple t from an aggregation is derived from a set of input tuples that belong to the same group (have the same grouping attribute values as t). Fig. 1(b) shows the provenance for three queries (q_a, q_b, and q_c).

4.3 Transformation Provenance

For *transformation* provenance, we model which parts of a transformation contribute to an output tuple. In contrast to *data* provenance, transformation provenance is *operator-centric* rather than *data-centric*. In motivation, it is similar to provenance approaches for workflow-management systems, that traditionally have focused more on transformations [22]. We model the transformation provenance of a transformation q using an annotated algebra tree for q. For an output tuple t and a witness list w in the data provenance of t, the transformation provenance will include 1 and 0 annotations on the operators of the transformation q. A 1 indicates this operator on w influences t, a 0 indicates it does not.

DEFINITION 2 (ANNOTATED ALGEBRA TREE). *An annotated algebra tree for a transformation q is a pair $(Tree_q, \theta)$ where $Tree_q = (V, E)$ is a tree that contains a node for each algebra operator used in q (including the base relations as leaves) and $\theta : V \in Tree_q \rightarrow \{0, 1\}$ is a function that associates each operator in the tree with an annotation from $\{0, 1\}$. We define a preorder on the nodes to give each node an identifier (and to order the children of binary operators). Let $I(op)$ denote the identifier of the node representing operator op.*

We now have the necessary preliminaries to formally define transformation provenance based on *data* provenance. Intuitively, each witness list of the *data* provenance of a tuple t represents one evaluation of an algebra expression q. For each witness list w, each part of the algebra expression has either contributed to the result of evaluating q on w or not. Therefore, we represent the transformation provenance as a set of annotated algebra trees of q with one member per witness list w. We use *data* provenance to decide whether an operator op in q should get a 0 or a 1 annotation. Basically, if evaluating the subtree sub_{op} under op on w results in the empty set ($sub_{op}(w) = \emptyset$), then op has contributed nothing to the result tuple t and should not be included in the transformation provenance.

DEFINITION 3 (TRANSFORMATION PROVENANCE). *The transformation provenance of an output tuple t of q is a set $\mathcal{T}(q, t)$ of annotated-trees defined as follows:*

$$\mathcal{T}(q, t) = \{(Tree_q, \theta_w) \mid w \in \mathcal{P}(t)\}$$
$$\theta_w(op) = \begin{cases} 0 \textit{ if } sub_{op}(w) = \emptyset \\ 1 \textit{ else} \end{cases}$$

Fig. 1 shows the *data* (b) and *transformation* (c) provenance for a result tuple (2) of query q_c. The output tuple (2) has two witness lists. The transformation provenance of (2) for the first witness list is a tree with every node annotated by a 1 (shown on the left of Fig. 1(c)). For the second witness list the node for the base relation S carries a 0 annotation, because it does not contribute to the result.

4.4 Mapping Provenance

In a mapping scenario, transformations may be derived from a set of declarative schema mappings. For most non-trivial mappings several transformations exists that implement the mappings correctly (they produce a target instance that satisfies Σ_{st} and Σ_t). A single transformation may implement more than one mapping or vice versa. For debugging we would like to know not only what parts of a transformation produced a target tuple t, but also from what mappings these transformations (or operators within a transformation) were derived. Therefore, we define mapping provenance based on transformation provenance and the relationship between transformations and mappings. The relationship between a mapping and part of a transformation is modeled by adding additional annotations (specifically, mapping identifiers) to the algebra tree for a transformation. For an algebra tree, $Tree_q = (V, E)$, we introduce one new annotation function, μ_M, per mapping $M \in \Sigma_{st}$. The function μ_M is 1 for each operator that implements this mapping and 0 otherwise. For example, consider the mappings $\mathbf{M_1} : S(a) \Rightarrow \exists b : T(a, b)$ and $\mathbf{M_2} : S(a) \wedge R(a, b) \Rightarrow T(a, b)$

$q_d = \Pi^{1}{}_{c\to d, b\to e}(S^{3} \overrightarrow{\bowtie}^{2}{}_{c=a} R^{4})$

$\mathbf{Q_d}$

d	e
2	3
2	5
3	ε

$\mathcal{M}(q_d,(2,3)) = \{\{M_2\}\}$
$\mathcal{M}(q_d,(2,5)) = \{\{M_2\}\}$
$\mathcal{M}(q_d,(3,\varepsilon)) = \{\{M_1\}\}$

$\mu_{M_1}(op) = \begin{cases} 1 & I(op) \in \{1,2,3\} \\ 0 & I(op) \in \{4\} \end{cases}$

$\mu_{M_2}(op) = 1$

Figure 2: Example Mapping Provenance

for target relation T with schema $\mathbf{T} = (d, e)$ and R and S as presented in Fig. 1. One possible transformation implementing these mappings is presented in Fig. 2 (q_d). We use superscripts (red) to denote the identifier $I(op)$ of an operator op. The access to base relation S (operator 3) implements both mappings $\mathbf{M_1}$ and $\mathbf{M_2}$. Tuples from S may flow through every operator above operator 3 in the tree so μ_{M_1} and μ_{M_2} annotate every operator on the path from 3 (denoting access to S) to the root with annotation 1. Only μ_{M_2} assigns a 1 to the node for R (operator 4), because it implements only $\mathbf{M_2}$.

Notice that the mapping annotation function will depend on the language used for mappings. We have implemented mapping annotation functions for source-to-target tgds, but of course this could be extended to other languages, including the visual mapping languages of some commercial tools. Below we formalize *mapping* provenance using the annotation functions μ_M.

DEFINITION 4 (MAPPING PROVENANCE). *The mapping provenance $\mathcal{M}(q,t)$ for a tuple t from the result of query q is defined using the mapping annotation functions μ_M over the transformation provenance $\mathcal{T}(q,t)$ as follows:*

$$\mathcal{M}(q,t) = \{\mathcal{M}_w \mid w \in \mathcal{P}(q,t)\}$$
$$\mathcal{M}_w = \{M \mid \forall op \in V : \mu_M(op) = \theta_w(op)\}$$

As an example of *mapping* provenance consider Fig. 2, which presents the result of applying q_d to the source instance from Fig. 1 and the *mapping* provenance for each result tuple of q_d.

