Scalable Data Exchange with Functional Dependencies

Bruno Marnette[1] Giansalvatore Mecca[2] Paolo Papotti[3]

[1] Oxford University Computing Laboratory, UK and INRIA Saclay, France
[2] Dipartimento di Matematica e Informatica – Università della Basilicata – Potenza, Italy
[3] Dipartimento di Informatica e Automazione – Università Roma Tre – Roma, Italy

ABSTRACT

The recent literature has provided a solid theoretical foundation for the use of schema mappings in data-exchange applications. Following this formalization, new algorithms have been developed to generate optimal solutions for mapping scenarios in a highly scalable way, by relying on SQL. However, these algorithms suffer from a serious drawback: they are not able to handle key constraints and functional dependencies on the target, i.e., equality generating dependencies (egds). While egds play a crucial role in the generation of optimal solutions, handling them with first-order languages is a difficult problem. In fact, we start from a negative result: it is not always possible to compute solutions for scenarios with egds using an SQL script. Then, we identify many practical cases in which this is possible, and develop a best-effort algorithm to do this. Experimental results show that our algorithm produces solutions of better quality with respect to those produced by previous algorithms, and scales nicely to large databases.

1. INTRODUCTION

Schema mappings are expressions that specify how an instance of a source database can be transformed into an instance of a target database. In recent years, they have received an increasing attention both from the research community and the tool market.

A *schema-mapping system* is used to support the process of generating and executing mappings in practical scenarios. It typically allows users to provide an abstract specification of the mapping as a set of correspondences among schema elements, specified through a friendly user-interface. Based on such specification, the mapping system will first generate a number of mappings – usually under the form of *tuple-generating dependencies (tgds)* [4] that correlate source tables with target tables; then, based on these mappings, an executable transformation, i.e., a runtime script in SQL or XQuery that can be practically used to run the mappings and generate solutions.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

After the seminal Clio papers [20, 21] introduced the key algorithmic techniques needed to generate mappings from correspondences, and executable scripts from mappings, a solid theoretical foundation for the *data-exchange problem* was laid in the framework of data exchange research [10, 12]. More recently, sophisticated algorithms have been proposed [17, 24] to take advantage of the theoretical background of data exchange and adopt a more principled approach to the generation of solutions. In fact, a data-exchange problem may have many different solutions. *Universal solutions* [10] were first identified as preferred solutions, since they contain only information that follows from the source instance and the mapping. Among these, the notion of *core solution* [12] was identified as the "optimal one", since it is the smallest among universal solutions. Second-generation mapping systems [17, 24] made a significant step forward by introducing algorithms that materialize core solutions by using runtime SQL scripts.

Despite their increasing maturity, these techniques still suffer from a significant limitation: they provide very limited support for target dependencies. While target constraints are recognized as an important feature of data exchange, they introduce a number of subtleties in the computation of solutions. Notice that target constraints typically come in two forms: target tgds, and target *equality-generating dependencies (egds)* [4]. These two kinds of constraints have received an unequal share of attention in schema-mappings research. Target tgds corresponding to foreign key constraints – by far the most common form – are handled quite nicely by mapping systems [21, 18]: in fact, the intuition of chasing foreign keys to generate source-to-target tgds is at the core of the original Clio mapping-generation algorithm.

On the contrary, state-of-the-art schema-mapping systems cannot handle target egds, i.e., there is currently no system to efficiently generate optimal solutions for mapping scenarios with key constraints and functional dependencies, as discussed in the next example.

Motivation Suppose we want to solve the mapping problem shown in Figure 1. This is a typical *data-fusion example* [5] which requires to merge together data from three different tables – possibly from three different original data sources – as shown in Figure 1.a: (*i*) a table about students and their birthdates; (*ii*) a table about employees and their salaries; (*iii*) a table about drivers and the cars they drive. The target schema contains two tables, one about persons, the second about cars. On these tables, we have two keys: *name* is a key for *Person*, while *plate* is a key for *Car*. Based on these requirements, it is natural to expect the generation of a solution as the one shown in Figure 1.b.

a. Source Tables

Student

name	*bdate*
Jim	1980
Ray	1990

Employee

name	*salary*
Jim	25,000
Mike	17,000

Driver

name	*carPlate*
Jim	abc123
Joe	abc123
Mike	cde345

b. Expected Solution

Person

name	*bdate*	*salary*	*carId*
Jim	1980	25,000	C1
Ray	1990	NULL	NULL
Mike	NULL	17,000	C2
Joe	NULL	NULL	C1

Car

id	*plate*
C1	abc123
C2	cde345

c. Pre-Solution (does not enforce keys)

Person

name	*bdate*	*salary*	*carId*
Jim	1980	NULL	NULL
Jim	NULL	25,000	NULL
Jim	NULL	NULL	C1
Ray	1990	NULL	NULL
Mike	NULL	17,000	NULL
Mike	NULL	NULL	C3
Joe	NULL	NULL	C2

Car

id	*plate*
C1	abc123
C2	abc123
C3	cde345

Figure 1: Mapping Person Data

However, there is currently no schema-mapping tool capable of generating an SQL script that materializes this expected instance.

Notice that the required mapping can be easily expressed as a data-exchange scenario, i.e., as a set of *source-to-target tgds* that specify how data should be moved from the source to the target, and a set of *target egds* that encode the required key constraints on the target, as follows. Note how the third tgd, m_3, "invents" a value to perform a vertical partition of the *Driver* source table:

$$m_1.\ \forall n, bd\colon \mathit{Student}(n, bd) \rightarrow \exists Y_1, Y_2 : \mathit{Person}(n, bd, Y_1, Y_2)$$
$$m_2.\ \forall n, s\colon \mathit{Employee}(n, s) \rightarrow \exists Y_1, Y_2 : \mathit{Person}(n, Y_1, s, Y_2)$$
$$m_3.\ \forall n, \mathit{plate}\colon \mathit{Driver}(n, \mathit{plate}) \rightarrow \exists Y_1, Y_2, Z\colon (\mathit{Person}(n, Y_1, Y_2, Z) \wedge \mathit{Car}(Z, \mathit{plate}))$$
$$e_1.\ \forall n, b, s, c, b', s', c' : \mathit{Person}(n, b, s, c) \wedge \mathit{Person}(n, b', s', c') \rightarrow (b = b') \wedge (s = s') \wedge (c = c')$$
$$e_2.\ \forall p, i, i' : \mathit{Car}(i, p) \wedge \mathit{Car}(i', p) \rightarrow (i = i')$$
$$e_3.\ \forall i, p, p' : \mathit{Car}(i, p) \wedge \mathit{Car}(i, p') \rightarrow (p = p')$$

In fact, the tgds above are exactly those that a schema mapping tool as Clio [21] or +Spicy [18] would generate. Formally speaking, since the desired solution is a universal solution for the mappings, it can be materialized by chasing the dependencies above, possibly with a post-processing step to minimize the solution. Even though there exist chase engines [23] that are capable of performing this task, as it will be shown in our experimental evaluation, they hardly scale to large databases. This is one of the reasons why schema-mapping systems generate SQL or XQuery scripts to perform the translation.

However, by using the script generation algorithms described in [17, 24], the best we can achieve is to generate a *pre-solution*, i.e., a solution for the tgds only, as shown in Figure 1.c. It is easy to see how the pre-solution is unsatisfactory from several points of view. In fact, it violates the required key constraints, and therefore it is not even a legal instance for the target. Moreover, it suffers from an unwanted *entity fragmentation* effect, in the sense that information about the same entities (e.g., *Jim*, *Mike* or the car *abc123*) is spread across several tuples, each of which gives a partial representation of the entity. If we take into account the usual dimensions of data quality [5], it should be clear that such an instance must be considered of very low quality in terms of compactness (or minimality). In fact, on large source instances, the level of redundancy due to entity fragmentation can seriously impair both the efficiency of the translation and the quality of answering queries over the target database.

The reason why state-of-the-art mapping systems perform so poorly on this example is that handling egds is a complicated task. In fact, we start from an expected negative result: it is not possible, in general, to enforce a set of egds using a first-order language such as SQL. This was conjectured first in [24] and is hardly surprising, since, as it can be seen by looking at Figure 1, chasing egds has the effect of equating values and merging tuples, effect that in general requires the power of recursion to be implemented.

Despite this, in this paper we argue that it is actually possible to generate solutions for egds in many practical cases, and make several important contributions towards this goal.

Contributions The main technical problem addressed in this paper is the following: given a mapping scenario containing a set of source-to-target tgds and a set of target egds, our goal is to generate an executable SQL script that can be run to generate good solutions for the given scenario. To do this:

(*i*) we introduce a best-effort rewriting algorithm that takes as input a scenario with s-t tgds and egds and, whenever this is possible, rewrites it into a new scenario without egds that can be efficiently implemented using an SQL script; in the paper, we show that our algorithm succeeds in many practical cases, including the example above; a key intuition behind the algorithm is that *source* constraints can be of high value in order to generate solutions that satisfy the required target constraints;

(*ii*) the rewriting takes advantage of a number of novel techniques; among these, a notion of *overlap tgds*, based on the idea of chasing egds at the formula level to avoid the introduction of unneeded null values, and a sophisticated skolemization strategy; in this way, we significantly push forward the expressibility of our SQL scripts;

(*iii*) then, we investigate the issue of generating optimal solutions, i.e., core universal solutions; we show that the rewriting algorithm to handle egds is modular in nature, since it can be coupled with the core-computation algorithms developed in [17, 24]; this process is definitely non trivial, due to the complex rewriting that we use for egds; in this way, we provide a much needed extension to the core-computation techniques in [17, 24];

(*iv*) finally, the techniques developed in the paper have been implemented in the +Spicy mapping system [18]; using the system, we provide a comprehensive evaluation of the algorithms presented in the paper, to show that they scale very well to large databases, and that they actually generates solutions that are much more compact than those generated by current mapping algorithms.

Applications This is the first algorithm that enables the generation of solutions for mapping scenarios with egds in a scalable way. To see the relevance of this achievement, consider that schema-mappings have been identified as a key component of several classes of applications that exchange or integrate data, for instance *ETL* [8], *object-relational mapping* [19], *data fusion* [5], and *schema-integration* [22]. It can also be seen that key constraints – and therefore egds

– play a very important role in all of these applications. Therefore, we believe that this paper represents a significant contribution towards the goal of reaching the full maturity of schema-mappings systems.

Outline The paper is organized as follows. Preliminary definitions are in Sections 2 and 3. The main algorithms of the paper are in Sections 4, 5, and 6. Experimental results are in Section 7. Finally, a discussion of related work is in Section 8.

2. BACKGROUND

We fix two disjoint sets: a set of *constants*, CONST, a set of *labeled nulls*, NULLS. Labeled nulls are used to "invent" values according to existential variables in tgd conclusions. We assume the standard definition of a relational schema, relational instance over CONST $\cup$ NULLS [10], functional dependency, and homomorphism [10] between instances, as detailed in Appendix A.

