A Non-intrusive Data-driven Approach to Debugging Schema Mappings for Data Exchange

Laura Chiticariu and Wang-Chiew Tan
UC Santa Cruz
{laura,wctan}@cs.ucsc.edu

1. Problem and Motivation

Data exchange is the problem of transforming data structured according to a source schema into data that conforms to a target schema, usually created independently of the source schema, with constraints of its own. Data exchange is ubiquitous: in general, different entities may store similar data in different formats, thus the ability of restructuring data from one format to another is invaluable in allowing effective use of each other's information. The behavior of a data exchange system is largely governed by the specification of schema mappings. A schema mapping is a high-level declarative specification of how data structured under a source schema is to be converted into data structured under a target schema. A schema mapping based exchange system [5,10] compiles a schema mapping into queries (e.g., SQL, XSLT or XQuery) or other type of executable code. These executables are applied to a given source instance in order to generate a target instance that satisfies the schema mapping.

Schema mappings have several advantages over "lower-level" languages such as XSLT or executable code. They are higher-level programming constructs that specify an exchange at a logical level, hiding the implementation details at the physical level, thus allowing for optimizations. Furthermore, they are easier to manipulate and understand. In fact, schema mappings serve a similar goal as model management [4], where the aim is to reduce programming effort by manipulating higher-level abstractions, called models and mappings between models.

Valuable programming effort needed to implement the desired exchange could be saved if the schema mapping is accurately specified to reflect the user's intention. To date, developmental support for programming schema mappings is still lacking. In particular, a tool for debugging data exchange at the level of schema mappings has not yet been developed, to the best of our knowledge. Our work can be seen as an effort towards developmental support for programming with the language of schema mappings.

The need for debugging support for schema mappings arises for several other reasons. First, schema mappings in data exchange systems are often generated by schema matching tools [13]. Anecdotal evidence shows that while the generated mappings are often close to a user's intention, they likely need to be further refined before accurately reflecting the user's intention. Second, the schema mappings, whether they are known or generated through schema matching tools, can often be large and therefore difficult to debug or understand. Finally, a debugging tool provides a facility for users to understand their specification through "trial-and error". Therefore, a facility that would allow a user to understand the schema mappings by browsing through and probing the data at hand will be extremely useful for enhancing the user's understanding of the specification of the system.

We propose a data-driven method for understanding and debugging schema mappings by visualizing routes for some data in the source or target. A route essentially illustrates the relationship between some source and target data with the schema mapping. A route is informative in that it shows the source data, the source and target schema elements, as well as the intermediate data in the target instance that led to the target data. It has a logical semantics, independent of the underlying procedures used to implement the exchange. Hence, our method is non-intrusive: it can be easily deployed on any data exchange system based on a similar schema mappings formalism as the one considered in this paper, without referring to the underlying execution engine.

Roadmap We discuss preliminaries and illustrate our approach through examples in Section 2. Section 3 discusses two complete, polynomial time algorithms for computing routes. In Section 4 we present results of an empirical evaluation of the feasibility of our algorithms and conclude with an overview of our contributions.

