DEMo: Data Exchange Modeling Tool *

Reinhard Pichler
Technische Universität Wien A-1040 Vienna, Austria
pichler@ dbai.tuwien.ac.at

Vadim Savenkov
Technische Universität Wien A-1040 Vienna, Austria
savenkov@ dbai.tuwien.ac.at

ABSTRACT

Minimality is an important optimization criterion for solutions of data exchange problems, well captured by the notion of the core. Though tractability of core computation has been proved, it has not yet become a part of any industrial-strength system, still being highly computationally expensive. In this demonstration, we show how core computation can be used in a data exchange modeling tool, allowing data engineers to design more robust data transfer scenarios and better understand the sources of redundancy in the target database.

1. INTRODUCTION

Data exchange is concerned with the transfer of data between databases with different schemas, governed by source-to-target dependencies and target dependencies. The source and target schemas together with the sets of dependencies constitute the *schema mapping*. The *data exchange problem* associated with a schema mapping $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ is the task of constructing, given a source instance I, a target instance J whose attribute values are those of I plus some newly invented *labeled nulls*, such that all of the source-to-target dependencies Σ_{st} and target dependencies Σ_t are satisfied. Such a J is called a *solution* to the data exchange problem. Following [3], we confine ourselves to dependencies of the form $\phi(\vec{x}) \rightarrow \exists\vec{y}\ \psi(\vec{x},\vec{y})$ where the free variables $\vec{x}$ are understood to be universally quantified. Moreover, the *antecedent* ϕ is a conjunction of atoms and the *conclusion* ψ is either a conjunction of atoms or a conjunction of equalities. In the former case, we speak of a *tuple generating dependency* (TGD) while the latter case is referred to as an *equality generating dependency* (EGD). Typically, the number of possible solutions to a data exchange problem is infinite. A natural requirement (proposed in [3]) on the solutions is *universality*, that is, there should be a homomorphism from the materialized solution to any other possible solution.

*This work was supported by the Vienna Science and Technology Fund (WWTF), project ICT08-032. V. Savenkov additionally receives a scholarship from the European program "Erasmus Mundus External Cooperation Window".

VLDB '09, August 24-28, 2009, Lyon, France

EXAMPLE 1. *Let the source instance consist of two relations* BasicUnit(course, lecturer, tutor)*:* $\{(java, john, john)\}$ *and* AdvancedUnit(course)*:* $\{java\}$*, i.e., John gives both the lecture and the tutorial on Java programming, and there are advanced chapters in the Java course. Let the target schema comprise four relation symbols* Faculty(idf, name)*,* NeedsLab(id_faculty, lab)*,* Course(idc, course) *and* Teaches(id_faculty, id_course)*, and consider the source-to-target TGDs (1, 2) and the target foreign key:*

1. $\mathsf{AdvancedUnit}(\mathsf{C}) \rightarrow \exists \mathsf{Idc}, \mathsf{Idf}, \mathsf{N}\ \mathsf{Course}(\mathsf{Idc}, \mathsf{C}), \mathsf{Faculty}(\mathsf{Idf}, \mathsf{N}), \mathsf{Teaches}(\mathsf{Idf}, \mathsf{Idc})$.
2. $\mathsf{BasicUnit}(\mathsf{C}, \mathsf{L}, \mathsf{T}) \rightarrow \exists \mathsf{Idc}, \mathsf{Idt}, \mathsf{Idl}\ \mathsf{Course}(\mathsf{Idc}, \mathsf{C}), \mathsf{Faculty}(\mathsf{Idl}, \mathsf{L}), \mathsf{Teaches}(\mathsf{Idl}, \mathsf{Idc}), \mathsf{Faculty}(\mathsf{Idt}, \mathsf{T}), \mathsf{Teaches}(\mathsf{Idt}, \mathsf{Idc})$.
3. $\mathsf{Teaches}(\mathsf{Idf}, \mathsf{Idc}) \rightarrow \exists \mathsf{L}\ \mathsf{NeedsLab}(\mathsf{Idf}, \mathsf{L})$.