Tgds and Egds Given two schemas, $\mathbf{S}$ and $\mathbf{T}$, in the following by $\forall \overline{x} : \phi(\overline{x})$ we shall denote a conjunction of atomic formulas that may contain relational atoms in $\mathbf{S}$ or $\mathbf{T}$ and equations of the form $x_i = x_j$. A *source-to-target tgd (s-t tgd)* [4] is a first-order formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{x}, \overline{y}))$ where $\phi(\overline{x})$ ranges over $\mathbf{S}$ and $\psi(\overline{x}, \overline{y})$ is a conjunction of relational atoms over $\mathbf{T}$. An *equality generating dependency (egd)* [4] is a formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow (x_i = x_j))$. Examples of s-t tgds and egds were provided in Section 1. In the following, universal quantifiers will be omitted.

The Chase Given a vector of variables $\overline{v}$, an *assignment* for $\overline{v}$ is a mapping $a : \overline{v} \rightarrow$ CONST $\cup$ NULLS that associates with each universal variable a constant in CONST, and with each existential variable either a constant or a labeled null. Given a formula $\phi(\overline{x})$ with free variables $\overline{x}$, and an instance I, we say that I *satisfies* $\phi(a(\overline{x}))$ if $I \models \phi(a(\overline{x}))$.

Given instances I, J, during the *naive chase* a tgd $\phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{x}, \overline{y}))$ is fired for all assignments a such that $I \models \phi(a(\overline{x}))$; to fire the tgd, a is extended to $\overline{y}$ by injectively assigning to each $y_i \in \overline{y}$ a fresh null, and then adding the facts in $\psi(a(\overline{x}), a(\overline{y}))$ to J.

To chase an egd $\phi(\overline{x}) \rightarrow (x_i = x_j)$ over an instance J, for each assignment a such that $J \models \phi(a(\overline{x}))$, if $a(x_i) \neq a(x_j)$, the chase tries to equate the two values. We distinguish two cases: (*i*) both $a(x_i)$ $a(x_j)$ are constants; in this case, the chase procedure *fails*, since it attempts to identify two different constants; (*ii*) at least one of $a(x_i)$, $a(x_j)$ is a null – say $a(x_i)$ – in this case chasing the egd generates a new instance *J'* obtained from J by replacing all occurrences of $a(x_i)$ by $a(x_j)$. To give an example, consider egd e_1:

$$e_1.\ Person(n, b, s, c) \wedge Person(n, b', s', c') \rightarrow (b = b') \wedge (s = s') \wedge (c = c')$$

On the two tuples generated by chasing the tgds, $Person\ (Jim, 1980, N_3, N_4)$, $Person\ (Jim, N_5, 25,000, N_6)$, chasing the egd has two different effects:
(*i*) it replaces nulls by constants; in our example, it equates N_3 to the constant $25,000$, and N_5 to the constant 1980, based on the same value for the key attribute, *Jim*;
(*ii*) on the other side, the chase may equate nulls; in our example, it equates N_4 to N_6, to generate a single tuple $Person(Jim, 1980, 25,000, N_4)$.

Mapping Scenario Given a source schema $\mathbf{S}$ and a target schema $\mathbf{T}$, in this paper we concentrate on *mapping scenarios* consisting of a set of s-t tgds, Σ_{st}, and a set of target egds Σ_t that correspond to functional dependencies over $\mathbf{T}$. We find it useful to consider as part of the input scenario also a set of *source egds*, Σ_s, corresponding to source functional dependencies. In fact, as it will become apparent in the following sections, reasoning about source dependencies is a key component of our algorithms. In light of this, we will often write our scenarios as $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_s, \Sigma_{st}, \Sigma_t)$, and we will assume that any source instance I is valid with respect to Σ_s.

A target instance J is a solution of $\mathcal{M}$ and a source instance I, denoted $J \in \mathsf{Sol}(\mathcal{M}, I)$, iff $(I, J) \models \Sigma_{st} \cup \Sigma_t$. A solution J is *universal* [10], denoted $J \in \mathsf{USol}(\mathcal{M}, I)$, iff for every solution K there is an homomorphism from J to K. We call a *canonical solution* a solution obtained by chasing the dependencies in $\Sigma_{st} \cup \Sigma_t$ over I. Canonical solutions are universal solutions [10].

Given a scenario $\mathcal{M}$, and an instance I, a *core* [12] of a universal solution $J \in \mathsf{USol}(\mathcal{M}, I)$, denoted $\mathbf{C} \in \mathsf{Core}(\mathcal{M}, I)$, is a subinstance $\mathbf{C} \subseteq J$ such that there is a homomorphism from J to $\mathbf{C}$, but there is no homomorphism from J to a proper subinstance of $\mathbf{C}$. Cores of universal solutions are themselves universal solutions [12], and they are all isomorphic to each other. It is therefore possible to speak of *the core solution* as the "optimal" solution, in the sense that it is the solution of minimal size [12].

3. COMPUTING SOLUTIONS WITH SQL

We concentrate on mapping scenarios consisting of s-t tgds and egds. We do not consider target tgds, since in schema-mapping systems target tgds corresponding to foreign key constraints are typically rewritten into the s-t tgds using the techniques in [20, 21]. To generate solutions by means of SQL, we rewrite the original scenario as a set of *first-order rules*, i.e., essentially s-t tgds with negation in the premise and Skolem terms in the conclusion. Then, from these dependencies, we generate the needed SQL script.

First-Order Rules Given a set of variables $\overline{x}$, a *Skolem term* over $\overline{x}$ is a term of the form $t(\overline{x}) = f(t_1(\overline{x}), \ldots, t_k(\overline{x}))$ where f is a function symbol of arity k and $\{t_1(\overline{x}), \ldots, t_k(\overline{x})\}$ are either universal variables in $\overline{x}$ or in turn Skolem terms over $\overline{x}$. Skolem terms are used to create fresh labeled nulls on the target. Given an assignment of values a for $\overline{x}$, with the Skolem term above we (injectively) associate a labeled null $N_{f(t_1(a(\overline{x})), \ldots, t_k(a(\overline{x})))}$.

Given a source schema $\mathbf{S}$ and a target schema $\mathbf{T}$, an *FO-rule* is a dependency of the form $\varphi(\overline{x}) \rightarrow \psi(\overline{x})$ where $\varphi(\overline{x})$ is a first-order query over $\mathbf{S}$ with output tuple $\overline{x}$ and ψ is a conjunction of atoms of the form $R(t_1, \ldots, t_n)$ over $\mathbf{T}$ such that each term t_i is either a variable $t_i \in \{\overline{x}\}$ or a Skolem term over $\overline{x}$. In our setting, FO-rules will often have the standard form $\phi(\overline{x}) \wedge \neg\phi'(\overline{x}') \rightarrow \psi(\overline{x})$, where $\phi(\overline{x})$ is a conjunctive query, and $\phi'(\overline{x}')$ is a disjunction of conjunctive queries (possibly with equalities and inequalities). We shall call this a CQ$\wedge\neg$UCQ rule. Consider the *Person-Car* example in the previous Section; following is an FO-rule from its rewriting:

$$r_2 : Student(n_1, b_1) \wedge Employee(n_2, s_2) \wedge n_1 = n_2 \wedge \neg\exists n_3, p_3 : (Driver(n_3, p_3) \wedge n_1 = n_3) \rightarrow Person(n_1, b_1, s_2, f(n_1))$$

Given a source instance I over $\mathbf{S}$, a set of FO-rules Σ_{st}^{FO} can be chased to generate a *canonical target instance*. The naive chase straightforwardly generalizes to FO-rules (the pseudocode is detailed in the Appendix, Algorithm 1). Given a source instance I, we call $chase_{\Sigma_{st}^{FO}}(I)$ the instance obtained by chasing the FO-rules. Notice that the chase of FO-rules can be naturally implemented as an SQL script. Consider, for example, rule r_2 above. It can be implemented by the following SQL statement:

```
INSERT into Person
    SELECT s.n, s.b, e.s, append('f(',s.n,')')
    FROM   Student s, Employee e
    WHERE  s.n = e.n AND
           s.n NOT IN (SELECT d.n FROM Driver d)
```

First-Order Implementations Given a mapping scenario $\mathcal{M}$, our goal is to derive a finite set of FO-rules, Σ_{st}^{FO}, that represents an *FO-implementation* of $\mathcal{M}$. From the rules, we shall then derive an SQL script to generate solutions. However, the process of computing solutions has some subtleties. We want to emphasize an important difference between the original scenario and its FO-implementation.

The original scenario $\mathcal{M}$ includes a set of target egds Σ_t; since the chase of egds may fail, as discussed above, for some source instances there may be no solutions. On the contrary, an FO-implementation is simply a set of FO-rules, i.e., it corresponds to a new scenario $\mathcal{M}^{FO} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}^{FO})$ in which target egds have been dropped, and therefore it does not have a similar failure condition. To make things more symmetric, we say that a set of FO-rules *fails* on I with respect to a set of target constraints Σ_t iff it generates a target instance J that does not satisfy the target egds, i.e., $J \not\models \Sigma_t$. Otherwise we say that Σ_{st}^{FO} *succeeds* on I wrt Σ_t.

Based on this, in order to compute solutions for the original scenario $\mathcal{M}$, we do as follows: (*i*) we generate a set of FO-rules that represents an implementation of $\mathcal{M}$; (*ii*) we translate the rules into an SQL script; (*iii*) we run the script on the source instance to materialize the resulting instance, $J = chase_{\Sigma_{st}^{FO}}(I)$, into a temporary database; (*iv*) we also generate a number of additional boolean queries to check if the target egds are actually satisfied by J; if not, the script fails and rollbacks the overall transaction (*v*) otherwise, we return the output as a solution.

Completeness Ideally, we would like to be able to generate a *complete implementation* for every given scenario $\mathcal{M}$. We call a set of FO-rules, Σ_{st}^{FO}, a *complete FO-implementation* of $\mathcal{M}$ if, for every valid source instance I: (*i*) if $\mathcal{M}$ has solutions on I, then Σ_{st}^{FO} succeeds on I and $chase_{\Sigma_{st}^{FO}}(I)$ is a universal solution for $\mathcal{M}$ over I, i.e., $chase_{\Sigma_{st}^{FO}}(I) \in \mathsf{USol}_{\mathcal{M}}(I)$; (*ii*) if $\mathcal{M}$ has no solutions on I, i.e., $\mathsf{Sol}(\mathcal{M}, I) = \emptyset$, then Σ_{st}^{FO} fails on I wrt Σ_t.

Complete FO-implementations are desirable since they allow computing a universal solution whenever there exists one and to identify the source instances that, together with Σ_{st}, contradict the target egds in Σ_t. With this in mind, a natural question is whether complete FO-implementations always exist. Unfortunately, this is not the case for scenarios with target egds. In fact, we can state the following result, which was first conjectured in [24].[1] A sketch of the proof is reported in Appendix C.

THEOREM 3.1. *There is a scenario $\mathcal{M} = (\boldsymbol{S}, \boldsymbol{T}, \Sigma_{st}, \Sigma_t)$ where Σ_t is a set of functional dependencies over $\boldsymbol{T}$ such that no complete FO-implementation exists for $\mathcal{M}$.*

[1]A similar result was reported in [9] for LAV mappings in data integration systems.

To give an intuition about the intrinsic complexity of handling egds with FO-languages, consider that by progressively equating nulls during the chase, egds may generate large blocks of facts that are connected with each other via labeled nulls. Intuitively, such connected component of unbounded width in the graph of facts are impossible to capture using a first-order language such as SQL. Notice also that this behavior does not happen without egds, since s-t tgds always generate fact-blocks of bounded size wrt the size of tgd conclusions.