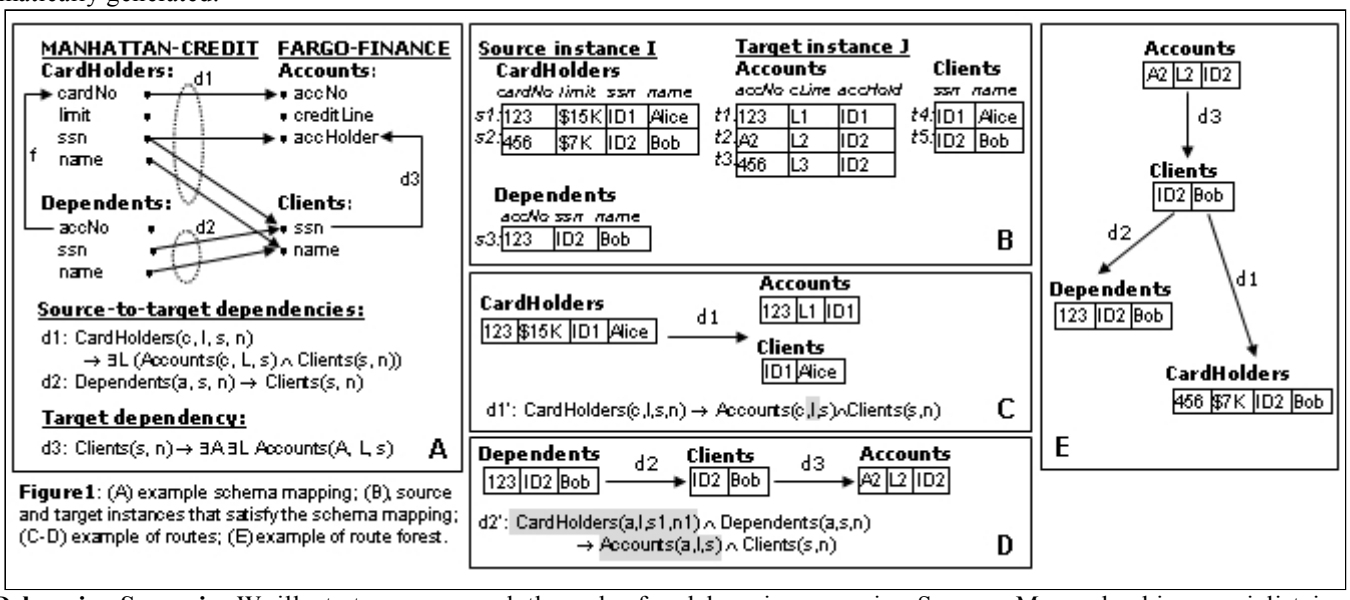
2. Background and Related Work

The specification of a data exchange is given by a *schema mapping M=(S,T, D)* where *S* is the source schema, *T* is the target schema and *D* is a set of dependencies specifying the relationships between the two schemas, as well as constraints on the target schema. The language of schema mappings we consider is widely adopted in research on data exchange and integration [11,12] and it is based on the formalism of *tuple generating dependencies* and *equality generating dependencies* [1].

Figure 1A shows a schema mapping where the source schema Manhattan-Credit and the target schema Fargo-Finance are relational. The specification of the exchange is described by the source-to-target dependencies (s-t dependencies) *d1* and *d2* and the target dependency *d3*. In this example, the objective is to migrate every card holder and dependent of Manhattan Credit as a client of Fargo Finance. According to *d1*, for each CardHolders tuple of Manhattan Credit (refer to the left-hand side of *d1*), there must exist Clients and Accounts tuples in Fargo Finance with corresponding social security number, name and account number values (right-

hand side of *d1*). Similarly, *d2* specifies that for each Dependents tuple in the source, there must be a Clients tuple in the target with the same social security number and name. In addition to *d1* and *d2*, the schema mapping includes a target dependency *d3* that the exchanged target instance must satisfy. This constraint specifies that for every Clients tuple, there must be an Accounts tuple with the same account number. (Note the source constraint *f* which specifies that each dependent must have a corresponding card holder with the same card number. Source constraints are not part of the schema mapping, as we assume that the instance of the source that is being exchanged already satisfies all its constraints.) The s-t dependencies are also illustrated as arrows in the figure. Mapping based exchange systems such as Clio [10] and HePToX [5] generate the arrows in Figure 1A semi-automatically (based on user input), interpret the arrows into dependencies and compile the resulting schema mapping into queries in order to execute the exchange.

Figure 1B illustrates an instance *I* of the source schema, together with a target instance *J* that is a *solution for I under the schema mapping M*. We say that *J* is a solution for *I* under *M* if *J* is a finite target instance such that *I* together with *J* satisfy all dependencies in *D*. Note that the instance *J* may contain "unknown" or null values created during the exchange (e.g., *A2*, *L1*, *L2*, *L3*). The schema mapping allows for incomplete specification in general, hence it might not contain information concerning the creation of certain required elements in the target (e.g., the creditLine value of Accounts tuples is left unspecified in *d1*). Consequently, such values are automatically generated.