5. IMPLEMENTATION

We have implemented *TRAMP* as an extension to the *Perm* relational provenance management system [13, 14]. We chose Perm, because it supports *PI-CS* for data provenance. Furthermore, it uses a native SQL implementation of provenance (using query rewrite) which other available provenance systems such as Trio do not [2]. Implementing provenance computation as query rewrite has the advantage that it allows for seamless integration of on-demand provenance computation in a DBMS and enables SQL query functionality for provenance, while benefiting from the advanced optimization techniques implemented by standard DBMS. We first present a short overview of how *PI-CS data* provenance is implemented in *Perm*. Afterwards we discuss how *transformation* and *mapping* provenance support is added to the system.

5.1 Data Provenance

The *data* provenance in *TRAMP* is based on an extension of the facilities available in *Perm*. Given a relational algebra expression q for which the *data* provenance should be computed, an expression q^+ is generated that computes the provenance of each result tuple of q and returns the result tuples extended with provenance information. Each tuple in Q^+ contains an original result tuple t and one of its witness lists w ((t,w)). A tuple t is duplicated if it has more than one witness list. The provenance computation in q^+ is realized by propagating witness lists from the input to the output of the query. The query q^+ is generated from q by applying a set of algebraic rewrite rules. Each of these rewrite rules operates on a single operator of q. These rules were designed in such a way that an arbitrary algebra statement can be rewritten by recursively applying the rewrite rules for each operator of the statement. The current version of *Perm* supports *PI-CS* and a form of *C-CS* (*Where*-provenance).

$$(R)^T = \Pi_{\mathbf{R},\{I(R)\}\to\mathcal{T}}(R) \quad \text{(T1)}$$
$$(\sigma_C)^T(q) = \Pi_{\mathbf{Q},(\mathcal{T}\cup\{I(\sigma_C)\})\to\mathcal{T}}(\sigma_C(q^T)) \quad \text{(T2)}$$
$$(\Pi_A)^T(q) = \Pi_{A,(\mathcal{T}\cup\{I(\Pi_A)\})\to\mathcal{T}}(q^T) \quad \text{(T3)}$$
$$(q_1 \bowtie_C q_2)^T = \Pi_{\mathbf{Q_1},\mathbf{Q_2},(\mathbf{Q_1^T}.\mathcal{T}\cup\mathbf{Q_2^T}.\mathcal{T}\cup\{I(q_1\bowtie_C q_2)\})\to\mathcal{T}}(q_1^T \bowtie_C q_2^T) \quad \text{(T4)}$$
$$(q_1 \overrightarrow{\bowtie}_C q_2)^T = \Pi_{\mathbf{Q_1},\mathbf{Q_2},(\mathbf{Q_1^T}.\mathcal{T}\cup\mathbf{Q_2^T}.\mathcal{T}\cup\{I(q_1\overrightarrow{\bowtie}_C q_2)\})\to\mathcal{T}}(q_1^T \overrightarrow{\bowtie}_C q_2^T) \quad \text{(T5)}$$
$$(q_1 \cup q_2)^T = \Pi_{\mathbf{Q_1},(\mathbf{Q_1^T}.\mathcal{T}\cup\mathbf{Q_2^T}.\mathcal{T}\cup\{I(q_1\cup q_2)\})\to\mathcal{T}}(q_1^T \cup q_2^T) \quad \text{(T6)}$$

Figure 3: Transformation Provenance Rewrite Rules

Perm is realized as an extension of PostgreSQL. The query rewrites are implemented in a module placed between the parser and optimizer of PostgreSQL. Thus, *Perm* operates on the analyzed internal query tree representation of SQL statements used by PostgreSQL. SQL is extended with several new keywords that trigger and control provenance computation. For instance, the use of the keyword *PROVENANCE* marks a query for data provenance computation. A canonical translation between algebra expressions and query trees is used to be able to implement the algebra rewrite rules on the internal query tree representation of *PostgreSQL* [14].

5.2 Transformation Provenance

We implement *transformation* provenance in *TRAMP* as an additional set of query rewrite rules. These rules transform a query q into a query q^T that will compute and output the transformation provenance of q. As for *data* provenance the result of q^T contains the query result tuples enhanced with provenance information. To be more specific, the schema $\mathbf{Q}$ of q is extended with an additional attribute $\mathcal{T}$ that is used to store transformation provenance information. In Section 4, we modeled the *transformation* provenance of a result tuple t as a set of annotated algebra trees (one tree per witness list w). Note that each element of this set represents the same algebra tree with different annotation functions θ_w. Therefore, we factor out the static part (that is the tree) and use the provenance attribute $\mathcal{T}$ to store only the annotation functions. Each value of $\mathcal{T}$ stores the θ_w for one witness-list w of t (represented as the set of operators that have a 1-annotation). As for *data* provenance, result tuples are duplicated if necessary to represent their complete provenance. We internally represent an annotation set as a bit-vector, because its space requirements are low, and the union operation used frequently in the rewrite rules is efficient (bit-wise disjunction).

Each *transformation* provenance rewrite rule computes a new set of annotations from the annotation sets of the rewritten inputs of an operator. Fig. 3 presents the rewrite rules for the most common algebra operators (see Glavic [13] for the other outer join types and set operations). The rewrite rule **T1** for a base relation access adds the singleton annotation set for the operator $\{I(R)\}$ as the value

for attribute $\mathcal{T}$ to its result relation. A selection is rewritten by rule **T2** by applying the unmodified selection, but in q^T the identifier of the selection is added to the annotation set. **T3**, the rewrite rule for projection, works analogously. The rewrite rule for inner and outer join (**T4** and **T5**) union the annotation sets of their rewritten inputs and add the identifier for the join to the result. Note that this is correct behavior for outer joins if we define the union of a set with the null value as $\mathcal{T}_1 \cup \varepsilon = \mathcal{T}_1$.

To provide a useful *transformation* provenance representation to the user and enable *meta-querying* of this type of provenance the bit-vector representation is transformed into either SQL text with markup or XML in the result of a query. Which representation is chosen is specified by the user by issuing the keyword *TRANSSQL* or *TRANSXML* to trigger transformation provenance computation. The translation from the bit-vector to the external representation is implemented as UDFs f_{SQL} and f_{XML} that are applied in the outermost projection of the rewritten query. The SQL representation is the original query except for parts that do not belong to the *transformation* provenance, which are enclosed by `<NOT>` and `</NOT>`. The XML representation is a hierarchical representation of an SQL statement where each clause is modeled as an XML element.

5.2.1 Rewrite Algorithm

The query rewrite for *transformation* provenance computation consists of the following steps:

(1) Statically analyze the query tree to enumerate the operators in the query and attach a bit-vector that represents the singleton set $I(op)$ to each operator *op*.