A Best-Effort Approach Theorem 3.1 above leaves us no hope of defining an algorithm that always returns a complete implementation for a mapping scenario with target egds. In spite of this, we have two crucial observations: (*i*) first, in practical cases many scenarios do have FO-implementations; (*ii*) second, by reasoning on the *source constraints* we can find good implementations for these cases. Consider again the example above. As soon as additional constraints are imposed on the source – as it typically happens in real-life mapping scenarios – the emergence of critical cases becomes less likely.

In our example, since the tgd actually performs a vertical partition of the *Driver* table, we might want to consider the source functional dependency $Driver.name \rightarrow Driver.plate$, stating that each driver drives at most one car, without which the vertical partition does not make sense (a driver that drives more than one car would cause a violation of the key constraint on the car id). If the source functional dependency holds, it is actually possible to generate a complete implementation, as follows. Notice how the rule makes use of non-standard Skolem terms in order to capture the semantics of the target egds (as discussed in Section 5):

$$r_1.\ Driver(n, p) \rightarrow Person(n, f(p)) \wedge Car(f(p), p)$$

In the following sections, we develop a best-effort algorithm that, given a mapping scenario $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_s, \Sigma_{st}, \Sigma_t)$, generates a sound FO-implementation for $\mathcal{M}$. A set of FO-rules, Σ_{st}^{FO}, is a *sound FO-implementation* of $\mathcal{M}$ if, for every valid source instance I, Σ_{st}^{FO} succeeds on I, then $chase_{\Sigma_{st}^{FO}}(I)$ is a universal solution for $\mathcal{M}$ over I, i.e., $chase_{\Sigma_{st}^{FO}}(I) \in \mathsf{USol}_{\mathcal{M}}(I)$.

Despite the fact that our algorithm does not always return complete implementations, it has two nice properties that make it quite effective in order to handle scenarios with egds.

First, our solutions turn out to be complete in most practical cases. We prove that, given a sound implementation Σ_{st}^{FO} for $\mathcal{M}$, it is decidable if it represents also a complete implementation for $\mathcal{M}$, and provide an algorithm for this check.

Second, and more important, whenever Σ_{st}^{FO} turns out to be complete, we can derive from it an alternative implementation, Σ_{st}^{FO*}, that generates core solutions for $\mathcal{M}$. Notice, in fact, that complete implementations return universal solutions that are not guaranteed to be minimal. Computing core solutions adds further complexity to the problem and is considered here as a separate step (as discussed in Section 6).

The next sections are devoted to the development of the rewriting algorithm. Recall that chasing egds has the effect of reducing the number of nulls in the target instance, in two ways. The first one is by replacing one null by a constant; the second one is by equating one null to another null. We simulate these two effects in our FO-rules by two different techniques:

(i) the first technique is concerned with discovering *overlaps* among tgds in order to properly equate nulls to constants;

(ii) the second technique tries to discover an appropriate skolemization strategy for the tgds obtained at the previous step in order to properly equate null values.

4. DISCOVERING OVERLAPS

Two atoms overlap when they may generate facts for a relation having the same key and different tuple values. More formally, given a set of atoms $\varphi(\overline{x}, \overline{y})$, we say that two atoms $R(t_1, \ldots, t_n)$, $R(t'_1, \ldots, t'_n)$ in $\varphi(\overline{x}, \overline{y})$ generate an *overlap* for a functional dependency $R.\langle i_1 \ldots i_k \rangle \rightarrow j$ if, for each $l = 1, \ldots, k$, either t_{i_l} and t'_{i_l} are both universal variables, or they are the same existential variable, i.e., $t_{i_l} = t'_{i_l} \in \overline{y}$. Given two tgds $m_1 : \phi_1(\overline{x}_1) \rightarrow \exists \overline{y}_1(\psi_1(\overline{x}_1, \overline{y}_1))$, $m_2 : \phi_2(\overline{x}_2) \rightarrow \exists \overline{y}_2(\psi_2(\overline{x}_2, \overline{y}_2))$, we say that they overlap whenever there are overlaps in $\psi_1(\overline{x}_1, \overline{y}_1) \cup \psi_2(\overline{x}_2, \overline{y}_2)$.

Consider the *Person-Car* example in Section 1: it can be seen that mappings m_1, m_2 generate an overlap on the *Person* atoms (in both cases the key, n, is universal). Our intuition is that it is possible to rewrite the mappings into a new tgd that directly generates the target atoms that are produced by chasing the original tgds first and then the egd, as follows:

$$o_1. \; \mathit{Student}(n_1, \mathit{bdate}) \wedge \mathit{Employee}(n_2, \mathit{salary}) \wedge n_1 = n_2 \rightarrow \mathit{Person}(n_1, \mathit{bdate}, \mathit{salary}, Y_0)$$

Mapping o_1 above is called an *overlap tgd*. It is interesting to note that the conclusion of o_1 may be constructed using the following intuitive algorithm: (i) take the conjunction of the conclusions of the two original tgds; (ii) "chase" the atoms according to the egd to equate existential and universal variables. In essence, we are chasing at the formula level to incorporate the semantics of the egds into a set of new s-t tgds that are easier to chase at the instance level.

Algorithm 3 in the Appendix generates the actual overlap tgds. It takes as input a mapping scenario $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$, and generates a new set tgds, $\Sigma^{ovl}_{\mathcal{M}}$, by recursively chasing overlaps. In our example, besides o_1 above, the algorithm will also generate the following tgds:

$$o_2. \; \mathit{Student}(n_1, b_1) \wedge \mathit{Employee}(n_2, s_2) \wedge \mathit{Driver}(n_3, p_3) \wedge n_1 = n_2 \wedge n_2 = n_3 \rightarrow \mathit{Person}(n_1, b_1, s_2, C_3) \wedge \mathit{Car}(C_3, p_3)$$

$$o_3. \; \mathit{Student}(n_1, b_1) \wedge \mathit{Driver}(n_3, p_3) \wedge n_1 = n_3 \rightarrow \mathit{Person}(n_1, b_1, S_3, C_3) \wedge \mathit{Car}(C_3, p_3)$$

$$o_4. \; \mathit{Employee}(n_2, s_2) \wedge \mathit{Driver}(n_3, p_3) \wedge n_2 = n_3 \rightarrow \mathit{Person}(n_2, B_3, s_2, C_3) \wedge \mathit{Car}(C_3, p_3)$$

However, the union of the original tgds and these new tgds is not logically equivalent to the original scenario. To see why, consider that, by replacing variables, egds do not simply add new tuples, but typically also *remove* some existing ones: consider for example the following two tuples: $\mathit{Person}(\mathit{Jim}, 1980, N_0,\ N_1)$, $\mathit{Person}(\mathit{Jim}, N_4,\ 25{,}000,\ N_5)$; when the chase enforces the key constraint over *Person*, a new tuple is generated, $\mathit{Person}(\mathit{Jim}, 1980,\ 25{,}000,\ N_1)$, but at the same time the original tuples are removed from the solution. Therefore, to correctly simulate the effect of egds by means of overlap tgds, a further rewriting step is necessary: we need to rewrite tgds so that they fire only when no overlap can be generated. To do this, we use negation in the premise, as shown in Algorithm 4 in the Appendix. This generates a set of CQ∧¬UCQ rules, ADDNEG$(\Sigma_{st} \cup \Sigma^{ovl}_{\mathcal{M}})$. To give an example, the procedure above rewrites tgd m_1 as follows:

$$m'_1. \; \mathit{Student}(n_1, b_1) \wedge \neg(\mathit{Employee}(n_2, s_2) \wedge n_1 = n_2) \wedge \neg(\mathit{Employee}(n_2, s_2) \wedge \mathit{Driver}(n_3, p_3) \wedge n_1 = n_2 \wedge n_2 = n_3) \wedge \neg(\mathit{Driver}(n_3, p_3) \wedge n_1 = n_3) \rightarrow \mathit{Person}(n_1, b_1, S_1, C_1)$$

By doing this, we have obtained a new scenario, with a new set of s-t tgds, ADDNEG$(\Sigma_{st} \cup \Sigma^{ovl}_{\mathcal{M}})$. This new scenario is logically equivalent [11] to the original one, i.e., the two scenarios have the same solutions for any source instance, as stated by the following theorem.

THEOREM 4.1. *Given a scenario $\mathcal{M} = (\boldsymbol{S}, \boldsymbol{T}, \Sigma_{st}, \Sigma_t)$, the scenario $\mathcal{M}^{ovl} = (\boldsymbol{S}, \boldsymbol{T}, \text{ADDNEG}(\Sigma_{st} \cup \Sigma^{ovl}_{\mathcal{M}}), \Sigma_t)$ is logically equivalent to $\mathcal{M}$.*

In light of this, in our rewriting we assume that, as a first step, $\mathcal{M}$ is always rewritten as $\mathcal{M}^{ovl}$, and that further rewritings are performed on $\mathcal{M}^{ovl}$. Note that, in the worst case, the number of new dependencies generated by the algorithm is exponential in the size of the original tgds. In fact, in the case of n s-t tgds that overlap on the same key, the algorithm would generate $O(2^n)$ overlap tgds.

5. SKOLEMIZATION STRATEGY

Once the original s-t tgds have been rewritten wrt to overlaps, we need to find an appropriate way to skolemize existential variables in order to generate the FO-rules that will be used as a basis for the final SQL script. Recall that the skolemization strategy must be chosen in such a way to appropriately equate nulls. We know that this is not always doable. In fact, the algorithm introduced in this Section may fail.

Given a tgd, the standard skolemization strategy would generate a Skolem term for each existential variable whose arguments are all universal variables occurring in the tgd conclusion. Our goal is to develop a novel skolemization scheme capable of capturing the semantics of egds.

Determinations In order to find the right skolemization for an existential variable, our algorithm looks for *determinations* for that variable. Given a tgd $m : \phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{x}, \overline{y}))$, and an existential variable $y_i \in \overline{y}$, a *determination* for y_i is a pair $[d_j, \overline{x}_k]$, where (i) d_j is a functional dependency $R.\langle i_1 \ldots i_k \rangle \rightarrow j$ in Σ_t; (ii) for some atom $R(t_1, \ldots, t_n)$ in $\psi(\bar{x}, \overline{y})$ it is the case that $\{t_{i_1}, \ldots, t_{i_k}\} = \overline{x}_k \subseteq \overline{x}$, and $t_j = y$. Intuitively, a determination tells us that, according to the egds, the value of the existential variable y_i functionally depends on the values of the universal variables in x_k; in fact, with each determination $[d_j, \overline{x}_k]$ we associate a Skolem term of the form $f_{d_j}(\overline{x}_k)$. In order to take care of variable occurrences for which there is no relevant functional dependency, we also include the *standard determination*, $[(m, y_i), \overline{x}]$, to which we associate the standard Skolem term $f_{(m,y_i)}(\overline{x})$.

Our algorithm is based on two main intuitions. The first one is that determinations can be in many cases minimized by looking at the source functional dependencies, in order to make them more compact. The second one is that, albeit an existential variable has multiple determinations, in many cases it is possible to find a *most general determination*, i.e., a determination such that by satisfying it, all other determinations – i.e., all other functional dependencies – are satisfied.