Debugging Scenarios We illustrate our approach through a few debugging scenarios. Suppose Mary, a banking specialist, is interested to debug the schema mapping in Figure 1A. We expect that in most cases, Mary would provide her own (small) test data for the source. The underlying data exchange system is used to exchange the source data under *M* and obtain a target instance that is a solution for the source test data under *M*. In our example, we assume Mary uses the source instance *I* and the target instance *J* from Figure 1B.

Scenario 1 While browsing through *J*, Mary discovers that the creditLine value of the tuple *t1* in Accounts contains a null *L1*. Knowing that Manhattan Credit would not allow accounts with no pre-assigned credit limit value, she probes *t1*. Our debugger shows a route from the source that is a witness for *t1* in the target, schematically depicted in Figure 1C. The route consists of the source tuple *s1* in CardHolders, the dependency *d1*, as well as an assignment *h:{c->123,l->$15K,s->ID1,n->Alice,L->L1}* of variables of *d1*. (The assignment *h* is not shown in the figure.) Under this assignment, the right-hand side of *d1* is the tuples *t1* and *t4* in Figure 1B. Hence, *d1* witnesses the presence of *t1* with *s1* and *h*. With this route, Alice discovers that the limit of Alice's card (i.e., the value "$15K") was not copied over by the dependency. Indeed, Figure 1 shows that there is no arrow between any schema element of CardHolders and creditLine of Accounts. Mary therefore corrects *d1* to *d1'* in Figure 1C, which adds the missing correspondence between limit of CardHolders and creditLine of Accounts. In this scenario, our debugger has helped Alice discover an incomplete association between source and target schema elements.

Scenario 2 Further browsing through the target instance, Mary sees that the accNo of the account of a customer identified by I*D2* is unspecified (the null *A2* of tuple *t2*). Mary probes *t2* and the our debugger shows the route with two steps depicted in Figure 1D. This route demonstrates that *t2* was created through the target dependency *d3* with the tuple *t5*. Furthermore, *t5* was created through *d2* with the source tuple *s3*. With this information, Mary discovers that *d2* is in fact missing an association with the source relation CardHolders. Indeed, every dependent card holder must have a sponsoring card holder in CardHolders and they share the same credit limit. Furthermore, the target now includes an Accounts relation that is used to hold the account number and ssn of the dependent card holder, as well as the credit limit of the sponsoring card holder. Mary may correct *d2* to obtain *d2'* as in Figure 1D, or she may

choose to remove *d2* completely if dependent card holders of Manhattan Credit are not automatically customers of Fargo Finance.

Various other types of errors that may arise in a schema mapping specification are identified in [6]. 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 from the source, as shown in Scenario 2. Second, there are situations in which a computed route may not reveal any problems with the schema mapping. Hence, the user may need the knowledge of other alternative routes or all routes (for the same selected data). It is conceivable, for example, that the tuple *t2* contains sensitive information and in this situation, the knowledge of all routes for *t2* would be crucial for the purpose of identifying dependencies that export sensitive information. The knowledge of all routes is also valuable in scenarios where data from multiple, highly overlapping sources is exchanged into a single target. We believe that in general however, it is valuable to incorporate both features in a debugger. In some cases, the user may be satisfied with one route. However, it is also useful to be able to determine all routes whenever desired. Third, in our debugging scenarios, we have only illustrated the use of routes for understanding the schema mapping through anomalous tuples. However, routes for correct tuples are also useful for understanding the behavior of the schema mapping in general.

Related Work A framework for understanding and refining schema mappings in Clio is proposed in [19]. The main focus of [19] is the selection of a good source and target instance that is illustrative for the behavior of a schema mapping, where the source and target are relational and there are no explicit target dependencies. Our debugger differs from [19] in that: (1) Our debugger works for relational or XML schema mappings, and explicitly considers target dependencies. (2) 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 approach of [19] is intended to assist in creating an initial schema mapping "from scratch" and it is thus complementary to our debugger, which allows for debugging or refining an existing schema mapping. Incorporating the functionality of [19] into our debugger and investigating what are representative instances for debugging in general is subject of our current investigations [7].