(2) Apply the *transformation* provenance rewrite rules in a top-down manner to instrument the query to propagate and combine the per operator bit-vectors.

(3) Add an invocation of function f_{SQL} or f_{XML} to the top query block that transforms the bit-vector of the top query block into SQL or XML representations.

For certain operators the *transformation* provenance is independent of the actual data processed by the query. If a subtree of a query contains only operators with static *transformation* provenance we avoid redundant run-time computations by pre-computing the bit-vector for this subtree. See Glavic [13] for details.

5.3 Mapping Provenance

To implement *mapping* provenance, SQL is extended with the ability to annotate parts of a query using the new keyword *ANNOT*. These annotations are used to represent the correspondences between mappings and transformations as defined by the annotation functions μ_M presented in Section 4. If the transformations are generated from a set of s-t tgd mappings, we can automatically annotate the base relations of a transformation according to the mappings.[2] For other mapping languages, a user can specify the appropriate mapping annotations of a transformation or the mapping system can be changed to create these annotations. These annotations are only used for *transformation* provenance computation and are ignored during normal query processing.

To support *mapping* provenance the *transformation* provenance computation is modified to use these annotations in the final representation construction. The mapping annotation function μ_M for a mapping $\mathbf{M}$ is derived from the annotations and represented as a bit-vector that contains a zero at a position i, if $\mu_M(op) = 0$ for the operator *op* with $I(op) = i$, and a one otherwise. Recall that a mapping belongs to the transformation provenance iff $\mu_M(op) = \theta_w(op)$ holds for each *op*. Translated to the bit-vector representations of μ_M and θ_w this is an equality-check on these bit-vectors. In the final result representation, mappings are represented as additional annotations in the representation (e.g., SELECT... FROM <M1>R</M1> ...). Examples for transformation and mapping provenance computations are presented in Appendix B.

[2]The annotations for the other operators in the query are derived by propagating an annotation on a child to its parent.

THEOREM 1. *The algorithms for transformation and mapping provenance based on the algebraic rewrite rules generate complete and correct results, i.e., they correctly compute the transformation provenance (Def. 3) and mapping provenance (Def. 4).*

PROOF. To prove this claim for transformation provenance we define a relational representation of this provenance type based on Def. 3. The equivalence of the relational representation and the result of rewritten queries is proven by induction over the structure of an algebra expression. Details can be found in Glavic [13]. The correctness and completeness for mapping provenance follows from the correctness and completeness of transformation provenance. If the transformation provenance rewrite generates correct annotation sets then evaluating $\mu_M = \theta_w$ over these sets generates mapping provenance satisfying Def. 4. □

6. PERFORMANCE EVALUATION

6.1 Experimental Setup

To evaluate the scalability of *TRAMP*, we conducted a series of performance measurements. All experiments were performed on an Intel 1.66 GHz dual core machine with 1GB of main memory running Mac OS X 10.5.8. The goal of the experiments is to determine the overhead of the *transformation* provenance computation with respect to running standard queries. For the experiments we used the Amalgam benchmark [19]. We used one of the Amalgam schemas as the source schema, generated a new schema as the target schema, and created mappings between these schemas. The target schema was carefully defined for the mapping scenario to include different kinds of mappings that lead to a broad variety of transformations (different relationships between mappings and transformations, different kinds of SQL queries). Instances of the source schema of sizes ranging from 1.000 to 1.000.000 publications were generated for the experiments. The original Amalgam data set contains about 10.000 publications. Larger instances were filled with randomly generated entries. We generated two sets of queries. The first set of queries (*Q1*) contains the transformations for the Amalgam schema mappings scenario discussed above. To evaluate the effect of different types of mappings, especially types not present in *Q1*, on the performance of provenance computation we generated a second set of queries (*Q2*) each implementing a mapping that represents one of the basic scenarios from the *STbenchmark* [4]. The benchmark scenarios had to be adapted to the relational model, because the original benchmark generates XML data. These mappings are also defined over the Amalgam source schema (see Appendix D for a detailed description of the schemas, mappings, and transformations).

6.2 Scalability

For both sets of queries and for all database sizes, we measured the normal query execution time (*norm*), the execution time for transformation provenance computation without generating a representation from the raw bit-vectors (*tprov*), and the execution time including SQL (*tsql*) and XML (*txml*) representation construction. Each query was repeated (10 to 100 times depending on the database size). The average execution time for executing the complete query sets *Q1* and *Q2* are presented in Fig. 4 (a) and (b). In addition we

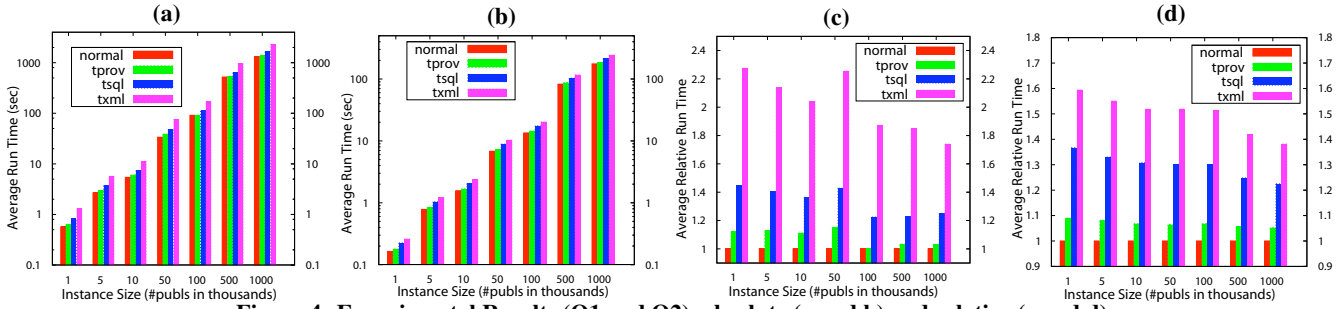


Figure 4: Experimental Results (Q1 and Q2), absolute (a and b) and relative (c and d)

present the relative runtime overhead of the three forms of transformation provenance computation in comparison to the normal query execution times (Fig. 4 (c) and (d)). The first set of queries exhibits a linear runtime for query execution without provenance computation, because the operations applied by these queries are mostly unions and joins on indexed attributes that process complete input tables and mostly each input tuple is only joined with a few other tuples. In comparison with the normal query execution the provenance computation induces a maximal overhead of 13% for *tprov*, 45% for *tsql*, and 128% for *txml*. As expected the SQL and XML representation constructions incur an additional overhead, because they have to be computed in addition to the bit-vector computations and rewrites. For query set *Q2* we measured maximal overheads of 9% for *tprov*, 37% for *tsql* and 60% for *txml*. These results demonstrate that it is feasible to use *TRAMP* for debugging mappings scenarios with large instances as the cost of the basic queries is reasonable even without having explored several potential optimizations. We would like to emphasize that a constant factor overhead for computing the provenance of a transformation which maps a complete source relation is rather impressive if compared to the cost exhibited by the computation of alternative provenance types, e.g., *Why-Not* provenance (see Section 8).