We illustrate these ideas by means of the following example, where the source table *PaperAuthors* lists names of authors of papers along with the order in which they appear:

$$\begin{array}{l} m.\ PaperAuthors(title, name, order) \rightarrow Author(A, name) \\ \quad \wedge AuthorPaper(A, P, order) \wedge Paper(P, title) \\ d_1 : Author.2 \rightarrow Author.1 \quad d_2 : Paper.2 \rightarrow Paper.1 \\ d_3 : PaperAuthors.2 \rightarrow PaperAuthors.3 \end{array}$$

The standard Skolem term for the existential variables is $f_{(m,y_i)}(title, name, order)$, which corresponds to the standard determination $[(m, y_i), \{title, name, order\}]$. However, if we consider functional dependency d_3 on the source, we can minimize this as $[(m, y_i), \{title, name\}]$, since the value of variable *order* is functionally dependent on *name*.

Consider now variable A. It has two occurrences and two determination. One occurrence is in relation *AuthorPaper*, for which there are in fact no functional dependencies, and we take the standard (minimized) determination, $[(m, A), \{title, name\}]$. However, the second occurrence is in relation *Author*, for which we have an egd, d_1. The egd suggests that this occurrence should depend on variable *name* only, which gives us a determination $[d_1, \{name\}]$. We need therefore to reconcile the two determinations in order to uniquely identify a skolemization strategy for A. However, in this case, we may easily see that the set of variables $\{name\}$ functionally depends on $\{name, title\}$. The intuition behind this is the following: consider two different instantiations of the tgd m; if the variable A has equal values whenever variable *name* has equal values, then it certainly has equal values whenever both *name* and *title* have equal values. We therefore select $[d_1, \{name\}]$ as the most general determination for A, since enforcing it guarantees that all constraints on the variable will be satisfied. Thus, the chosen Skolem term for A will be $f_{d_1}(name)$.

Minimizing Determinations We make extensive use of the standard implication algorithms for functional dependencies in order to reason about determinations. Recall [1] that, given a set of functional dependencies over a relation R, there exists a linear-time algorithm to compute the *closure*, $\bar{A}^+$, of set of attributes $\bar{A}$; $\bar{A}^+$ represents the set of all attributes that functionally depend on $\bar{A}$, directly or transitively.

We extend this algorithm in order to work on the universal variables of a tgd. More specifically, given a tgd $m : \phi(\overline{x}) \rightarrow \exists\overline{y}(\psi(\overline{x}, \overline{y}))$, consider a subset $\overline{x_i}$ of $\overline{x}$. We compute the *closure*, $\overline{x_i}^+$ of $\overline{x_i}$ by the following fixpoint algorithm: (*i*) initialize $\overline{x_i}^+ = \overline{x_i}$; (*ii*) for each source functional dependency $R.\langle i_1 \dots i_k\rangle \rightarrow j$ such that, for some atom $R(t_1, \dots, t_n)$ in m, it is the case that $\{t_{i_1}, \dots, t_{i_k}\} \subseteq \overline{x_i}^+$, and $t_j \in \overline{x}$, add t_j to $\overline{x_i}^+$, until a fixpoint is reached.

We say that a set of variables $\overline{x_j}$ in a tgd m *functionally depends* on a set of variables $\overline{x_i}$ if $\overline{x_j} \subseteq \overline{x_i}^+$. Given a set of universal variables, $\overline{x_i}$ we may also introduce a notion of *minimization*, $\text{MIN}(\overline{x_i})$, as any minimal subset such that $\text{MIN}(\overline{x_i})^+ = \overline{x_i}^+$.

Skolemization Given an existential variable in a tgd m, we find all of its determinations, one for each occurrence in m; in order to find the most general determination, if it exists, we need a way to compare two (minimized) determinations. To do this, we introduce a partial order among determinations, as follows. We say that a determination $[d_1, \overline{x}_1]$ is *more general* than a determination $[d_2, \overline{x}_2]$, in symbols $[d_1, \overline{x}_1] \leq [d_2, \overline{x}_2]$, if there exist minimizations $\text{MIN}(\overline{x_1})$, $\text{MIN}(\overline{x_2})$ such that $\text{MIN}(\overline{x_1})$ functionally depends on $\text{MIN}(\overline{x_2})$.

Whenever a variable has multiple determinations, our algorithm looks for a greatest lower-bound according to the partial order defined above. The pseudo-code is reported in Algorithm 5 in the Appendix. Notice that the algorithm may fail and return $\perp$ whenever it is unable to find a most general determination for a variable.

Consider again the *Driver* example in Section 3:

$$\begin{array}{l} m_3.\ Driver(n, p) \rightarrow Person(n, Y) \wedge Car(Y, p) \\ d_1.\ Person.1 \rightarrow Person.2 \quad d_2.\ Car.2 \rightarrow Car.1 \end{array}$$

In this case, variable Y has two determinations: $[d_1, \{n\}]$, and $[d_2, \{p\}]$. But, by the source functional dependency, we know that the values of the car plate depend functionally on those of the driver name; thus, the most general determination for Y is $[d_2, \{p\}]$. This is what the Skolem function in the rewriting does.

If the algorithm succeeds, we have identified a sound skolemization strategy, SKOLEMIZE, that takes a tgd, and associates with each existential variable the Skolem term $f_{d_j}(\overline{x}_k)$ corresponding to its most general determination, $[d_j, \overline{x}_k]$.

6. GETTING TO THE CORE

We are now ready to introduce our main results. Given a mapping scenario, $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_s, \Sigma_{st}, \Sigma_t)$, our algorithms will either fail, or generate a set of FO-rules

$$\Sigma_{st}^{FO} = \text{SKOLEMIZE}(\text{ADDNEG}(\Sigma_{st} \cup \Sigma_{\mathcal{M}}^{ovl}))$$

Our first result is that Σ_{st}^{FO} is a sound implementation of $\mathcal{M}$, as stated below (sketches of the proofs are in Appendix C).

THEOREM 6.1. *Σ_{st}^{FO} is a sound implementation of $\mathcal{M}$.*

A natural question is whether Σ_{st}^{FO} is also complete. It is possible to prove that the property of being complete is decidable, i.e., once Σ_{st}^{FO} has been computed, we can also establish if it is complete. Recall that Σ_{st}^{FO} is a set of CQ∧¬UCQ rules. In light of this, we have the following important decidability result.

THEOREM 6.2. *The following problem is decidable: given a sound FO-implementation Σ_{st}^{FO} of a scenario $\mathcal{M}$ where the body of each FO-rule is in CQ∧¬UCQ, is Σ_{st}^{FO} complete ?*

It is worth noting that sound implementations in most cases turn out to be complete. In fact, for all examples discussed in the paper, including those used in the experimental evaluation, we were able to generate complete implementations. Even more important, complete implementations can be used as a basis for the generation of core solutions. Recall that FO-implementations are guaranteed to return universal solutions for a mapping scenario with egds, but not necessarily core ones.

We say that a set of FO-rules is a *core implementation* for $\mathcal{M}$ if it is a complete implementation that always returns core solutions for $\mathcal{M}$. We show that the core-computation algorithms developed for scenarios without egds [17, 24] can be used as building-blocks to turn a complete implementation Σ_{st}^{FO} into a core implementation Σ_{st}^{FO*}.

THEOREM 6.3. *Given a complete implementation for $\mathcal{M}$ it is always possible to derive a core implementation for $\mathcal{M}$*

The latter result is absolutely non trivial and it is based on a sophisticated encoding of the egds in the core scenario. The main intuition is that, by relying on a complete implementation Σ_{st}^{FO} of $\mathcal{M}$, for each source instance it is possible to "materialize" all functional dependencies in a universal solution. To do this, we introduce an additional relation symbol F_d for each functional dependency

$d : R\langle i_1, \dots, i_k\rangle \rightarrow j$ in Σ_t; then, for each rule $\phi(\bar{x}) \rightarrow \psi(\bar{x})$ in Σ_{st}^{FO} that contains an atom $R(t_1, \dots, t_n)$ in its conclusion, we introduce a tgd of the form $\phi(\bar{x}) \rightarrow F_\alpha(t_{i_1}, \dots, t_{i_k}, t_j)$. This allows for the treatment of egds using core-oriented rewritings that were conceived for s-t tgds only. Further details are reported in Appendix C.

7. EXPERIMENTAL RESULTS

The algorithms introduced in the paper have been implemented in the working prototype of the +SPICY system. In this section we study the performance of our rewriting algorithm on mapping scenarios of various kinds and sizes. We show that the rewriting algorithm efficiently computes solutions of quality for scenarios with egds, even for large databases and high numbers of tgds.

For our experiments we selected 8 scenarios, called s_a, s_b, s_c, s_d, s_{25}, s_{50}, s_{75}, s_{100}, as detailed in Appendix D.

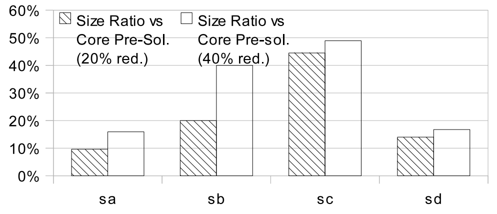


Figure 2: Size Reduction in Solutions

Quality of Solutions (Figure 2) One of the main claims of this paper is that egds significantly improve the quality of solutions with respect to scenarios that do not take them into consideration. To support this claim, we concentrate on the compactness dimension of data quality. For experiments s_a–s_d, we generated two different SQL scripts. One script was generated by our rewriting algorithm to generate core solutions for the egds. The second script was generated to materialize a core pre-solution, i.e., a core solution for the s-t tgds only. Differently from the first script, this solution does not satisfies the egds. Then, we measured the sizes of the two solutions in terms of tuples and reported the ratio.

Since egds are triggered when equal values are generated in different target tuples, we are interested in testing how the size ratio changes in presence of different levels of redundancy in the source instance; here, redundancy means the probability that the same atomic value appears in more than one tuple. For each scenario we generated two synthetic source instances based on a pool of values of decreasing size. This generates different levels of redundancy (20% and 40%) in the source database.

Results are in Figure 2. It can be seen that in all experiments the egds brought a significant reduction in the size of solutions. As it was to be expected, such reduction increases as the redundancy in the source instance increases, since there is a higher probability that different tuples have equal values on key attributes and can be merged together.

An obvious question is at what cost this improvement of quality comes. In the following, we discuss the efficiency of the rewriting algorithm along two different dimensions. First, we study the cost of computing solutions for source instances of increasing size. Then, we study the cost for scenarios of increasing size.

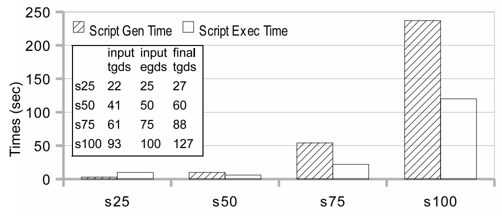
Scalability wrt Large Source Instances (Figure 3) To study how the algorithm performs on databases of large sizes, we considered scenarios s_a–s_d and generated source instances of increasing size (100k, 500k, and 1M tuples). Then, we computed solutions and reported execution times in Figure 3.