Then the following instances are all valid solutions:

$$J = \{\ \mathsf{Course}(\mathsf{C}_1, \mathsf{java}), \mathsf{Course}(\mathsf{C}_2, \mathsf{java}), \mathsf{Faculty}(\mathsf{F}_1, \mathsf{N}_1), \mathsf{Faculty}(\mathsf{F}_2, \mathsf{john}), \mathsf{Faculty}(\mathsf{F}_3, \mathsf{john}), \mathsf{Teaches}(\mathsf{F}_1, \mathsf{C}_1), \mathsf{Teaches}(\mathsf{F}_2, \mathsf{C}_2), \mathsf{Teaches}(\mathsf{F}_3, \mathsf{C}_2), \mathsf{NeedsLab}(\mathsf{F}_1, \mathsf{L}_1), \mathsf{NeedsLab}(\mathsf{F}_2, \mathsf{L}_2), \mathsf{NeedsLab}(\mathsf{F}_3, \mathsf{L}_3)\}.$$

$$J_c = \{\mathsf{Course}(\mathsf{C}_1, \mathsf{java}), \mathsf{Faculty}(\mathsf{F}_1, \mathsf{john}), \mathsf{Teaches}(\mathsf{F}_1, \mathsf{C}_1), \mathsf{NeedsLab}(\mathsf{F}_1, \mathsf{L}_1)\}.$$

$$J' = \{\mathsf{Course}(\mathsf{java}, \mathsf{java}), \mathsf{Faculty}(\mathsf{F}_1, \mathsf{john}), \mathsf{Teaches}(\mathsf{F}_1, \mathsf{java}), \mathsf{NeedsLab}(\mathsf{F}_1, \mathsf{L}_1)\}.$$

Note that J' in Example 1 is not universal. Indeed, a homomorphism has to map every constant onto itself [3]. Hence, there exists no homomorphism $h\colon J' \rightarrow J$, since the fact Course(java,java) cannot be mapped onto any fact in J.

Universal solutions can be obtained with the well-known *chase* procedure (cf. [3]), which is essentially an iterative "patching" of the target database instance in order to satisfy all the dependencies, for a given source database. In particular, an iteration (*chase step*) enforcing a TGD adds missing atoms to the target instance, while an EGD is enforced by unifying values in the target relations. More formally, for a source database I and a target database J,

- if there is a TGD $\phi(\vec{x}) \rightarrow \exists\vec{y}\psi(\vec{x},\vec{y})$, s.t. $\phi(\vec{a})$ is satisfied for some assignment $\vec{a}$ on $\vec{x}$, and $J \not\models \exists\vec{y}\,\psi(\vec{a},\vec{y})$ then J is extended with facts corresponding to $\psi(\vec{a},\vec{z})$, where the elements of $\vec{z}$ are *fresh labeled nulls*;
- if there exists an EGD $\phi(\vec{x}) \rightarrow x_i = x_j$, s.t. $J \models \phi(\vec{a})$ for some assignment $\vec{a}$ on $\vec{x}$, but the corresponding values a_i, a_j in $\vec{a}$ are not equal, then a null a' among $\{a_i, a_j\}$ is chosen and *every occurrence* of a' in J is replaced by the other term. Moreover, in a case when both a_i, a_j are constants and $a_i \neq a_j$, the chase halts with *failure*.

The result of chasing the source instance with the source-to-target dependencies and further chasing the target instance with the target dependencies is called the *canonical universal solution*. In our example, J is such a solution, obtained by enforcing the dependencies in the order they are listed:

1. TGD (1) adds the facts $\mathsf{Course}(\mathsf{C}_1, \mathsf{java})$, $\mathsf{Faculty}(\mathsf{F}_1, \mathsf{N}_1)$ and $\mathsf{Teaches}(\mathsf{F}_1, \mathsf{C}_1)$.
2. By TGD (2), the facts $\mathsf{Course}(\mathsf{C}_2, \mathsf{java})$, $\mathsf{Faculty}(\mathsf{F}_2, \mathsf{john})$, $\mathsf{Faculty}(\mathsf{F}_3, \mathsf{john})$, $\mathsf{Teaches}(\mathsf{F}_2, \mathsf{C}_2)$, and $\mathsf{Teaches}(\mathsf{F}_3, \mathsf{C}_2)$ are added.
3. The three final chase steps add the NeedsLab facts.