7. DEBUGGING EXAMPLE

To illustrate the type of debugging process supported by *TRAMP*, we present an example debugging session for one of the error types identified in Section 3. Debugging examples for the remaining error types are presented in Appendix A.

7.1 Missing Mappings

Source Schema	Article(Title, Journal)
	InProceedings(Title, Conference)
Target Schema	Publication(Title, Venue)

Mappings

M_1 :$\forall Article(a,b) \Rightarrow Publication(a,b)$

M_2 :$\forall InProceedings(a,b) \Rightarrow Publication(a,b)$

Transformations

$\mathbf{T_1}$: SELECT Title, Journal AS Venue FROM Article;

As mentioned in Section 3, a missing mapping may be caused by a missing correspondence. For instance, if in the example schemas presented above the correspondences between the *InProceedings* relation and the *Publications* relation is missed, then only mapping M_1 (and not M_2) is created. To investigate the cause of the missing conference publications in the target instance, a user may start by querying the mapping and transformation provenance of transformation T_1 that generates the *Publication* relation (mapping M_1):

```
SELECT TRANSPROV * FROM Publication;
```

This query reveals that all tuples have been created by mapping M_1 and that the *InProceedings* relation is not accessed by this mapping and its transformation. This means a mapping is missing that maps data from the *InProceedings* relation. By explicitly modeling the correspondences and making them query-able *TRAMP* enables the user to track the reasons for the missing mapping. E.g., the user can search for correspondences between *InProceedings* and another relation by using XPath to query the XML representation of the correspondences. This would reveal that no correspondences are defined for this relation and new correspondences need to be created to enable a mapping tool to generate mapping M_2.

8. RELATED WORK

The need to support users in understanding the results of schema mapping has been addressed before, although not in as comprehensive and inclusive manner as in TRAMP. The *Clio data-viewer* [26] and *Muse* [3] systems helps a user in understanding a schema mapping by presenting the result of applying a transformation for a schema mapping on small example source instances (examples chosen by the tool). Both approaches do not allow a user to trace the origins of an arbitrary target tuple as data, mapping and transformation provenance do. In addition, they do not directly support understanding of either incorrect transformations (generated from correct mappings) or dirty source data.

SPIDER [8] uses provenance in defining *routes* computed for a subset of a target instance. Each route is a possible way of producing the tuples of interest by sequentially applying mappings to tuples (*route-steps*) in the source instance (and the tuples generated by previous mapping applications in the route). A route combines *data* with *mapping* provenance. SPIDER does not provide any querying facilities over the routes and lacks support for debugging incorrect transformations. *MXQL* [25] generates provenance eagerly during the execution of a transformation. The generated target instance is enriched with transformations that store *mapping* provenance and provenance that relates source to target schema elements (*schema* provenance). In contrast to *TRAMP*, MXQL supports neither *meta-querying* nor *transformation* provenance.

Our notions of *data*, *transformation* and *mapping* provenance are related to past work on data provenance. Since TRAMP is built on top of *Perm*, it supports both *PI-CS* and *C-CS* provenance [14]. *PI-CS* bears some similarities to the *Why*-provenance definition of Cheney et al. [7], but the *PI-CS* representation (witness lists) provides more information (combination of input tuples). *Transformation* provenance has some similarities with *How* provenance [15] and *Why-not* provenance [6]. *How* provenance has the disadvantage that, unlike *transformation* provenance, it does not record any information about which operators of a query contributed to a result. Since many provenance types can be modeled using the semiring framework on which *how* provenance is based on, one obvious

question is if *PI-CS* and *transformation* provenance are also instances of this model. Interestingly, this is not the case. For both types of provenance there exists a database instance of annotated relations (such relations are called *K-relations* [15]), two queries q_1 and q_2, and a tuple t in the result of the queries that carries the same semiring annotation, but has different *transformation* and *PI-CS* provenance (see appendix E). Note that this holds independently of which semiring is used. The major difference between Why-not provenance and our approach is that we compute and present *transformation* provenance for each witness list, whereas *Why-not* provenance is computed for an input pattern and there is only one output for a certain input pattern. Furthermore, no method to query provenance is provided. Finally, to compute *Why-not* provenance, the *data* provenance of several parts of the query have to be computed. Our implementation has the advantage that, though we also define *transformation* provenance based on *data* provenance, we never have to pay the price of instantiating this information in the computation of *transformation* provenance.

Different forms of *mapping* provenance have been implemented in *Orchestra* [15] and *MXQL* [25]. *Orchestra* uses schema mappings to propagate updates between peers with different schemas. *How* provenance is used to store the origin of data in a peer instance. *Mapping* provenance is modeled by adding functions to the *how*-provenance semiring model. The *mapping* provenance provided by *MXQL* represents static (meaning instance independent) information [25]. MXQL uses annotations to associate information about both mappings and source schemas with target data. However, using static information (instead of combining this with runtime information as *TRAMP* does) has the disadvantage that it is not possible to determine exactly which mappings produced a tuple.

9. CONCLUSION

We have presented *TRAMP*, a holistic approach to schema mapping debugging. *TRAMP* integrates several kinds of provenance, including the novel notions of *transformation* and *mapping* provenance, in a single system, and provides efficient and powerful query support for this information. We have shown how only the full combination of all these facilities can answer many questions that are important in understanding and disambiguating the sources of the myriad of errors than can occur when integrating data. We demonstrated by means of an extensive experimental evaluation that this new functionality can be efficiently computed by using bit-set operations and run-time pruning based on static query analysis.

In the future it would be interesting to investigate whether it is possible to develop an approach that explains how to change a transformation to cause the inclusion of some tuples in its result (as the recent *Artemis* system [16] does for data). Furthermore, we could combine our approach with example-based approaches like *Muse* and approaches that operate on the logical specification of mappings like *SPIDER*. Another promising venue for future work is extending the meta-querying facilities of the system and providing a tighter integration with mapping systems.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. An Introduction to ULDBs and the Trio System. *IEEE Data Engineering Bulletin*, 29(1):5–16, 2006.