Notice that, while in this paper we restrict ourselves to SQL, and therefore FO-logic, a number of recent works about mappings (see, for example [15]) have undertaken the approach of coupling SQL for database access with a controlled form of recursive control-logic implemented in a procedural programming language, typically Java. These works show that it is possible in some cases to achieve good scalability even in the execution of recursive queries. Unfortunately, this is not true in the setting we consider in this paper, i.e., chasing egds and/or generating core solutions. To show this, we have compared computing times of the rewriting algorithm to those of a custom chase engine for egds [23] that uses a combination of SQL and Java to generate solutions. We fixed a timeout of one hour. If one experiment was not completed by that time, we stopped it.

Figure 3.a shows computing times for egd-compliant core solutions generated by our rewriting. Figure 3.b shows computing times for core pre-solutions generated without considering egds. Finally, Figure 3.c shows computing times for the chase engine.

It is easy to see that the chase engine hardly scales to large databases. While execution times for the SQL scripts scaled nicely to 1M tuples, in order to meet the timeout with the chase engine we had to reduce the size of instances to 1k, 5k, 10k. Even so, computing times were significantly higher. In essence, our experiments show that, using the custom chase engine, even simply chasing a set of egds – let alone computing a core solution – may take quite a long time. To see why, note that during the chase, whenever a null is mapped to some other value, it must be replaced everywhere in the database; this may require a high number of queries. In essence, once the nulls have been generated, removing them in a scalable way becomes very hard. This confirms the impression that, whenever this is possible, the SQL-based approach should be preferred to the actual chase of the egds.

It is also interesting to note that execution times for the rewriting on all scenarios were comparable to those needed to generate the pre-solution, so that we can conclude that the increased quality discussed in previous paragraph comes at a very acceptable cost.



	input tgds	input egds	final tgds
s25	22	25	27
s50	41	50	60
s75	61	75	88
s100	93	100	127

Figure 4: Execution Times for Large Scenarios

Scalability wrt Large Scenarios (Figure 4) In the table in Figure 4, we report several values for scenarios $s_{25}, s_{50}, s_{75}, s_{100}$: (a) the number of input tgds; (b) the number of input egds; (c) the number of final tgds, after overlaps have been processed. Then, we report two different times. Since we wanted to study how the rewriting phase scales with the size of the scenario, we measured the time needed to generate the SQL script. Finally, we report execution times in seconds for source databases of 100K tuples.

As it was to be expected, the cost of the script genera-

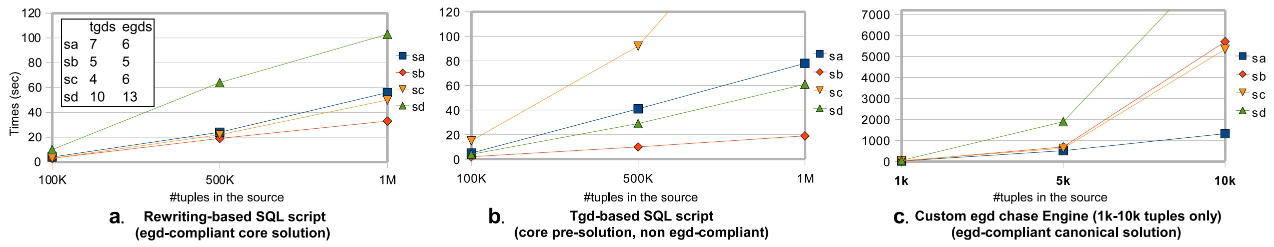


Figure 3: Execution Times for Source Instances of Increasing Size

tion phase increased exponentially with the number of tgds. This is due to the fact that the algorithm has in general to inspect an exponential number of possible overlaps. On the contrary, the actual script execution times remained pretty low, even for scenario s_{100}, where more than 120 tgds were processed.

8. RELATED WORK

As discussed in the previous sections, the notion of a *data exchange problem* was originally introduced in [10] and the properties of core solutions were first studied in [12]. Sophisticated polynomial algorithms for core computation have been given in [12] first, and then in [13, 23, 16]. These algorithms assume that a specialized engine is used to post-process a canonical solution, find endomorphisms and generate the core. Rewriting algorithms to generate core solutions by means of SQL scripts have been given in [17, 24]. As it was already discussed, these approaches are not applicable to scenarios with target dependencies.

More recently, a rewriting algorithm for mappings that also considers target egds [14] has been proposed. However, in this case the purpose of the rewriting is quite different, since it aims at optimizing and normalizing the input constraints; intuitively, the goal is to minimize the constraints to make them easier to handle and to improve the quality of solutions. The rewriting of [14] is therefore independent from the one proposed in this paper and the two can be easily combined.

The complexity of dealing with functional dependencies has also been studied in the context of data integration, both for LAV [9, 2] and GAV [7] mappings. In that context, query rewriting techniques were developed to compute query rewritings in presence of functional dependencies.

The presence of key constraints plays a key role also in the data fusion literature [5]. However, these works adopt a different approach: they merge data as a separate step from the data translation and do not consider the presence of labeled nulls (i.e., generated values).

An early attempt to partially incorporate key constraints in mapping systems has been proposed by [6]. There, users are supposed to provide specialized inputs so that the mapping algorithm can handle keys.

Acknowledgments Many thanks go to Vadim Savenkov and Reinhard Pichler for making available their chase engine. The second author would like to thank Assunta Lacerra, for her precious support during the writing of the paper. This work has been partially funded by the Engineering and Physical Sciences Research Council (EPSRC) and the Advanced European Research Council grant Webdam.

9. REFERENCES

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

[2] F. N. Afrati and N. Kiourtis. Computing Certain Answers in the Presence of Dependencies. *Inf. Systems*, 35(2):149–169, 2010.

[3] B. Alexe, W. Tan, and Y. Velegrakis. Comparing and Evaluating Mapping Systems with STBenchmark. *PVLDB*, 1(2):1468–1471, 2008.

[4] C. Beeri and M. Vardi. A Proof Procedure for Data Dependencies. *J. of the ACM*, 31(4):718–741, 1984.

[5] J. Bleiholder and F. Naumann. Data fusion. *ACM Comp. Surv.*, 41(1):1–41, 2008.

[6] L. Cabibbo. On Keys, Foreign Keys and Nullable Attributes in Relational Mapping Systems. In *EDBT*, 2009.

[7] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data integration under integrity constraints. *Inf. Systems*, 29(2):147–163, 2004.

[8] S. Dessloch, M. A. Hernandez, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, 2008.

[9] O. M. Duschka and A. Y. Levy. Recursive Plans for Information Gathering. In *IJCAI*, pages 778–784, 1997.

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

[11] R. Fagin, P. Kolaitis, A. Nash, and L. Popa. Towards a Theory of Schema-Mapping Optimization. In *ACM PODS*, 2008.

[12] R. Fagin, P. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *ACM TODS*, 30(1):174–210, 2005.

[13] G. Gottlob and A. Nash. Efficient Core Computation in Data Exchange. *JACM*, 55(2):1–49, 2008.

[14] G. Gottlob, R. Pichler, and V. Savenkov. Normalization and Optimization of Schema Mappings. *PVLDB*, 2(1):1102–1113, 2009.

[15] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *VLDB*, 2007.

[16] B. Marnette. Generalized Schema Mappings: From Termination to Tractability. In *ACM PODS*, 2009.

[17] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings. In *SIGMOD*, 2009.

[18] G. Mecca, P. Papotti, S. Raunich, and M. Buoncristiano. Concise and Expressive Mappings with +Spicy. *PVLDB*, 2(2):1582–1585, 2009.

[19] S. Melnik, A. Adya, and P. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD*, 2007.

[20] R. J. Miller, L. M. Haas, and M. A. Hernandez. Schema Mapping as Query Discovery. In *VLDB*, 2000.

[21] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *VLDB*, 2002.

[22] A. D. Sarma, X. Dong, and A. Y. Halevy. Bootstrapping Pay-As-You-Go Data Integration Systems. In *SIGMOD*, pages 861–874, 2008.

[23] V. Savenkov and R. Pichler. Towards Practical Feasibility of Core Computation in Data Exchange. In *LPAR*, 2008.

[24] B. ten Cate, L. Chiticariu, P. Kolaitis, and W. C. Tan. Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries. *PVLDB*, 2(1):1006–1017, 2009.

APPENDIX

A. DATA MODEL

In this section we recall some well-known definition about the relational data model, taken from data-exchange literature [10].

We fix two disjoint sets: a set of *constants*, CONST, a set of *labeled nulls*, NULLS. We also fix a set of *labels* $\{A_0, A_1, \ldots\}$, and a set of *relation symbols* $\{R_0, R_1, \ldots\}$. With each relation symbol R we associate a *relation schema* $R(A_1, \ldots, A_k)$. A *schema* $\mathbf{S} = \{R_1, \ldots, R_n\}$ is a collection of relation schemas. An *instance* of a relation schema $R(A_1, \ldots, A_k)$ is a finite set of tuples of the form $R(A_1 : v_1, \ldots, A_k : v_k)$, where, for each i, v_i is either a constant or a labeled null.

An *instance* of a schema $\mathbf{S}$ is a collection of instances, one for each relation schema in $\mathbf{S}$. In the paper, we interchangeably use the positional and non positional notation for tuples and facts; also, with an abuse of notation, we will often blur the distinction between a relation symbol and the corresponding instance. A *ground* instance is an instance I without labeled nulls.

Given two disjoint schemas, $\mathbf{S}$ and $\mathbf{T}$, we shall denote by $\langle \mathbf{S}, \mathbf{T} \rangle$ the schema $\{S_1 \ldots S_n, T_1 \ldots T_m\}$. If I is an instance of $\mathbf{S}$ and J is an instance of $\mathbf{T}$, then the pair $\langle I, J \rangle$ is an instance of $\langle \mathbf{S}, \mathbf{T} \rangle$.

Given a relation schema $R(A_1, \ldots, A_k)$, a *functional dependency* is an expression of the form $R.\bar{A} \rightarrow \bar{B}$, where $\bar{A}$ and $\bar{B}$ are sets of attributes in $\{A_1, A_2, \ldots A_k\}$. Key constraints are functional dependencies such that $\bar{B} = \{A_1, A_2, \ldots A_k\}$. As an alternative, we also write functional dependencies using the positional notation $R.[i_0, \ldots, i_n] \rightarrow j$.

Given two instances J, J' over a schema $\mathbf{T}$, a *homomorphism* $h : J \rightarrow J'$ is a mapping from *dom(J)* to *dom(J')* such that for each $c \in$ *const(J)*, $h(c) = c$, and for each tuple $t = R(A_1 : v_1, \ldots, A_k : v_k)$ in J it is the case that $h(t) = R(A_1 : h(v_1), \ldots, A_k : h(v_k))$ belongs to J'. h is called an *endomorphism* if $J' \subseteq J$; if $J' \subset J$ it is called a *proper endomorphism*.

We say that two instances J, J' are *homomorphically equivalent* if there are homomorphisms $h : J \rightarrow J'$ and $h' : J' \rightarrow J$. Note that a conjunction of atoms may be seen as a special instance containing only variables. The notion of homomorphism extends to formulas as well.

B. PSEUDO-CODE

Chasing FO-Rules Given a source instance I over $\mathbf{S}$, FO-rules are executed by running the *naive-chase procedure* to generate a *canonical target instance*. The chase essentially fires rules to generate atoms in the target whenever a rule premise is satisfied by I, as detailed in Algorithm 1.