Commercial systems such as Altova's Mapforce and Stylus Studio ship with integrated facilities for data exchange, but debug only at the "lower-level"; their debuggers are for the XSLT or XQuery language that is used to specify the exchange.

The problem of computing routes is similar in spirit to the problem of computing the provenance (or lineage) of data. There are two approaches for computing provenance. In the *eager* or *bookkeeping approach*, the transformation is re-engineered to keep extra information from the execution, in order to subsequently answer provenance related questions. In contrast, the transformation is not reengineered in the *lazy approach*, where provenance is computed by examining only the transformation, and the source and target databases. There is existing research on both the lazy [8] and eager [4,9] 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 [17]. 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 and we emphasize that this is a design choice, as we want our techniques to readily work on top of Clio or other data exchange systems that are based on a similar formalism for schema mappings. Apart from requiring the reengineering the underlying data exchange system, we believe that type of provenance considered in [17] may be unsatisfactory in debugging schema mappings. In Scenario 1, for example, the system of [17] would enable Mary to learn that *t1* is witnessed by *d1* and a CardHolder tuple, but it does not indicate which CardHolder tuple!

Our route algorithms bear resemblance to sophisticated top-down resolution techniques [14,16,18] used in deductive databases. As our route algorithms, these approaches use memoization to avoid redundant computations and infinite loops. A fundamental difference, however, is that we make use of the target instance, which is available to us. In consequence, we may able to detect if a tuple has no routes early in the computation and achieve a scalable implementation. In contrast, the output of a deductive program is not available during resolution and the computation always continues down to the source tuples before deciding whether a tuple belongs to the output.

3. Our Approach

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. Let M be a schema mapping, *I* be a source instance, *J* be a solution for *I* under M and let *Js* be a set of selected target tuples. A route for *Js* with M, *I* and *J* is a finite non-empty sequence of satisfaction steps that demonstrates the existence of *Js* with the schema mapping and some source and target data. Intuitively each individual satisfaction step demonstrates *how* a dependency is satisfied with tuples in *I* and *J*. For example, the route for the tuple *t1* illustrated in Figure 1C consists of a single satisfaction step which demonstrates, through the assignment *h*, how *d1* is satisfied with tuples from *I* and *J*. Routes have declarative semantics, not tied to any procedural semantics associated with the transformation of data from the source to the target according to the schema mapping. Hence, our techniques can be easily deployed any data exchange system based on similar schema mappings formalisms.

We designed an algorithm (*ComputeAllRoutes*) for computing all routes for a given set of target tuples *Js* in J, where *J* is a *solution*

for a source instance *I* under a fixed schema mapping *M*. The input to *ComputeAllRoutes* consists of *I*, *J* and *Js*. Our algorithm works for any solution *J* for *I* under *M* and so, we are not limited to the solutions that are generated by a specific data exchange engine. Our algorithm constructs a route forest, in polynomial time in the size of the input, that concisely represents *all routes* for *Js*. In [6], 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 route forest constructed for the tuple *t2* in our example is schematically illustrated in Figure 1E. It embeds two routes for *t2*: one of them is shown in Figure 1D, while the other demonstrates *t2* with *d1* and *d3*. Intuitively, the construction of the route forest proceeds as follows. We first match each selected target tuple *t* in *Js* against the right-hand side of the s-t and target dependencies. For each dependency *d* that can witness *t* and for each assignment that witnesses *t* with *d* and some tuples *U*, we add to the route forest a branch from *t* to a compound node containing all the tuples *U*. If *d* is a target dependency (i.e., *U* are target tuples), the computation proceeds recursively to expand each fact in *U*. In other words, we continue by constructing the route forest for *U*. Assignments that demonstrate *t* with *d* are obtained by posing two queries with appropriate selection predicates on *I* and *J* (or *J* only, if *d* is a target dependency), corresponding to the left-hand side and respectively the right-hand side of the dependency. The process repeats until all the target tuples encountered during the computation have been expanded.