[3] B. Alexe, L. Chiticariu, R. Miller, and W. Tan. Muse: Mapping Understanding and Design by Example. In *ICDE*, pages 10–19, 2008.

[4] B. Alexe, W. Tan, and Y. Velegrakis. STBenchmark: Towards a Benchmark for Mapping Systems. *PVLDB*, 1(1):230–244, 2008.

[5] M. Blow, V. R. Borkar, M. J. Carey, C. Hillery, A. Kotopoulis, D. Lychagin, R. Preotiuc-Pietro, P. Reveliotis, J. Spiegel, and T. Westmann. Updates in the AquaLogic Data Services Platform. In *ICDE*, pages 1431–1442, 2009.

[6] A. Chapman and H. Jagadish. Why Not? In *SIGMOD*, pages 523–534, 2009.

[7] J. Cheney, L. Chiticariu, and W. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[8] L. Chiticariu and W. Tan. Debugging Schema Mappings with Routes. In *VLDB*, pages 79–90, 2006.

[9] Y. Cui and J. Widom. Lineage Tracing in a Data Warehousing System. In *ICDE*, page 683, 2000.

[10] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. *Clio: Schema Mapping Creation and Data Exchange*. Springer, 2009.

[11] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[12] A. Fuxman, M. Hernandez, H. Ho, R. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *VLDB*, pages 67–78, 2006.

[13] B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010.

[14] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE*, pages 174–185, 2009.

[15] T. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *PODS*, pages 31–40, 2007.

[16] M. Herschel, M. Hernández, and W. Tan. Artemis: A System for Analyzing Missing Answers. In *VLDB*, pages 1550–1553, 2009.

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

[18] G. Mecca, P. Papotti, and S. Raunich. Core schema mappings. In *SIGMOD Conference*, pages 655–668, 2009.

[19] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam Schema and Data Integration Test Suite, 2001. www.cs.toronto.edu/ miller/amalgam.

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

[21] E. Rahm and P. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.

[22] Y. L. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in e-Science. *SIGMOD Rec.*, 34(3):31–36, 2005.

[23] B. ten Cate, L. Chiticariu, P. G. Kolaitis, and W.-C. Tan. Laconic schema mappings: Computing the core with sql queries. *PVLDB*, 2(1):1006–1017, 2009.

[24] J. Van den Bussche, S. Vansummeren, and G. Vossen. Towards Practical Meta-Querying. *Information Systems*, 30(4):317–332, 2005.

[25] Y. Velegrakis, R. Miller, and J. Mylopoulos. Representing and Querying Data Transformations. In *ICDE*, pages 81–92, 2005.

[26] L. Yan, R. Miller, L. Haas, and R. Fagin. Data-driven Understanding and Refinement of Schema Mappings. In *SIGMOD*, pages 485–496, 2001.

APPENDIX

A. EXAMPLES FOR DEBUGGING DIFFERENT ERROR TYPES

To illustrate the usefulness of *TRAMP* for debugging the error types presented in Section 3, we present an example for each of these error types and demonstrate how *TRAMP* is used to debug the errors.

A.1 Incomplete Mapping

Source Schema	Employee(Name,Dep)
Target Schema	WorksAt(Name, DepId) Department(DepId, Name)

Mappings

$$M_1 : Employee(a, b) \Rightarrow \exists c Department(c, b)$$
$$M_2 : Employee(a, b) \Rightarrow \exists c WorksAt(a, c) \wedge Department(c, b)$$

Transformations

$\mathbf{T_1}$: SELECT $SK_1(Name)$ AS DepId, Dep AS Name FROM Employee;

Mapping M_1 presented above is an incomplete version of mapping M_2. Assume the correspondence between the attribute *Dep* in the source and the *Name* attribute from *Department* has been missed. This will result in an empty *WorksAt* relation in the target instance. This error can be debugged by querying the XML representations of mappings (to ascertain that no mapping maps data to the *WorksAt* relation) and correspondences (to check if a correspondence is defined between some source relation and the *WorksAt* relation). These analyses identify the missing correspondence between the *Employee* and *WorksAt* relations. A mapping tool can be used to create the missing correspondence and transform M_1 into the correct mapping M_2.

A.2 Oversized Mappings

Source Schema	Scientist(Name, Affiliation)
Target Schema	Researcher(Name, Affiliation) Institute(Name, Location)

Mappings

$$M_1 : Scientist(a, b) \Rightarrow \exists c Researcher(a, b) \wedge Institute(b, c)$$

Transformations

$\mathbf{T_1}$: SELECT Name, Affiliation FROM Scientist s;
$\mathbf{T_2}$: SELECT Affiliation AS Name, $SK_1(Name)$ FROM Scientist s;

Assume the *Affiliation* attribute in the source schema presented above represents the PhD granting institution, but *Affiliation* in the target is a *works-in* relationship. In this case mapping M_1 is oversized. A correct mapping would only map the name of a scientist. M_1 is implemented by transformations T_1 and T_2. T_1 generates the *Researcher* relation and T_2 generates the *Institute* relation. *Data* provenance can be used to trace the source of this error:

```
SELECT PROVENANCE *
FROM Researcher r, Institute i
WHERE r.Affiliation = i.Name;
```

The query above uses a join between the *Researcher* and *Institute* relations to retrieve the provenance of each researcher and her associated institute. As evident in the result of this query, each *Researcher* tuple and associated *Institute* tuple is derived from a single *Scientist* tuple in the source. So either the source instance data is erroneous or the mapping contains an error. To decide which case applies, the user can state queries over the provenance to retrieve the provenance of researchers for which the affiliation and PhD granting institutions are known. Having realized that M_1 is oversized, the mapping tool should be used to generate a mapping that maps only the *name* attribute of a scientist.