Universal solutions are not unique: For instance, if we swap the first two chase steps, then the first three facts would not need to be created. Indeed, on our particular source database, enforcement of the TGD (2) also satisfies the rule (1). In other words, J is not minimal, i.e., it contains unnecessarily many tuples and, in particular, unnecessarily many nulls, which we shall refer to as *redundant* in the sequel. Note however that J is universal. Hence, there is a homomorphism mapping it onto another, smaller, solution. As shown in [4], the smallest possible universal solution, called the *core*, is always contained in every other universal solution, and can be found by computing a *proper endomorphism*, mapping a database instance onto a proper subset of it (in our example, J_c is the core). Unlike universal solutions, cores are unique up to isomorphism [4], that is, renaming of labeled nulls. The typical data exchange procedure, proposed in [4], is illustrated in Figure 1.

The complexity of core computation was investigated in several papers ([4, 5, 6]). The most general polynomial time algorithm for core computation is the one developed by Gottlob and Nash [6], which has been further enhanced in [8], where also the first prototype implementation of core computation is reported. Recently, a new promising approach of core computation through a specially adapted chase has been reported [7]. However, such a core computation is only possible in absence of target constraints so far, while our tool works with more expressive mappings, covering sets of weakly acyclic target TGDs and EGDs.

As our experience with an implementation of the core computation suggests [8], the amount of redundancy introduced in the target instance by the chase can differ enormously for a different design of the mappings and target dependencies in the data exchange scenario. Hence, especially for large databases, where the final minimization step is not feasible in reasonable time, it is essential to keep the redundancy introduced by the chase as small as possible. To this end, we present the DEMo system for **D**ata **E**xchange **Mo**delling that focuses on evaluating and debugging data exchange scenarios with respect to redundancy in the generated target databases. It assists the data engineer in designing the mappings and target dependencies and in the choice of a chase order. It also allows to test the transfer on a sample of the source data and to check the optimality of the obtained solution by computing its core.

The focus on the redundancy aspect of the data exchange is a key difference of our tool from other systems for mapping analysis, like, e.g., SPIDER [1] or MUSE [2]. To the best of our knowledge, no other system addressing this issue has been reported so far.

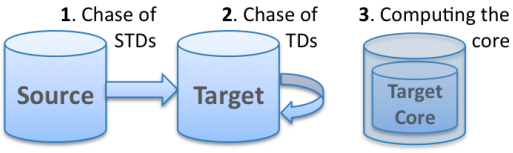

Figure 1: Complete data exchange process.

2. CORE COMPUTATION

Core computation plays a central role in our tool, and it is, therefore, described in more detail here. Core computation essentially comes down to eliminating as many nulls as possible from the target database. Those nulls (each being assigned a label) are introduced during the chase by the application of TGDs with existentially quantified variables. The core computation algorithms in [6] and [8] take the target instance J resulting from the chase and successively compute a series of nested subsets $J = J_0 \supset J_1 \supset \ldots$ via endomorphisms $h_i \colon J_i \to J_i$ with $J_{i+1} = h_i(J_i)$. The core is reached when no further proper (i.e., non-surjective) endomorphism exists. In order to search for these endomorphisms, we have to check for all possible pairs (x, y) of domain elements in J_i with $x \neq y$ if there exists an endomorphism h_i that maps x and y to the same value. Clearly, such an h_i is non-injective and, over a finite domain, also non-surjective.

The search for such an endomorphism h_i in polynomial time requires care, bearing in mind that deciding if there exists a homomorphism between two graphs (or, more generally, between two structures) is a classical NP-complete problem. Tractability in our case is ensured by splitting the search for h_i in two major substeps: First, we have to search for a non-injective homomorphism $h_i' \colon J_i' \to J_i$ for a subset J_i' of J_i with some structural property ensuring that the search is feasible in polynomial time. Then, h_i' is extended to a homomorphism $h_i \colon J_i \to J_i$. For details, see [6, 8].

The overall structure and the asymptotic worst-case behavior of the algorithms in [6] and [8] are very similar. The principal difference between these two algorithms is how they deal with EGDs in the target dependencies: The algorithm in [6] simulates EGDs by means of an auxiliary, binary predicate E (where an equality $s = t$ is encoded by the atom $E(s, t)$) and by introducing TGDs which ensure that E has the essential properties of equality (like reflexivity, symmetry, transitivity, replacement property, etc.). The target chase is done with the modified target dependencies and the enforcement of the original EGDs happens in the course of the core computation. In particular, the result of the target chase is, in general, not a solution of the data exchange problem (since the EGDs are not fulfilled), while the core is guaranteed to be a solution. In contrast, in the approach of [8], the EGDs are applied directly as part of the target chase. Hence, the result of the target chase indeed is a solution to the data exchange problem.