Algorithm 1 CHASING FO-RULES

```
Input:  a set of FO-rules, $\Sigma_{st}^{FO}$ over $\langle \mathbf{S}, \mathbf{T} \rangle$,
        an instance $I$ of $\mathbf{S}$
Output: an instance $chase_{\Sigma_{st}^{FO}}(I)$

Let $chase_{\Sigma_{st}^{FO}}(I) = \emptyset$
For each $\varphi(\overline{x}) \rightarrow \psi(\overline{x}) \in \Sigma_{st}^{FO}$
    Let $Q_\varphi(I) = \{a(\overline{x}) \mid a\ assignment\ s.t. I \models \varphi(a(\overline{x}))\}$
    For each $a(\overline{x}) \in Q_\varphi(I)$
        $chase_{\Sigma_{st}^{FO}}(I) = chase_{\Sigma_{st}^{FO}}(I) \cup \{\psi(a(\overline{x}))\}$
```

Computing Overlap Tgds Overlap tgds are computed by means of a chase procedure that works on formulas, as detailed in Algorithm 2. The procedure assumes, without loss of generality, that the s-t tgds are in *normal form*. A set of tgds Σ is in *normal form* if for each m_i, $m_j \in \Sigma$, $(\overline{x}_i \cup \overline{y}_i) \cap (\overline{x}_j \cup \overline{y}_j) = \emptyset$, i.e, the tgds use disjoint sets of variables. Given an attribute A_i and an atom $R(\overline{v})$, we use the notation $v_{R(\overline{v}),A_i}$ to denote the variable associated with attribute A_i in atom $R(\overline{v})$.

There are a few observations in order here. First, note that the definition of overlap above does not require that m_i and m_j be distinct tgds. In fact, overlaps may occur also among atoms in the same tgd, like, for example, in: $A(x, y, z) \rightarrow R(x, N_1, z, N_2) \wedge R(y, N_1, N_3, z)$. The algorithm needs to take care of these cases as well. Second, when an overlap is processed, the chase algorithm also generates a number of *pre-conditions* and a number of *consistency conditions*. Pre-conditions will be used shortly to construct the actual overlap tgd. On the contrary, consistency conditions we will used later on to infer a number of further constraints that the source instance must comply with.

Algorithm 2 CHASING FORMULAS WITH EGDS

```
Input:  a collection of atoms, $\mathcal{R}$, over $\mathbf{T}$,
        an overlap $O$ for atoms $R(\overline{v_1})$, $R(\overline{v_2}) \in \mathcal{R}$,
        and functional dependency $\bar{A} \rightarrow \bar{B} \in \Sigma_t$
Output: a new set of atoms $\text{CHASE}_O(\mathcal{R})$,
        a set of preconditions $\text{PRECON}_O$,
        a set of consistency conditions $\text{CONS}_O$

Let $\text{CHASE}_O(\mathcal{R}) = \mathcal{R}$
Let $\text{PRECON}_O = \emptyset$
Let $\text{CONS}_O = \emptyset$
For $i = 0, \ldots, |\bar{A}|$
    Replace all occ. of $x_{R(\overline{v_1}),\bar{A}_i}$ in $\text{CHASE}_O(\mathcal{R})$ by $x_{R(\overline{v_2}),\bar{A}_i}$
    $\text{PRECON}_O = \text{PRECON}_O \cup \{x_{R(\overline{v_1}),\bar{A}_i} = x_{R(\overline{v_2}),\bar{A}_i}\}$
Repeat until fixpoint
    For each functional dependency $\bar{A}^n \rightarrow \bar{B}^n \in \Sigma_t$
        For each pair of atoms $R(\overline{v_h})$, $R(\overline{v_k})$ in $\text{CHASE}_O(\mathcal{R})$
            If $v_{R(\overline{v_h}),\bar{A}_i^n} = v_{R(\overline{v_k}),\bar{A}_i^n}$, for each $i = 0, \ldots |\bar{A}^n|$
                For $j = 0, \ldots |\bar{B}^n|$
                    If $v_{R(\overline{v_h}),\bar{B}_j^n}$ is universal
                        Replace all occurrences of $v_{R(\overline{v_k}),\bar{B}_j^n}$
                        in $\text{CHASE}_O(\mathcal{R})$ by $v_{R(\overline{v_h}),\bar{B}_j^n}$
                    Else
                        Replace all occurrences of $v_{R(\overline{v_h}),\bar{B}_j^n}$
                        in $\text{CHASE}_O(\mathcal{R})$ by $v_{R(\overline{v_k}),\bar{B}_j^n}$
                    If both $v_{R(\overline{v_h}),\bar{B}_j^n}$ and $v_{R(\overline{v_k}),\bar{B}_j^n}$ are univ.
                        $\text{CONS}_O = \text{CONS}_O \cup \{v_{R(\overline{v_h}),\bar{B}_j^n} = v_{R(\overline{v_k}),\bar{B}_j^n}\}$
```

Based on Algorithm 2, Algorithm 3 generates the actual overlap tgds. It takes as input a mapping scenario $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$, and generates a new set tgds, $\Sigma_{\mathcal{M}}^{ovl}$, by recursively chasing overlaps for tgds; to speed-up the process while keeping it sound, overlaps are processed only among tgds that do not have common ancestors. In order to do this, we keep track of the provenance of overlap tgds. We say that an overlap tgd $m_{O_{i,j}}$ is *derived from* m_i, m_j. $m_{O_{h,k}}$ is *transitively derived from* m_i if it is derived from a tgd m_h such that either $m_h = m_i$ or m_h is transitively derived from m_i; in this case, m_i is called an *ancestor* for $m_{O_{h,k}}$.

Algorithm 4 takes care of adding the necessary negations to generate the final set of CQ$\wedge\neg$UCQ rules.

As noted above, chasing formulas to generate overlap tgds

Algorithm 3 OVERLAP TGDS

```
Input: a mapping scenario $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$
Output: a set of overlap tgds $\Sigma^{ovl}_{\mathcal{M}}$
Let $\Sigma^{ovl}_{\mathcal{M}} = \emptyset$
Repeat until fixpoint
  For each overlap $O_{i,j}$ between $m_i$, $m_j$ in $\Sigma_{st} \cup \Sigma^{ovl}_{\mathcal{M}}$
    Let $m_i : \phi_i(\overline{x}_i) \rightarrow \exists \overline{y}_i(\psi_i(\overline{x}_i, \overline{y}_i))$
    Let $m_j : \phi_j(\overline{x}_j) \rightarrow \exists \overline{y}_j(\psi_j(\overline{x}_j, \overline{y}_j))$
    If $m_i$, $m_j$ do not have common ancestors
      $m_{O_{i,j}} : (\phi_i(\overline{x}_i) \cup \phi_j(\overline{x}_j)) \wedge \textsc{precon}_{O_{i,j}} \rightarrow$
          $\textsc{chase}_{O_{i,j}}(\psi_i(\overline{x}_i, \overline{y}_i) \cup \psi_j(\overline{x}_j, \overline{y}_j))$
      $\Sigma^{ovl}_{\mathcal{M}} := \Sigma^{ovl}_{\mathcal{M}} \cup \{m_{O_{i,j}}\}$
```

Algorithm 4 ADDING NEGATED ATOMS

```
Input: a set of tgds $\Sigma = \Sigma_{st} \cup \Sigma^{ovl}_{\mathcal{M}}$
Output: a new set of tgds $\textsc{addneg}(\Sigma)$
Let $\textsc{addneg}(\Sigma) = \emptyset$
For each tgd $m_i : \phi_i(\overline{x}_i) \rightarrow \exists \overline{y}_i(\psi_i(\overline{x}_i, \overline{y}_i))$ in $\Sigma$
  If there is no tgd $m_j$ transitively derived from $m_i$
    $\textsc{addneg}(\Sigma) := \textsc{addneg}(\Sigma) \cup \{m_i\}$
  Else
    Let $m'_i := \phi_i(\overline{x}_i) \rightarrow \exists \overline{y}_i(\psi_i(\overline{x}_i, \overline{y}_i))$
    For each $m_j : \phi_j(\overline{x}_j) \rightarrow \exists \overline{y}_j(\psi_j(\overline{x}_j, \overline{y}_j)) \in \Sigma^{ovl}_{\mathcal{M}}$
    transitively derived from $m_i$
      Add $\neg(\phi_j(\overline{x}_j))$ to the premise of $m'_i$
    $\textsc{addneg}(\Sigma) := \textsc{addneg}(\Sigma) \cup \{m'_i\}$
```

may also generates a number of *consistency conditions*, i.e., additional variable equations. In fact, during the chase, it may be the case that occurrences of a universal variable are replaced by occurrences of another universal variable; of course, this makes sense only in those cases in which the two variables have the same value.

To check these potential failures as early as possible, our algorithm also infers a number of egds that a source instance should comply with in order to have solutions for the given scenario. This is done quite easily: for each overlap tgd $\phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{x}, \overline{y}))$, we consider all consistency conditions of the form $v_i = v_j$ produced during the chase for that tgd, and generate a source egd of the form $\phi(\overline{x}) \rightarrow (v_i = v_j)$.

Most General Determinations We use the notation $m : \phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{z}, \overline{y}))$ to denote a tgd in which $\overline{z}$ is the set of universal variables occurring in the body, with $\overline{z} \subseteq \overline{x}$. The pseudo-code that looks for most-general determinations is detailed in Algorithm 5.

Please note that we might also decide to reconsider the skolemization strategy in such a way that it never fails. The algorithm may fail during the skolemization phase whenever it is not possible to find a most general determination for an existential variable. As an alternative, in order to push further our best-effort approach, we may as well decide to pick one of the candidate determinations for that variable and output the rewriting anyhow. The resulting implementation would still be sound, since we return a solution only when it actually satisfies the given target egds. As an advantage, however, for some scenarios there might be specific input instances for which a solution can be computed anyway.

C. SKETCH OF THE PROOFS

Theorem 3.1 *There is a scenario $\mathcal{M} = (\boldsymbol{S}, \boldsymbol{T}, \Sigma_{st}, \Sigma_t)$*

Algorithm 5 DETERMINATIONS

```
Input:  a tgd $m : \phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{z}, \overline{y}))$,
        a set of functional dependencies $\Sigma_t$
Output: a mapping $\textsc{mgd}_m : \overline{y} \rightarrow \{[d_i, \overline{x}_j]\}$ or $\perp$
/* Step 1: Find all determinations */
For each existential variable $y_i \in \overline{y}$
  Let $\textsc{dets}_m(y_i) = \emptyset$
  For each occurrence $R.B_j : y_i$ in some atom of $m$
    Let $\textsc{fds}(R, B_j)$ the set of f.d. for $R$ s.t. $B_j \in \bar{B}$
    If $\textsc{fds}(R, B_j)$ is empty
      $\textsc{dets}_m(y_i) := \textsc{dets}_m(y_i) \cup [(m, y_i), \textsc{min}(\overline{z})]$
    Else
      For each $d_i : \bar{A}_i \rightarrow \bar{B}_i \in \textsc{fds}(R, B_j)$
        Let $\textsc{vars}(\bar{A})$ be the variables associated with $\bar{A}$
        If $\textsc{vars}(\bar{A})$ are all universal
          $\textsc{dets}_m(y_i) := \textsc{dets}_m(y_i) \cup [d_i, \textsc{min}(\textsc{vars}(\bar{A}))]$
/* Step 2: Find most-general determinations */
For each existential variable $y_i \in \overline{y}$
  If not exists $\textsc{glb}(\textsc{dets}_m(y_i))$
    return $\perp$
  $\textsc{mgd}_m(y_i) := \textsc{glb}(\textsc{dets}_m(y_i))$
```

where Σ_t is a set of functional dependencies over $\boldsymbol{T}$ such that no complete FO*-implementation exists for $\mathcal{M}$.*

PROOF. *(Sketch)* – Consider the following scenario $\mathcal{M}$, with a single tgd and a single egd:

m_1. $A(x, y) \rightarrow \exists N$: $R(x, N) \wedge R(y, N)$
d_1. $R.1 \rightarrow R.2$

The source instance I can be interpreted as the encoding of a directed graph $G = \{(n_1, n_2) | A(n_1, n_2) \in I\}$. The final effect of the egd on a solution J associated with I is that of assigning the same null values to all target tuples that originate from arcs belonging to a connected component in G. To see this, consider the source instance $I = \{A(a, b), A(b, c), A(d, e)\}$.