A.3 Incorrect Association Paths

Source Schema	Address(aId, City, Country) Employee(Name, dId, aId) Department(dId, Name, aId)
Target Schema	Person(Name, LivesAt, WorksFor)

Mappings

$$M_1 : Address(a, b, c) \wedge Employee(d, e, f) \wedge Department(e, g, a) \Rightarrow Person(d, a, g)$$
$$M_2 : Address(a, b, c) \wedge Employee(d, e, a) \wedge Department(e, f, g) \Rightarrow Person(d, a, f)$$

Transformations

$\mathbf{T_1}$:
```
SELECT e.Name,
       a.City AS LivesAt,
       d.Name AS WorksFor
FROM Address a, Employee e, Department d
WHERE a.aId = d.aId AND e.dId = d.dId;
```

Assume mapping M_1 was created to map addresses, employees and departments to the *Person* relation in the target schema presented above. If the *LivesAt* attribute of relation *Person* stores the home cities of persons, then the mapping tool (or user) has chosen an *incorrect association path*, because M_1 maps the address associated with an department to the *LivesAt* attribute. Both data provenance and transformation provenance can be used to trace this error. Each person is derived from an address, employee and department. This is revealed by asking the following query:

```
SELECT PROVENANCE * FROM target.Person;
```

That the address of a department instead of a employee is mapped to the target is apparent in the data provenance, because the *aId* attributes of the department and address tuples used to derive a target tuple are the same. If the user does not recognize this equality, (s)he may proceed by retrieving the transformation provenance of the *Person* relation. The transformation provenance clarifies that the department and address have been joined instead of employee and address. Retrieving the definition of mapping M_1 confirms that the mapping is the source of the error. The mapping tool can then be used to replace mapping M_1 by mapping M_2.

A.4 Incorrect Handling of Atomic Values

Source Schema	Employee(FirstName, LastName)
Target Schema	Person(Name, Gender)

Mappings

$$M_1 : Employee(a, b) \Rightarrow \exists c Person(a, c)$$
$$M_2 : Employee(a, b) \Rightarrow \exists c Person(concat(a, b), c)$$

Transformations

$\mathbf{T_1}$: SELECT FirstName AS Name , $SK_1(Name)$ FROM Employee

As an example for this type of error consider a source schema with an employee relation that stores the first and last names of employees and a target schema with persons (name and gender). Assume there are correspondences between the *FirstName* and *LastName* attributes in the source and the name attribute in the target

schema. We further assume that these correspondences have been misinterpreted leading to the creation of mapping M_1 presented above that copies the first name to the target name attribute. If this error is recognized by realizing that the names of persons in the target are all first name, then TRAMP could be used to investigate the source of the error. At first a *data* provenance query may be issued to investigate the source data from which the incorrect names are derived.

```
SELECT PROVENANCE * FROM target.Person;
```

This query reveals that the name attribute values have been copied from first names in the source and that the source contains correct last names for these persons (the data provenance contains the tuple with first and last names from which a person is derived from). Thus, the error must have been caused by the mapping or transformation. Mapping provenance can be used to reveal which mappings were used to derive the tuples in the target *Persons* relation. E.g.:

```
SELECT getAnnot(tprov), name
FROM
    (SELECT TRANSXML * FROM target.Person) AS tp;
```

This query returns the name attributes of all target person tuples with the mappings that generated them by computing the transformation provenance and extracting the mapping annotations (*getAnnot*, see Appendix C). The result reveals that all tuples have been created by mapping M_1. If the target relation contains a large number of tuples, the user can use the *hasAnnot* meta-querying function (see Appendix C) to check that in fact all tuples have been created by mapping M_1. E.g., by adding a selection condition that checks for tuples that have been derived by other mappings:

```
WHERE NOT hasAnnot(tprov, 'M1');
```

The source of the error has been narrowed down to mapping M_1 and the transformation that implements this mapping. For this simplistic example the user would probably directly use the mapping system to correct the mapping. For more complex cases it may be necessary to inspect the mapping and transformation beforehand.

A.5 Redundant Data

Source Schema	Employee(SSN, Name) Superior(empSSN, supSSN)
Target Schema	Staff(Name, Superior)

Mappings

M_1 : $Employee(a, b) \Rightarrow \exists c Staff(b, c)$

M_2 : $Employee(a, b), Employee(c, d), Superior(a, c) \Rightarrow Staff(b, d)$

Transformations

T_1 :

```
SELECT e.Name AS Name, SK1(e.Name) AS Superior
FROM Employee e
UNION
SELECT e.Name AS Name, e2.Name AS Superior
FROM Employee e, Employee e2, Superior s
WHERE s.empSSN = e.SSN AND s.supSSN = e2.SSN;
```

Assume transformation T_1 presented above was created by a mapping tool to implement mappings M_1 and M_2. This transformation generates redundant tuples for employees that have a supervisor, because these employees are mapped twice; once without the supervisor (M_1) and once with the supervisor (M_2). Note that there are mapping tools that would generate T_1 as the transformation for this set of mappings instead of using an outer join to avoid the generation of duplicates. A user may recognize that several employees are included two times in the target *Staff* relation. Retrieving the *data* provenance for these employees:

```
SELECT PROVENANCE * FROM target.Staff;
```

will reveal that both versions of an employee are derived from the same *Employee* tuple (and one version is also derived from a second *Employee* and *Superior* tuple). A check for duplicates of employee tuples in the source will clarify that the error is caused by the mapping and/or transformation. Asking the following query

```
SELECT TRANSPROV * FROM target.Staff;
```

confirms that one of the employee duplicates is generated by mapping M_1 and the left input of the union in T_1 and the second one by M_2 and, thus, the right input of the union in T_1. Checking the definitions of these mappings, the user will realize that both mappings are correct, and, thus, the error must have been caused by the transformation. If the mapping tool supports it, the user can request a different transformation or correct the transformation manually.

A.6 Instance Data Errors

Source Schema	Person(Name, AddrId) Address(Id, City)
Target Schema	LivesAt(Name, City)

Mappings

M_1 : $Person(a, b) \wedge Address(b, c) \Rightarrow LivesAt(a, c)$

Instances

Person

Name	AddrId
Clara	2
...	...

Address

Id	City
1	Bern
2	Paris
...	...

Mapping M_1 presented above maps persons and addresses to a *LivesAt* relation. Assume that the source instance contains the erroneous information that Clara lives in Paris, but actually she lives in Bern. This instance data error may be easily misconceived as a mapping error. If a user retrieves the *data* provenance for the $(Clara, Paris)$ tuple from the *LivesAt* relation, (s)he will realize that this tuple is derived from the $(Clara, 2)$ and $(2, Paris)$ tuples in the source instance. Hence, after verifying the faultiness of this part of the source instance, it is clear that the error is caused by erroneous instance data and the query has revealed exactly which parts of the source instance caused the error.