Our DEMo tool is based on an implementation of the core computation algorithm from [8], which has two major advantages:

1. The result J of the chase is indeed a solution to the data exchange problem and so are all nested subinstances $J = J_0 \supset J_1 \supset J_2 \ldots$ which are constructed on the way to the core. Hence, the core computation can stop at any time with an approximation J_i of the core. Such approximations are valid universal solutions, that are more compact than a canonical one, but might still not be cores. However, due to the use of heuristics for the core search, already the first approximation computed by the algorithm can be quite close to the ultimate core (for instances with high redundancy, up to 90% of redundant nulls can be eliminated already in the first approximation). This feature fits well into the modeling scenario where the data engineer needs a first quick evaluation of the dependency design. For most cases, core approximations allow to witness sub-optimality of the data exchange process in time substantially shorter than needed to compute the ultimate core. We return to this point when discussing the demonstration scenario.
2. Even more importantly, in the core computation algorithm

of [8], the EGDs are applied directly rather than simulated by TGDs. Hence, the redundancy introduced by the chase is exclusively due to the data exchange scenario (plus the chase order). In contrast, the redundancy introduced into the target database by the algorithm of [6] may be due to the additional TGDs which are needed to simulate the EGDs. Hence, even if a lot of redundancy is introduced by the chase and later eliminated by the core computation, the algorithm of [6] would not allow us to draw any conclusions on the quality of the dependency design.

3. OPTIMIZING THE DATA EXCHANGE

Consider the following target TGD:

$$S(x, y) \rightarrow \exists v, w, z\, R(y, v) \wedge P(x, w) \wedge P(z, w).$$

If applied to a database $\{S(1, 2), R(2, v_1)\}$, it will create the tuples $\{R(2, v_2), P(1, w), P(z, w)\}$. The tuple $R(2, v_2)$ is clearly redundant, but the standard chase procedure does not allow to add only part of the tuples specified by the right-hand side of the TGD. It is not difficult to show that the above dependency is logically equivalent to the conjunction of two other dependencies (the R-atom does not share existentially-quantified variables with the P-atoms): $S(x, y) \rightarrow \exists v\, R(y, v)$, and $S(x, y) \rightarrow \exists w, z\, P(x, w) \wedge P(z, w)$, allowing to avoid the unnecessary creation of $R(2, v_2)$. We call such a splitting of TGDs *normalization*. More generally, we can split any TGD τ according to the connected components of the *Gaifman graph* of the right-hand side $\psi(\vec{x}, \vec{y})$ of τ, considering only the new variables $\vec{y}$, i.e., this graph contains as vertices the variables in $\vec{y}$. Moreover, two variables y_i and y_j are adjacent if they jointly occur in some atom of $\psi(\vec{x}, \vec{y})$.

Next, also the tuple $P(z, w)$ is not part of the core. We would not have to introduce it, if the latter TGD were first minimized to a smaller, yet logically equivalent one: $S(x, y) \rightarrow \exists w\, P(x, w)$. Thus, the second optimization technique is based on conjunctive query minimization. Actually, conjunctive query minimization is NP-complete. However, in data exchange, only the data complexity is normally considered while the data exchange scenario is taken as fixed. Indeed, the tractability of data exchange [3] and of core computation [6] only refers to the data complexity.

The next technique is concerned with coincidences in the source data: in Example 1, both the BasicUnit table and AdvancedUnit table mention the same course (Java). As already shown earlier, mere changing the order of enforcement of TGDs (1) and (2) would suppress the creation of redundant tuples in such situations. Informally, this rule can be formulated as "more specific sets of tuples should be added first". Let us now look at the rule (2) alone. In cases where the same person gives lectures and tutorials in some course (like it happened with Java programming), this person is entered twice in the Faculty table. This effect could be prevented by the introduction of an additional, specialized version of (2): $\mathsf{BasicUnit}(\mathsf{C}, \mathsf{T}, \mathsf{T}) \rightarrow \mathsf{Course}(\mathsf{Idc}, \mathsf{C}), \mathsf{Faculty}(\mathsf{Idt}, \mathsf{T}), \mathsf{Teaches}(\mathsf{Idt}, \mathsf{Idc})$. This rule, if placed earlier in the chase sequence than the original one, would allow to avoid the creation of redundant tuples.