By chasing the s-t tgd we obtain a pre-solution $J' = \{R(a, N_0), R(b, N_0), R(b, N_1), R(c, N_1), R(d, N_2), R(e, N_2)\}$.

Chasing the egds produces the effect of equating N_0, N_1, that originate from arcs in the same connected component: $J = \{R(a, N_0), R(b, N_0), R(c, N_0), R(d, N_2), R(e, N_2)\}$.

Consider the target query $Q_t = \{(x, y) | \exists z : B(x, z) \wedge B(y, z)\}$ and let Q_s be a source query capturing the certain answers of Q_t, that is, such that for all source instance I we have $Q_s(I) = \bigcap\{Q_t(J) | J \in \mathsf{Sol}(\mathcal{M}, I)\}$.

Assimilating every instance I with an undirected graph, we can check that $Q_s(I)$ consists of the pair of constants (c_1, c_2) such that I contain a path from c_1 to c_2 and therefore Q_s cannot expressed by a first-order query.

We can finally observe that every scenario $\mathcal{M}$ that has a complete FO-implementation Σ^{FO}_{st} enjoys the following property: for every target conjunctive-query Q_t there is a first-order formula Q_s capturing the certain answers of Q_t in $\mathcal{M}$. Q_s can in fact be obtained from Q_t and Σ^{FO}_{st} by applying known query rewriting techniques.

It follows that $\mathcal{M}$ cannot have complete FO-implementations. $\square$

Theorem 6.1 *Σ^{FO}_{st} is a sound implementation of $\mathcal{M}$.*

PROOF. *(Sketch)* – Given an instance I that satisfies Σ_s, the solution $chase_{\Sigma^{FO}_{st}}(I)$ satisfies the original s-t tgds. To see this, consider that for every tgd $\phi(\bar{x}) \rightarrow \exists \bar{y}(\psi(\bar{x}, \bar{y}))$ in Σ_{st}, Σ^{FO}_{st} contains a rule $\phi(\bar{x}) \rightarrow \psi'(\bar{x})$. We can observe that for all tuple of constant $\bar{c}$ we have $\psi'(\bar{c}) \models \exists \bar{y}, \psi(\bar{c}, \bar{y})$, and therefore $(I, chase_{\Sigma_{st}^{FO}}(I)) \models \Sigma_{st}$.

If Σ_{st}^{FO} succeeds on I, then Σ_{st}^{FO} also satisfies Σ_t, and therefore is a solution for $\mathcal{M}$. To show that $chase_{\Sigma_{st}^{FO}}(I)$ is universal, consider that, by construction of the rules in Σ_{st}^{FO}, it only contains sound information. In fact, let F be the set of function symbols occurring in Σ_{st}^{FO}; consider the second-order formula $\Gamma_{\mathcal{M}} = \exists F, \bigwedge_{r \in \Sigma_{st}^{FO}} \forall \bar{x}\ \phi_r(\bar{x}) \rightarrow \psi_r(\bar{x})$. The property of soundness follows from the fact that this formula $\Gamma_{\mathcal{M}}$ is (by construction) logically implied by $\Sigma_s \wedge \Sigma_{st} \wedge \Sigma_t$.

Therefore, $chase_{\Sigma_{st}^{FO}}(I)$ is a universal solution. $\square$

Theorem 6.2 *The following problem is decidable: given a sound FO-implementation Σ_{st}^{FO} of a scenario $\mathcal{M}$ where the body of each FO-rule is in CQ$\wedge\neg$UCQ, is Σ_{st}^{FO} complete ?*

PROOF. *(Sketch)* – Given a source instance I such that $I \models \Sigma_s$ we say that I is *solvable* iff there is a solution for $\mathcal{M}$ and I (in which case there also exists a universal solution for $\mathcal{M}$ and I). Notice that we can easily decide whether a given instance I is solvable (for instance: by using the standard chase procedure).

A sound FO-implementation Σ_{st}^{FO} is complete iff for all solvable I we have $chase_{\Sigma_{st}^{FO}}(I) \models \Sigma_t$. The key idea of the proof is then to show that we have the following *small-model property*:

LEMMA C.1. *If a sound FO-implementation Σ_{st}^{FO} is not complete, then there exists a* small *solvable I_0 of size $||I_0|| \leq 2 \cdot ||\mathcal{M}||$ such that $chase_{\Sigma_{st}^{FO}}(I_0) \not\models \Sigma_t$.*

To prove this, let Σ_{st}^{FO} be a sound FO-implementation of $\mathcal{M}$ and assume that Σ_{st}^{FO} is not complete. Let I be a (possibly large) solvable instance such that $J = chase_{\Sigma_{st}^{FO}}(I) \not\models \Sigma_t$. Let (A_1, A_2) be a pair of atoms in J violating a functional dependency, that is, $\{A_1, A_2\} \not\models \Sigma_t$.

For each atom $A_i \in \{A_1, A_2\}$ we can find a rule $r_i : \phi_i(\bar{x}) \wedge \neg\phi_i'(\bar{x}) \rightarrow \psi_i(\bar{x})$ and a tuple $\bar{a}_i$ such that: $A_i \in \psi_i(\bar{a}_i)$, $I \models \phi_i(\bar{a}_i)$ and $I \not\models \phi'(\bar{a}_i)$.

Since ϕ_1 and ϕ_2 are two conjunctive queries we can easily construct a subinstance $I_0 \subseteq I$ such that $I_0 \models \phi_1(\bar{a}_1) \wedge \phi_2(\bar{a}_2)$ while $||I_0|| \leq ||\phi_1|| + ||\phi_2||$. Since ϕ_1' and ϕ_2' are two union of conjunctive queries, and $I_0 \subseteq I$, we have $I_0 \not\models \phi_1'(\bar{a}_1)$ and $I_0 \not\models \phi_2'(\bar{a}_2)$. Therefore, $\{A_1, A_2\} \subseteq chase_{\Sigma_{st}^{FO}}(I_0)$ and $chase_{\Sigma_{st}^{FO}}(I_0) \not\models \Sigma_t$.

In only remains to observe that solvability is monotonic: since I is solvable and $I_0 \subseteq I$, the instance I_0 is also solvable, and the Lemma is proven.

Based on the small-model property above, we can finally decide whether Σ_{st}^{FO} is complete by *(i)* enumerating all the possible source instances I of size $||I|| \leq 2 \cdot ||\mathcal{M}||$ (there is only an exponential number of them up to isomorphism), *(ii)* selecting the ones that are solvable, and finally *(iii)* testing whether for each remaining solvable instance I it is the case that $chase_{\Sigma_{st}^{FO}}(I) \models \Sigma_t$. $\square$

Before getting to the proof of Theorem 6.3, we need to introduce some preliminary notions. In the following, for the sake of space, we will simplify the notation as follows: a set of FO-rules, Σ_{st}^{FO}, will be denoted simply by $\mathcal{R}$; given a source instance I, the canonical instance $chase_{\Sigma_{st}^{FO}}(I)$, will be denoted simply by $\mathcal{R}(I)$.

Given a scenario $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_s, \Sigma_{st}, \Sigma_t)$ and a complete FO-implementation $\mathcal{R}$ for $\mathcal{M}$, Algorithm 6 takes care of generating a core implementation $\mathcal{R}^*$ for $\mathcal{M}$.

Our algorithm is the composition of two sets of FO-rules. As a first step, we introduce an additional set of relation symbols, $\mathbf{F}$, containing one relation F_{d_i} for each functional dependency $d_i \in \Sigma_t$. Intuitively, based on the complete implementation of $\mathcal{M}$ provided by $\mathcal{R}$, these relations are used to "materialize" the extent of each functional dependency in an instance $\mathcal{R}(I)$. This is done by a first set of rules, $\mathcal{R}_F$.

The second set of rules is a core-rewriting, $\mathcal{R}_C$. In order to compute such core rewritings we shall use as a building block one of the algorithms introduced in [17, 24]. These algorithms, however, assume a linear order on the active domain of the source. For example, the algorithm in [24] assumes that dependencies have premises in $\mathsf{FO}^{<}$ and may contain inequalities $<$ between variables. Since the core-enforcing script will be composed with another set of FO-rules, we need to properly define the behavior of a set of rules $\mathcal{R}$ over a non-ground instance I. To do this, we first extend the partial order on CONST to Skolem terms in the natural way by assuming a fixed order $<_F$ on the function symbols. Then, we assume that the canonical solution $\mathcal{R}(I)$ is obtained by chasing the rules in the usual way.

We obtain our core implementation, $\mathcal{R}^*$, by composing $\mathcal{R}_F$ and R_C. In fact, we can state the following composition lemma: for every pair of FO-rules, $\mathcal{R}_a$ and $\mathcal{R}_b$, we can compute a set of FO-rules $\mathcal{R}_{ab}$ such that, for all ground instances I we have $\mathcal{R}_{ab}(I) = \mathcal{R}_b(\mathcal{R}_a(I))$. Notice, however, that this is not even necessary in practice. In fact, in order to generate the final SQL script, it is sufficient to execute the two scripts in sequence: first the script derived from $\mathcal{R}_F$, and then and the one derived from $\mathcal{R}_C$ on the intermediate result generated by the first.

It is worth noting that, as a preliminary step, Algorithm 6 rewrites the s-t tgds in $\mathcal{M}$ by adding overlap tgds, i.e., it generates a new, logically equivalent scenario $\mathcal{M}' = (\mathbf{S}, \mathbf{T}, \Sigma_s, \Sigma_{st} \cup \Sigma_{st}^{ovl}, \Sigma_t)$. This step has no cost, since it has already been performed to generate the complete implementation $\mathcal{R}$, as discussed in Section 4, with a main difference: to make the resulting dependencies more manageable for core-computation purposes, here we do not add negated atoms to tgd premises. This choice is justified by two observations: (*i*) in this way, the core rewriting is drastically simplified; (*ii*) the needed negations will be added anyway by the core-rewriting algorithm.

Theorem 6.3 *Given a complete implementation $\mathcal{M}$ it is always possible to derive a core implementation for $\mathcal{M}$*

PROOF. *(Sketch)* Consider the set of rules $\mathcal{R}^*$ defined in Algorithm 6. We now show that, given a solvable source instance I for $\mathcal{M}$, $J = \mathcal{R}^*(I)$ is a core solution in $\mathsf{Core}(\mathcal{M}, I)$.