B. EXAMPLES FOR COMPUTING PROVENANCE

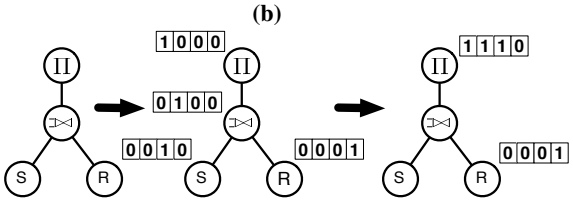
B.1 Transformation Provenance

Fig. 5(a) demonstrates the use of the *TRANSSQL* keyword to trigger *transformation* provenance computation for query q_d from the mapping provenance example presented in Section 4. The algebra tree of this query is depicted on the left in Fig. 5(b). Step **1** of the rewrite algorithm generates the bit-vectors for each operator in the query (Fig. 5(b) in the middle). The right-hand side of the left-join is static, therefore, a fixed bit-vector for this subtree is precomputed. The same applies for the combination of projection, left join and the left input of the join. Either all or none of these operators are in the *transformation* provenance. Therefore, they can be represented as a single bit-vector (shown on the right of Fig. 5(b)). Fig. 5(c) depicts the query after application of the *transformation* provenance rewrite rules (Step **2** of the rewrite algorithm). In this example, the basic structure of the query is preserved and the provenance computation is limited to a projection expression. Function ϵ in the example is used to check if attribute b is *null* (it returns

(a)

```
SELECT TRANSSQL S.c AS d, R.b AS e
FROM S LEFT JOIN R ON (S.c = R.a);
```

(b)



(c)

```
SELECT
  S.c AS d, R.b AS e, f_SQL(1110 ∨ ϵ(b,0001)) AS tprov
FROM S LEFT JOIN R ON (S.c = R.a);
```

(d)

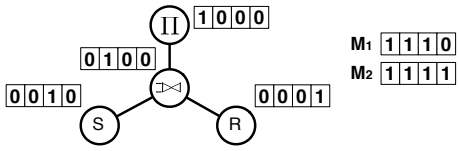
d	e	tprov
2	3	SELECT S.c AS d, R.b AS e FROM S LEFT JOIN R ON (S.c = R.a);
2	5	SELECT S.c AS d, R.b AS e FROM S LEFT JOIN R ON (S.c = R.a);
3	NULL	SELECT S.c AS d, R.b AS e FROM S LEFT JOIN <Not>R</Not> ON (S.c = R.a);

Figure 5: Example of the Computation of Transformation Provenance

(a)

```
SELECT S.c AS d, R.b AS e
FROM
    S ANNOT('M1','M2')
  LEFT JOIN
    R ANNOT('M2')
  ON (S.c = R.a);
```

(b)



(c)

d	e	tprov
2	3	<M2>SELECT S.c AS d, R.b AS e FROM S LEFT JOIN R ON (S.c = R.a);</M2>
2	5	<M2>SELECT S.c AS d, R.b AS e FROM S LEFT JOIN R ON (S.c = R.a);</M2>
3	NULL	<M1>SELECT S.c AS d, R.b AS e FROM S LEFT JOIN <Not>R</Not> ON (S.c = R.a);</M1>

Figure 6: Example of the Computation of Mapping Provenance

an empty bit-vector if its first argument is *null* and its second argument otherwise). The final representation is generated by function f_{SQL} (rewrite algorithm Step **3**). The result of the rewritten query is presented in Fig. 5(d). For the third tuple, the right input of the left outer join did not contribute anything to the result and is therefore not in the *transformation* provenance. In contrast, all operators contributed to the first and second result tuple.

B.2 Mapping Provenance

Fig. 6(a) demonstrates how the new keyword *ANNOT* is used to annotate parts of a query to model the relationships between mappings and their transformations (In this example mappings M_1 and M_2 from Section 4). Fig. 6(b) presents the bit-vectors for the mapping annotation functions μ_{M_1} and μ_{M_2} for query q_d. Recall that $\mathbf{M_2}$ corresponds to the complete query and $\mathbf{M_1}$ does not correspond to the right input of the outer join. Fig. 6(c) presents the result of computing the *transformation* provenance (and because of the annotations also the *mapping* provenance) for q_d using the *TRANSSQL* keyword.

C. META-QUERYING FUNCTIONS

Here we present the *THIS* construct and two XML meta-querying functions as an example of the kind of XSLT and XPath based meta-querying included in *TRAMP* (see Fig. 7): *hasAnnot* and *getAnnots*. The function *hasAnnot(XML,annot)* checks if the query represented as parameter *XML* has the annotation *annot*. This function is realized as a XPath expression over the XML query representation. *getAnnots(XML)* returns all annotations used in a query tree given as parameter XML. The *THIS* construct enables a query to inspect its own XML representation (or the XML representation of one of its subqueries). This is implemented as a query rewrite that replaces the $THIS$ construct with an XML constant that is generated using the same XML generation as in $f_{SQL \rightarrow XML}$.

Name	Semantics
THIS	Returns the XML representation of the query it is used in.
hasAnnot	Checks if an XML document contains a certain annotation.
getAnnot	Return all annotations used in an XML query representation.

Figure 7: Meta-querying Functions and Constructs

D. DATASET AND QUERY DESCRIPTIONS

In this part of the appendix, we give a short description of the Amalgam dataset, source and target schema, and the mappings that were used for the performance evaluation presented in Section 6. Fig. 8 presents the source and target schemas used in the evaluation. The schemas are based on the *Amalgam* [19] integration benchmark. Amalgam uses real-world data (containing errors) and several schemas which represent bibliographic data from various sources. For the experiments, we used a modified version of Schema S_1 from Amalgam as the source schema and a new schema as the target schema. The source schema models authors, institutions, various types of publications (e.g, relation *article*), and the relationships between authors and their publications (for instance, relation *techpublished*). The target schema represents the same information organized in a different way. Several properties of a publication are outsourced into separate relations (e.g., dates) and there is only a single relation that represents publications (regardless of their type). In addition, the affiliation of an author is recorded in the *tauthor* relation directly, rather than in a separate *institute* relation.

Fig. 9 presents some of the mappings we used to map data from the source and target schema. Mappings $\mathbf{M_1}$ and $\mathbf{M_2}$ map authors from the source to the target schema, splitting the *name* attribute into first and last name.[3] $\mathbf{M_1}$ handles authors independent of their affiliations and $\mathbf{M_2}$ maps authors with an affiliation (stored in the

[3]We assume the existence of functions *first* and *last* that return the first (respectively, last) name from a name string.