4. SYSTEM OVERVIEW

The the two main system components are the user interface, which will be described in the next section, and the data exchange engine, implementing the chase and core computation functionality. The system is written in Java. Additionally, the data exchange engine heavily relies on the connected DBMSs, to which the whole data transfer and large parts of the core computation are delegated, for instance, the search for an endomorphism is done by evaluating automatically generated SQL queries.

The implementation is standards-based and extensible: any JDBC data source can be used as a source instance, and either PostgreSQL or Oracle are currently supported for the target database. Internally, the data exchange scenarios are stored as XML files, and XSLT is heavily used to automatically generate SQL queries. In order to accommodate the system to any other DBMS product for the target instance, we would simply have to modify the XSLT templates.

To store null labels, the system creates additional columns in the target relations. However, the data engineer only needs to specify the logical schema; all auxiliary columns and other database objects are generated automatically by the system. Many such objects are used to facilitate the debugging of the data exchange scenario and to allow for effective core computation. Particularly important is the lineage information, tracking the chase process and explaining the creation of any particular tuple in the database. Exploring the lineage by means of the graphical user interface is shown in Figure 2.

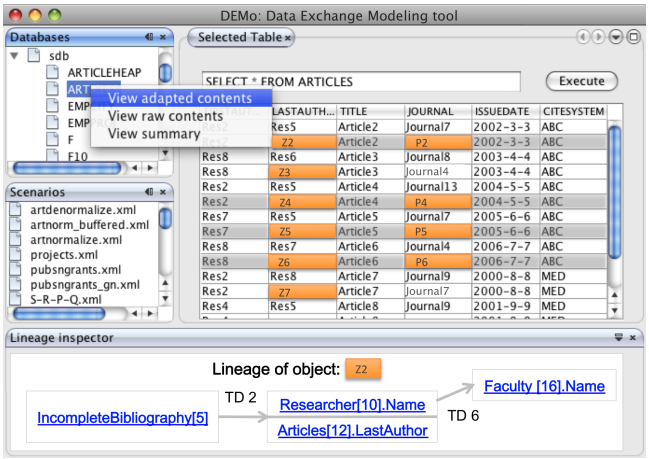

Figure 2: Lineage explorer.

Apart from generating additional information, also an automatic cleanup of the target database is supported, thus facilitating the gradual development of data exchange scenarios.

5. DEMONSTRATION SCENARIO

The DEMo system supports an interactive graphical user interface. We will showcase the system by working on a sample data exchange scenario, imitating the work session of a data engineer. Scenarios are comprised of two schemas, mappings between them (represented by a set of source-to-target dependencies), and target dependencies. The focus of DEMo is the design of dependencies and finding an appropriate chase order so as to arrive at a more compact target database.

We start with showing how our tool assists the data engineer in the dependency design: First, the user specifies the desired mappings and target dependencies and lets the system analyze them (to save time at the demonstration, we will prepare some sample scenarios illustrating the main issues addressed by our tool). If problems are detected, the corresponding dependencies are highlighted by the system, and fixes (e.g., dependency rewrites) are proposed. Every detected problem is accompanied by a textual explanation. For instance, for the TGD (2) from Example 1, a specialized version will be proposed to handle the case when the lecture and the tutorial are given by the same person, as explained above. We call

such a situation where the same value resides at different positions in the database, leading to the introduction of redundant tuples (that would otherwise not be redundant) a *coincidence*.

Clearly, it is important not to introduce specialized rules excessively. With the help of our system, the data engineer can first check if it makes sense to introduce such a rule: The system provides a convenience means for querying the source data, to find out how frequent a certain coincidence is and if the introduction of a new specialized rule is justified. Each dependency change proposed by the system can be either accepted or rejected by the user, see Figure 3.

The second step of the demonstration is concerned with the design of the chase sequence. Order constraints will be assigned to some groups of dependencies (e.g., more specific rules should go prior to more general rules). Each order constraint will have an annotation explaining its meaning. A default order satisfying all the constraints will be proposed, but the user can rearrange the dependencies manually, or delete any order constraint.

