Core Schema Mappings

Giansalvatore Mecca[1] Paolo Papotti[2] Salvatore Raunich[1]

[1] Dipartimento di Matematica e Informatica – Università della Basilicata – Potenza, Italy
[2] Dipartimento di Informatica e Automazione – Università Roma Tre – Roma, Italy

ABSTRACT

Research has investigated mappings among data sources under two perspectives. On one side, there are studies of practical tools for schema mapping generation; these focus on algorithms to generate mappings based on visual specifications provided by users. On the other side, we have theoretical researches about data exchange. These study how to generate a solution – i.e., a target instance – given a set of mappings usually specified as tuple generating dependencies. However, despite the fact that the notion of a core of a data exchange solution has been formally identified as an optimal solution, there are yet no mapping systems that support core computations. In this paper we introduce several new algorithms that contribute to bridge the gap between the practice of mapping generation and the theory of data exchange. We show how, given a mapping scenario, it is possible to generate an executable script that computes core solutions for the corresponding data exchange problem. The algorithms have been implemented and tested using common runtime engines to show that they guarantee very good performances, orders of magnitudes better than those of known algorithms that compute the core as a post-processing step.

Categories and Subject Descriptors

H.2 [**Database Management**]: Heterogeneous Databases

General Terms

Algorithms, Design

Keywords

Schema Mappings, Data Exchange, Core Computation

1. INTRODUCTION

Integrating data coming from disparate sources is a crucial task in many applications. An essential requirement of any data integration task is that of manipulating *mappings* between sources. Mappings are executable transformations – say, SQL or XQuery scripts – that specify how an instance of the source repository should be translated into an instance of the target repository. There are several ways to express such mappings. A popular one consists in using *tuple generating dependencies* (*tgds*) [3]. We may identify two broad research lines in the literature.

On one side, we have studies on practical tools and algorithms for *schema mapping generation*. In this case, the focus is on the development of systems that take as input an abstract specification of the mapping, usually made of a bunch of correspondences between the two schemas, and generate the mappings and the executable scripts needed to perform the translation. This research topic was largely inspired by the seminal papers about the Clio system [17, 18]. The original algorithm has been subsequently extended in several ways [12, 4, 2, 19, 7] and various tools have been proposed to support users in the mapping generation process. More recently, a benchmark has been developed [1] to compare research mapping systems and commercial ones.

On the other side, we have theoretical studies about *data exchange*. Several years after the development of the initial Clio algorithm, researchers have realized that a more solid theoretical foundation was needed in order to consolidate practical results obtained on schema mapping systems. This consideration has motivated a rich body of research in which the notion of a *data exchange problem* [9] was formalized, and a number of theoretical results were established. In this context, a *data exchange setting* is a collection of mappings – usually specified as tgds – that are given as part of the input; therefore, the focus is not on the generation of the mappings, but rather on the characterization of their properties. This has brought to an elegant formalization of the notion of a solution for a data exchange problem, and of operators that manipulate mappings in order, for example, to compose or invert them.

However, these two research lines have progressed in a rather independent way. To give a clear example of this, consider the fact that there are many possible solutions for a data exchange problem. A natural question is the following: "which solution should be materialized by a mapping system?" A key contribution of data exchange research was the formalization of the notion of *core* [11] of a data exchange solution, which was identified as an "optimal" solution. Informally speaking, the core has a number of nice properties: it is "irredundant", since it is the smallest among the solutions that preserve the semantics of the exchange, and represents a "good" instance for answering queries over

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

the target database. It can therefore be considered a natural requirement for a schema mapping system to generate executable scripts that materialize core solutions.

Unfortunately, there is yet no schema mapping generation algorithm that natively produces executable scripts that compute the core. On the contrary, the solution produced by known schema mapping systems – called a *canonical solution* – typically contains quite a lot of redundancy. This is partly due to the fact that computing cores is a challenging task. Several polynomial-time algorithms [11, 13, 20] have been developed to compute the core of a data exchange solution. These algorithms represent a relevant step forward, but still suffer from a number of serious drawbacks from a schema-mapping perspective. First, they are intended as post-processing steps to be applied to the canonical solution, and require a custom engine to be executed; as such, they are not integrated into the mapping system, and are hardly expressible as an executable (SQL) script. Second and more important, as it will be shown in our experiments, they do not scale to large exchange tasks: even for databases of a few thousand tuples computing the core typically requires many hours.