Source Schema
institute(inst_id, name, location)
inproceedings (inproc_id, title, bktitle, year, month, pages, vol, num, loc, class, note, annote)
article (article_id, title, journal, year, month, pages, vol, num, loc, class, note, annote)
techreport (tech_id, title, inst, year, month, pages, vol, num, loc, class, note, annote)
book (book_id, title, publisher, year, month, pages, vol, num, loc, class, note, annote)
incollection (coll_id, title, bktitle, year, month, pages, vol, num, loc, class, note, annote)
misc (misc_id, title, howpub, confloc, year, month, pages, vol, num, loc, class, note, annote)
manual (man_id, title, org, year, month, pages, vol, num, loc, class, note, annote)
author (auth_id, name, inst)
inprocpublished (inproc_id, auth_id)
articlepublished (article_id, auth_id)
techpublished (tech_id, auth_id)
bookpublished (book_id, auth_id)
incollpublished (coll_id, auth_id)
miscpublished (misc_id, auth_id)
manualpublished (man_id, auth_id)

Target Schema
tauthor (auth_id, first_name, last_name, affiliation)
dates (date_id, year, month)
classification (class_id, name)
journal (jname, publisher)
issue (issue_id, journal, vol, num)
publication (title, author_id, date_id, pages, class_id, issue_id)
notes (pub_title, pub_author, notetext)

Figure 8: Amalgam Schemas

$\mathbf{M_1}$:$author(a,b,c) \Rightarrow \exists d,e : tauthor(d, first(b), last(b), e)$

$\mathbf{M_2}$:$author(a,b,c) \wedge institute(c,d,e) \Rightarrow \exists f : tauthor(f, first(b), last(b), d)$

$\mathbf{M_3}$:$techreport(a,b,c,d,e,f,g,h,i,j,k,l) \wedge techpublished(a,m) \wedge author(m,n,o) \wedge institute(c,p,q) \wedge institute(o,r,s) \Rightarrow$
$\exists t,u,v,w : publication(b,t,u,f,v,w) \wedge date(u,d,e) \wedge classification(v,j) \wedge notes(b,t,k) \wedge notes(b,t,l) \wedge tauthor(t, first(n), last(n), r)$

$\mathbf{M_4}$:$article(a,b,c,d,e,f,g,h,i,j,k,l) \wedge articlepublished(a,m) \wedge author(m,n,o) \wedge institute(o,p,q) \Rightarrow$
$\exists r,s,t,u,v : publication(b,r,s,f,t,u) \wedge date(s,d,e) \wedge classification(t,j) \wedge notes(b,r,k) \wedge notes(b,r,l)$
$\wedge journal(c,v) \wedge issue(u,c,g,h) \wedge tauthor(r, first(n), last(n), p)$

Figure 9: Mappings between the Amalgam Schemas

relation *institute*). The *techreport* and *article* relations are vertically partitioned into relations *publications*, *journal*, *issue*, *notes*, *classification*, and *date* (using $\mathbf{M_3}$ and $\mathbf{M_4}$). In addition, the authors for each publication are also mapped by these two mappings. In mapping $\mathbf{M_3}$ the *institute* relation is referenced twice. The first reference represents the institute of an author and the second one the institute field of the *techreport* relation. The mappings between the other publication types in the source and the target *publication* relation are analogous to $\mathbf{M_3}$ and $\mathbf{M_4}$ and, therefore, not presented here. For each relation in the source that stores publications, we defined two mappings: one for $\mathbf{M_3}$ and $\mathbf{M_4}$ (or their analogs) and one that maps publications of authors without an affiliation. The transformations we generated to implement the mappings use outer joins to deal with overlapping mappings. If several, non-overlapping mappings map data to the same target relation (as is the case for the relation *publication*) they are implemented as a single transformation that unions the transformations of the individual mappings. E.g., the overlapping mappings $\mathbf{M_1}$ and $\mathbf{M_2}$ (mapping authors and their affiliations) are implemented by the following transformation:

```
SELECT
  'SK1' || a.name || COALESCE(i.name,'') AS auth_id,
  get_first_name(a.name) AS first_name,
  get_last_name(a.name) AS last_name,
  CASE WHEN i.name IS NULL
       THEN 'SK2' || a.name
       ELSE i.name END  AS affiliation
FROM
  source.author ANNOT('M1','M2') a
  LEFT JOIN
  source.institute ANNOT('M2') i
  ON (a.inst = i.inst_id);
```

As mentioned in Section 6, we use two sets of queries in the performance evaluation. The first set contains the transformations for the mappings described above. The second set of queries (*Q2*) models basic scenarios from the *STbenchmark* [4] schema mapping benchmark. These mappings are also defined over the Amalgam source schema. The following scenarios were modeled: *Copying*, *Horizontal Partition*, *Vertical Partition*, *self-Joins*, *Denormalization*, *Aggregation*, *Keys and Object Fusion*, *Atomic Value Management*. We could not use the original benchmark scenario generator, because it generates XML data. Table 1 gives a short description for each of these queries.

Scenario	Query Description
Copying	Returns all tuples from *inproceedings*
H. Part.	Returns articles from *publication*
V. Part.	Extracts dates from *techreport*
self J.	Returns all author, co-author combinations
Denorm.	Merges dates and publications
Aggr.	Returns the number of publications per author
KO Fusion	Generates a single publication relation from the source publication relations
AV Man.	Splits author names

Table 1: Description of Query Set 2

E. SEMIRING COUNTEREXAMPLE

We show that *transformation* provenance cannot be defined in the semiring model [15] by presenting two queries q_1 and q_2, a database instance I, and a tuple t such that for any semiring $(K, +, \cdot, 0, 1)$ and K-annotated version of I we have $Q_1^K(t) = Q_2^K(t)$ and $\mathcal{T}(q_1, t) \neq \mathcal{T}(q_2, t)$. That is, there are two queries for which t has different transformation provenance, but carries the same semiring-annotation. Hence, transformation provenance cannot be modeled in the semiring model (we use identifier sets to represent $\mathcal{T}$):

$$R = \{(a)\} \qquad S = T = \emptyset \qquad t = (a)$$
$$q_1 = R^2 \cup^1 S^3 \qquad q_2 = (R^3 \cup^2 S^4) \cup^1 T^5$$
$$\mathcal{T}(q_1, t) = \{1,2\} \neq \{1,2,3\} = \mathcal{T}(q_2, t)$$
$$Q_1^K(t) = (R \cup S)^K(t) = R^K((a)) + S^K((a)) = R^K((a))$$
$$= R^K((a)) + S^K((a)) + T^K((a)) = Q_2^K(t)$$

Note that we use the fact that in the semiring model tuples that do not belong to a relation are annotated with the 0 element of the semiring and that 0 is the neutral element of the $+$ operation. A similar example exists that demonstrates that *PI-CS* cannot be defined in the semiring model.