Part 1: $\mathcal{R}^*(I)$ is a universal solution – $\mathcal{R}^*(I) \in \mathsf{USol}(\mathcal{M}, I)$
It is clear that $J = \mathcal{R}^*(I)$ is a pre-solution, i.e., $(I, \mathcal{R}^*(I)) \models \Sigma_{st}$ and we can also check that $\mathcal{R}^*(I)$ is *sound*, meaning that for all $K \in \mathsf{Sol}(I, \mathcal{M})$ there is an homomorphism from $\mathcal{R}^*(I)$ to K. To show that $\mathcal{R}^*(I)$ is a universal solution it only remains to prove that $\mathcal{R}^*(I) \models \Sigma_t$.

We define the set N of *affected nulls* as the smallest set of nulls such that, for every $d : R\langle i_1, \ldots, i_k\rangle \rightarrow j$ in Σ_t and every atom $R(t_1, \ldots, t_n)$ in J, if $\{t_{i_1}, \ldots, t_{i_k}\} \subseteq (N \cup \text{CONST})$ then $t_j \in (N \cup \text{CONST})$. We say that an atom $R(t_1, \ldots, t_n)$ of J is *affected* when there exists some $d : R\langle i_1, \ldots, i_k\rangle \rightarrow j$ in Σ such that $\{t_{i_1}, \ldots, t_{i_k}\} \subseteq (N \cup \text{CONST})$.

Intuitively, the steps 2 and 4 of the algorithm ensure that Σ_t is satisfied by every pair of affected atoms in J while the steps 3 and 5 ensure that Σ_t is satisfied by the remaining atoms.

Consider $\Sigma_{st}'' = \text{ADDNEG}(\Sigma_{st}') = \text{ADDNEG}(\Sigma_{st} \cup \Sigma_{st}^{ovl})$, and the set of FO-rules $\mathcal{R}''$ obtained from Σ_{st}'' by replacing every existential variable in the head of a tgd by its

Algorithm 6 Core Computation

```
Input:  a scenario $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_s, \Sigma_{st}, \Sigma_t)$,
        a complete FO-implementation $\mathcal{R}$ for $\mathcal{M}$
Output: a core implementation $\mathcal{R}^*$ for $\mathcal{M}$
/* Step 1: Generate the additional schema $\boldsymbol{F}$ */
Let $\mathbf{F}$ be the schema $\{F_d \,|\, d : R\langle i_1, \ldots, i_k\rangle \to j \in \Sigma_t\}$,
where $F_d$ is a fresh predicate symbol of arity $k+1$
/* Step 2: Generate $\mathcal{R}_F$ from $\boldsymbol{S}$ to $\boldsymbol{F}$ */
Let $\mathcal{R}_F := \emptyset$
For every $\phi(\bar{x}) \to \psi(\bar{x})$ in $\mathcal{R}$, every $d : R\langle i_1, \ldots, i_k\rangle \to j$
in $\Sigma_t$ and every atom $R(t_1, \ldots, t_n)$ in $\psi$
    $\mathcal{R}_F := \mathcal{R}_F \cup \{\phi(\bar{x}) \to F_d(t_{i_1}, \ldots, t_{i_k}, t_j)\}$.
/* Step 3: Generate $\Sigma'_{st}$ by adding overlap tgds to $\Sigma_{st}$ */
Let $\Sigma'_{st} := \Sigma_{st} \cup \Sigma^{ovl}_{st}$
/* Step 4: Generate $\Sigma^F_{st}$ from $(\boldsymbol{S} \cup \boldsymbol{F})$ to $\boldsymbol{T}$ */
Let $\Sigma^F_{st} := \Sigma'_{st}$
Repeat until fixpoint
    For each $r : \forall \bar{x}\ \phi(\bar{x}) \to \exists y, \bar{z}, \psi(\bar{x}, y, \bar{z})$ in $\Sigma^F_{st}$, each $d$ :
    $R\langle i_1, \ldots, i_k\rangle \to j$ in $\Sigma_t$, and each atom $R(t_1, \ldots, t_n)$ in
    $\psi$ such that $t_j = y$ and $\{t_{i_1}, \ldots, t_{i_k}\} \subseteq \{\bar{x}\}$
        Replace $r$ in $\Sigma^F_{st}$ by the rule
        $r' : \forall \bar{x}, y\ F_d(t_{i_1}, \ldots, t_{i_k}, y) \wedge \phi(\bar{x}) \to \exists \bar{z}\ \psi(\bar{x}, y, \bar{z})$.
/* Step 5: Generate a core rewriting $\mathcal{R}_C$ for $\Sigma^F_{st}$ */
Considering the scenario $\mathcal{M}^F$=$((\mathbf{S} \cup \mathbf{A}), \mathbf{T}, \emptyset, \Sigma^F_{st}, \emptyset)$, use
the algorithm in [24] (or that in [17]) to generate a core
implementation $\mathcal{R}_C$ for $\mathcal{M}^F$.
/* Step 6: Generate a core implem. $\mathcal{R}^*$ for $\mathcal{M}$ */
Generate the set of rules $\mathcal{R}^*$ as the composition of $\mathcal{R}_F$ and
$\mathcal{R}_C$ such that, for every ground instance $I$ of $\mathbf{S}$ we have
$\mathcal{R}^*(I) = \mathcal{R}_C(I \cup \mathcal{R}_F(I))$.
```

standard skolemization. We can observe that the instance $J = \mathcal{R}^*(I)$ is contained (up to isomorphism) in the instance $J'' = \mathcal{R}''(I \cup \mathcal{R}_F(I))$. Consider now $d : R\langle i_1, \ldots, i_k\rangle \to j$ in Σ_t and two atoms $R(t_1, \ldots, t_n)$ and $R(t'_1, \ldots, t'_n)$ in J'' such that $(t_{i_1}, \ldots, t_{i_k}) = (t_{i'_1}, \ldots, t_{i'_k})$. If these atoms are (both) affected in J'', we can check that $\mathcal{R}_F(I)$ contains exactly one atom of the form $F_d(t_{i_1}, \ldots, t_{i_k}, t'')$. Then, the term t_b and t'_b are both equal to t'' and d is satisfied.

Otherwise, none of these two atoms is affected and there is some $l \in \{1, \ldots, k\}$ such that t_l is of the form $f_{m,y}\langle \ldots\rangle$ for some rule m of $\mathcal{R}''$ and some existential variable y in the head of $\mathcal{R}''$. The two atoms have therefore been produced by the same rule of $\mathcal{R}''$ and it follows from the definition of overlaps (and in particular, the use of the chase) that d is satisfied.

A crucial observation here is that, even though we refer to ADDNEG in this proof, it is not needed to (and preferable not to) use ADDNEG in the algorithm because the step of core computation already takes care of preventing the introduction of unnecessary atoms in J that may violate Σ_t. As an example, consider $\Sigma_{st} = \{m : S(x, y) \to \exists U, V, W, R(x, U, V) \wedge R(y, U, W) \wedge T(V, W)\}$ and $\Sigma_t = \{d : R\langle 1, 2\rangle \to 3\}$. The overlap algorithm produces a new tgd $m' : S(x', x') \to \exists U', V', R(x', U', V') \wedge T(V', V')$ and ADDNEG adds the inequality constraint $x \neq y$ to the body of m. After skolemizing the existential variables in a standard way we obtain $\mathcal{R}''$ such that, for all instance I_0 of $\{S\}$, $\mathcal{R}''(I_0) \models d$. Then for every core solution J_0 for $(\{A\}, \{R, T\}, \{m, m'\})$ and I_0, since J_0 is contained in $\mathcal{R}''(I_0)$ up to isomorphism, we have also $J_0 \models d$.

Part 2: $\mathcal{R}^*(I)$ is a core solution – $\mathcal{R}^*(I) \in \mathsf{Core}(\mathcal{M}, I)$

When the instance $I' = I \cup \mathcal{R}_F(I)$ is ground, the step 4 of the algorithm ensures that the universal solution $J = \mathcal{R}^*(I)$ is a core and therefore $J \in \mathsf{Core}(\mathcal{M}, I)$. The difficulty comes from the fact that I' generally contains some nulls (the Skolem terms introduced by $\mathcal{R}_F$). We can show however that the nulls of I' that are used by $\mathcal{R}_C$ can safely be treated as constants with respect to core computation. More precisely, if we let N be the set of affected nulls (as defined in Part 1) we can observe that N coincides precisely with the set of nulls occurring both in I' and J. We can then check that for every homomorphism $h : J \to J$ and every $n_i \in N$ we necessarily have $h(n_1) = n_i$. This property can finally be proven by induction after observing that, for every atom $a = R(t_1, \ldots, t_n)$ in J and every $d : R\langle i_1, \ldots, i_k\rangle \to j$ in Σ_t, if h is the identity on $\{t_{i_1}, \ldots, t_{i_k}\}$ then h is also the identity on $\{t_j\}$ (because $\{a, h(a)\} \subseteq J$ and $J \models d$). $\square$

D. DESCRIPTION OF SCENARIOS

For our experiments we selected 8 scenarios. Of these, 3 (denoted as s_a, s_b, s_c) are variants of scenarios taken from the literature (one from [12], one from [13], one from [3]). The fourth one (s_d) was explicitly constructed in order to test the behavior of the rewriting in case of an exponential number of overlaps. For these experiments, the number of tgds varies between 4 and 10, and the number of egds varies between 5 and 13.

Four additional synthetic scenarios were used to test the scalability of the algorithm with respect to larger number of relations and dependencies. Using the scenario generator developed for STBenchmark [3], we generated four relational scenarios ($s_{25}, s_{50}, s_{75}, s_{100}$) containing 20/50/75/100 tables, with an average join path length of 3, variance 1. To generate complex schemas we used a composition of basic cases with an increasing number between 1 and 15, in particular we used: Vertical Partitioning (3/6/11/15 repetitions), Denormalization (3/6/12/15), and Copy (1 repetition). With such settings we got schemas varying between 11 relations with 3 joins and 52 relations with 29 joins. The number of tgds varies between 22 and 93, and the number of egds between 25 and 100 (corresponding to one key for each relation).

All experiments have been executed on a Intel Core 2 Duo machine with 2.4Ghz processor and 2 GB of RAM under Linux. The DBMS was PostgreSQL 8.3.

By looking at Figures 3.a and 3.b it is possible to see that times for scenario s_c were strongly in favor of the rewriting. The reason for this is related to the Skolem minimization procedure. Scenario s_c includes existential variables for which the standard Skolem terms depends on 10 different universal variables; this means that, for source instances on which such variables may assume rather long values, the corresponding Skolem string may become rather large. It turns out that generating many large Skolem strings is often a bottleneck in the execution of the SQL script. This is probably due to the fact that appending strings is not always a very optimized operation in a DBMS. As a consequence, the generation of the pre-solution is rather slow. The rewriting does not incur this cost, since the skolemization algorithm minimizes Skolem terms, so that they depend on a single variable (representing the key value for the corresponding tuple); this generates much shorter strings that are manipulated more efficiently by the engine. It remains an open problem to find alternative encodings for Skolem terms that alleviate these problems.