The main challenge in designing *ComputeAllRoutes* was to avoid an exponential running time, as well as the trap of infinite loops, naturally arising in case of recursive schema mappings. Our algorithm terminates and produces a compact, polynomial size representation for all routes for *Js*, even thought there may be exponentially many routes for *Js*. In doing so, we ensure that every tuple *t* encountered during the construction of the route forest is expanded exactly once. Whenever *t* is subsequently encountered during the computation (in another position in the route forest), we add a reference to the position in the route forest where *t* was first encountered (and expanded). Thus, we avoid redundant computation and infinite loops and achieve a polynomial running time, as the number of possible tuple expansions is polynomial in the size of the input.

We have also designed an algorithm (*ComputeOneRoute*) that outputs one route for selected target tuples *Js* in time polynomial in the size of the input. Our algorithm is a non-trivial adaptation of *ComputeAllRoutes* where we stop the computation after discovering the first route and it is sound and complete: it outputs one route for *Js* if and only if there is a route for *Js*. Alternative routes can be retrieved by resuming the search from the point we last stopped.

Although not discussed here, we propose a similar notion of routes for selected source tuples (as opposed to target tuples), useful in identifying consequences of source data in the target. Algorithms for computing routes for source tuples are similar in spirit to our algorithms for computing routes for target tuples and are described in [2].

4. Results and Contributions

We have implemented our routes algorithms in the SPIDER schema mapping debugger [2], currently deployed on top of the Clio data exchange system [10]. Our implementation handles relational/XML to relational/XML data exchange although we have discussed only the relational-to-relational case in this paper. In this section we report on an empirical evaluation of the feasibility of our routes algorithms on both synthetic and real datasets. In the future, we would like to conduct a user-study in order to asses the effectiveness of our techniques in facilitating the process of debugging schema mappings.

Empirical evaluation We designed several synthetic data exchange scenarios based on the schema and dataset of the TPCH database benchmark [15], with the goal of measuring the influence of various parameters on the scalability of our algorithms. These parameters are: the size of source (and target) instances, the number of tuples selected by a user and the complexity of the schema mappings (i.e., the average number of joins between relations indicated in the left-hand side and respectively, the right-hand side of the dependencies). Details on the construction of our synthetic scenarios are in [6].

Figure 2A, illustrates the influence that the size of the input (i.e., the size of source and target instances, and the number of tuples for which a route needs to be computed) has on the performance of *ComputeOneRoute* in a fixed schema mapping. As expected, the performance degrades with an increase in the number of selected tuples, since in general, this increase translates into longer routes (i.e., more satisfaction steps are computed). The execution time also increases with the size of the source and target instances, since the time required to retrieve assignments from the database increases with the size of these instances. We have performed a similar suite of experiments with *ComputeAllRoutes* and observed similar trends. As expected, however, *ComputeAllRoutes* performs slower compared to *ComputeOneRoute*, since it is exhaustive. 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. Figure 2B shows a comparison between the running times of the two algorithms when the source and target instances are of size 100MB and respectively, 600MB. (Note the logarithmic scale). For 5 tuples, one route is found in 2 seconds, while *ComputeAllRoutes* requires about 100 seconds to construct the route forest. We emphasize that our synthetic scenarios have been designed to stress test our algorithms. We believe that a user is unlikely to select as many as 10 tuples, or use instances of hundreds of megabytes in debugging. In general, we

expect a user to be interested in a smaller number of tuples at any time and debug with small instances, cases in which both our algorithms perform well.