At the next step, the data transfer is performed, i.e., the chase procedure is executed. For each source table, it is possible to assign a condition limiting the number of transferred rows, which is useful for big source databases, where only a small sample database should be involved in the modeling. If EGDs are part of the target constraints, the chase may fail: this means that an attempt to unify different constants was made. In this situation, the offending tuple combination will be shown to the dependency designer. If the chase succeeds, then the solution can be explored in the database viewer. For convenience, the null labels and the usual "constant" values are represented in one column, nulls being marked with color. Along with the visual exploration, users can see the summary, showing the chase statistics, number of applications and timing for each rule, and the list of columns where most of the nulls are stored. For each null, it is possible to see all its occurrences (as a list of tuple references), and its lineage, i.e. the sequence of rules that led to its creation, and the tuples participating in the process, see Figure 2.

The last phase of the session is the redundancy analysis. It consists of sampling the variables at random positions in the database and attempting to eliminate them, exactly as in the core computation algorithm. Redundancy is then estimated as the number of variables that could be eliminated (mapped out with a proper endomorphism) related to the total number of variables tried. Besides the estimation, the precise redundancy measurement happens when the core is actually computed. Figure 2 shows redundant rows marked with grey color.

As described earlier, the core is computed iteratively as a sequence of nested endomorphisms. In order to keep track of the progress of the computation, the system produces output as in Table 1, showing the number of tuples and remaining nulls after each iteration, the eliminated redundancy (as percentage of nulls that were mapped out), and the timing in seconds. This output is very useful, since a big difference between the sizes of the universal solution and the core (like the one shown in Table 1, where more than half of the nulls are redundant) suggests, that there is some design flaw: e.g., some non-minimized constraint systematically introduces new nulls, or some frequent coincidence in the source data has not been treated by a specialized TGD, or the data engineer has "forgotten" to specify an EGD which in fact holds for the target instance, etc.

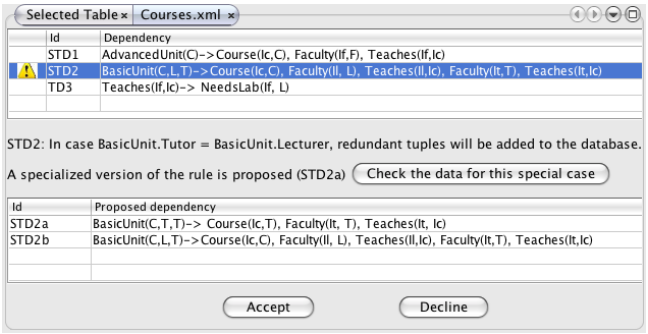

Figure 3: Dependency rewriting.

Table 1: Output by a core computation step

	#tuples	#nulls	shrink	timing (sec.)
univ. sol.	811	566	–	21
1 endo.	406	282	51%	63
2 endo.	398	274	3%	43
3 endo.	390	260	5%	39
4 endo.	382	258	1%	35
...	...	...	...	...
core	358	252	1%	25 (286 total)

After each phase, the statistics (number of nulls, redundancy, timings etc.) can be saved for future reference. After adjusting the dependencies or the chase order, the data transfer can be repeated — the tool supports automatic creation and dropping of the target schema, thus facilitating iterative development and ease of experimentation with different scenarios.

6. CONCLUSION

Core computation in the presence of target constraints is so far only available as a method for eliminating redundancy a posteriori from the target database in data exchange [4]. In this paper, we have presented our DEMo system, whose purpose is to assist the data engineer in designing expressive schema mappings that include target constraints, and to optimize the chase process, in order to reduce the redundancy in the target database. At the heart of this tool is a core computation implementation which, if applied to a small sample database, can give valuable insights on how superfluous tuples are introduced by a given data exchange scenario.

7. REFERENCES

[1] B. Alexe, L. Chiticariu and W. Tan, SPIDER: a schema mapPIng DEbuggeR. In *Proc. VLDB'06*, pages 1179–1182. 2006.

[2] B. Alexe, L. Chiticariu, R. J. Miller, and W. Tan, MUSE: a system for designing and understanding mappings. In *Proc. SIGMOD '08*, pages 1281–1284. 2008.

[3] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[4] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[5] G. Gottlob. Computing cores for data exchange: new algorithms and practical solutions. In *Proc. PODS'05*, pages 148–159.

[6] G. Gottlob and A. Nash. Data exchange: computing cores in polynomial time. In *Proc. PODS'06*, pages 40–49. 2006.

[7] G. Mecca, P. Papotti and S. Raunich. Core schema mappings. In *Proc. SIGMOD'09*.

[8] R. Pichler and V. Savenkov. Towards practical feasibility of core computation in data exchange. In *Proc. LPAR'08*, pages 62–78.