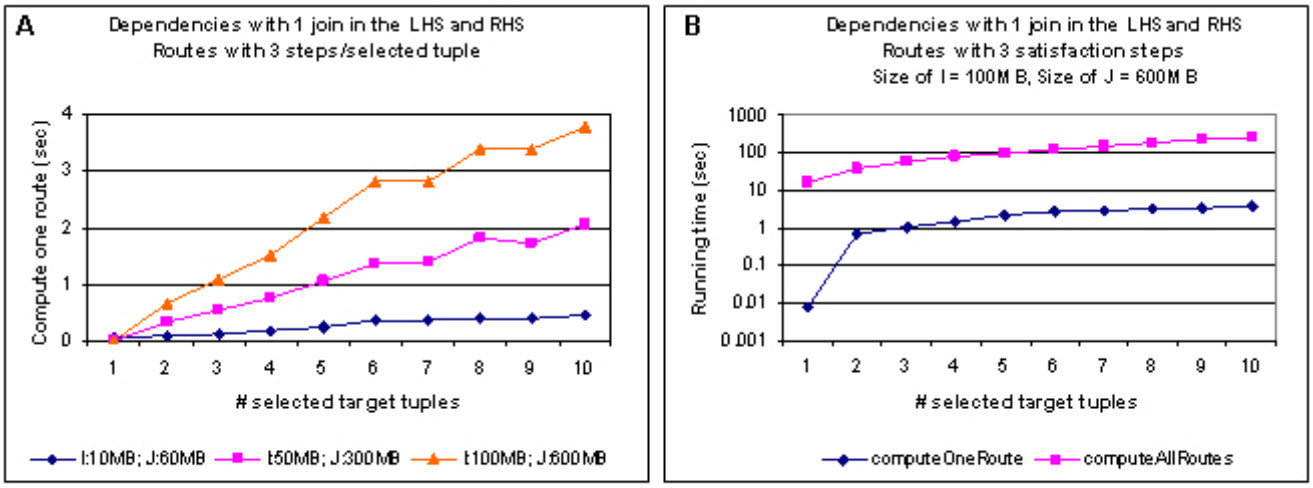


Figure 2: (A) Influence of the size of the input on *computeOneRoute*; (B) comparison in performance between the two algorithms.

The running time of both algorithms also increases with an increase in the complexity of the schema mapping (graphs not shown). This is expected, since the number of "intermediate" tuples encountered in the process is larger and the queries that retrieve assignments to witness these tuples are more complex.

We have also experimented with two data exchange scenarios which we created based on complex real relational and XML schemas and datasets available on-line [6]. We computed (one or all) routes for one to ten randomly selected target facts in both scenarios. In all cases, the time required to find one route was under 3 seconds, while *ComputeAllRoutes* took at most 18 seconds to construct the routes forest.

Contributions We describe a data-driven, non-intrusive technique for debugging schema mappings in data exchange, centered around a novel notion of routes for selected source or target data. To the best of our knowledge, ours is the first effort towards developing a full-fledged debugger for schema mappings. We designed two complete, polynomial time algorithms for computing all, and respectively one route for selected source or target data. Our experimental results demonstrate the feasibility of both algorithms under reasonable debugging conditions.

5. 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 (Demo). In VLDB, pages 1179-1182, 2006.

[3] P. A. Bernstein. Applying model management to classical meta data problems. In CIDR, pages 209–220, 2003.

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

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

[6] L. Chiticariu, and W. Tan. Debugging Schema mappings with Routes. In VLDB, pages 79-90, 2006.

[7] B. Alexe, L. Chiticariu, R.J. Miller, and W. Tan. Muse: Mapping Understanding and dEsign by Example. Technical Report, UC Santa Cruz, 2007.

[8] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. TODS, 25(2):179–227, 2000.

[9] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In ICDE, page 82–93, 2006.

[10] 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.

[11] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In PODS, pages 61–75, 2005.

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

[13] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. The VLDB Journal, 10(4):334–350, 2001.

[14] H. Tamak*I* and T. Sato. Old resolution with tabulation. In ICLP, pages 84–98, 1986.

[15] TPC Transaction Processing Performance Council. http://tpc.org.

[16] J. Ullman. Implementation of logical query languages for databases. In TODS, pages 289–321, 1985.
[17] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and querying data transformations. In ICDE, pages 81–92, 2005.
[18] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In EDS, pages 179–193, 1986.
[19] L. Yan, R. Miller, L. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In SIGMOD, pages 485–496, 2001.