In this paper we introduce the +Spicy[1] mapping system. The system is based on a number of novel algorithms that contribute to bridge the gap between the practice of mapping generation and the theory of data exchange. In particular:

(*i*) +Spicy integrates the computation of core solutions in the mapping generation process in a highly efficient way; after a set of tgds has been generated based on the input provided by the user, cores are computed by a natural rewriting of the tgds in terms of algebraic operators; this allows for an efficient implementation of the rewritten mappings using common runtime languages like SQL or XQuery and guarantees very good performances, orders of magnitude better than those of previous core-computation algorithms; we show in the paper that our strategy scales up to large databases in practical scenarios;

(*ii*) we classify data exchange settings in several categories, based on the structure of the mappings and on the complexity of computing the core; correspondingly, we identify several approximations of the core of increasing quality; the rewriting algorithm is designed in a modular way, so that, in those cases in which computing the core requires heavy computations, it is possible to fine tune the trade off between quality and computing times;

(*iii*) finally, the rewriting algorithm can be applied both to mappings generated by the mapping system, or to pre-existing tgds that are provided as part of the input. Moreover, all of the algorithms introduced in the paper can be applied both to relational and to nested – i.e., XML – scenarios; +Spicy is the first mapping system that brings together a sophisticate and expressive mapping generation algorithm with an efficient strategy to compute irredundant solutions.

In light of these contributions, we believe this paper makes a significant advancement towards the goal of integrating data exchange concepts and core computations into existing database technology.

The paper is organized as follows. In the following section, we give an overview of the main ideas. Section 3 provides some background. Section 4 provides a quick overview of the tgd generation algorithm. The rewriting algorithms are in Sections 5, 6. A discussion on complexity is in Section 7. Experimental results are in Section 8. A discussion of related work is in Section 9.

[1]Pronounced "more spicy".

2. OVERVIEW

In this section we shall introduce the various algorithms that are developed in the paper.

It is well known that translating data from a given source database may bring to a certain amount of redundancy into the target database. To see this, consider the mapping scenario in Figure 1. A source instance is shown in Figure 2. A constraint-driven mapping system as Clio would generate for this scenario several mappings, like the ones below.[2] Mappings are tgds that state how tuples should be produced in the target based on tuples in the source. Mappings can be expressed using different syntax flavors. In schema mapping research [12], an XQuery-like syntax is typically used. Data exchange papers use a more classical logic-based syntax that we also adopt in this paper.

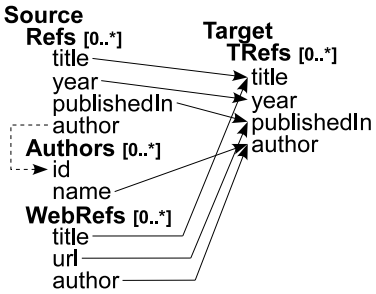

Figure 1: Mapping Bibliographic References

m_1. $\forall t, y, p, i$: $Refs(t, y, p, i) \rightarrow \exists N$: $TRefs(t, y, p, N)$
m_2. $\forall i, n$: $Auths(i, n) \rightarrow \exists T, Y, P$: $TRefs(T, Y, P, n)$
m_3. $\forall t, y, p, i, n$: $Refs(t, y, p, i) \wedge Auths(i, n) \rightarrow TRefs(t, y, p, n)$
m_4. $\forall t, p, n$: $WebRefs(t, p, n) \rightarrow \exists Y$: $TRefs(t, Y, p, n)$

Mapping m_3 above states that for every tuple in *Refs* that has a join with a tuple in *Authors*, a tuple in *TRefs* must be produced. Mapping m_1 is needed to copy into the target

Refs

title	year	publishedIn	author
A Relational Model...	1970	CACM	a1
DB Teaching Materials	1985	http://www....	a2
The SQL92 Standard	1992	ANSI Stds	NULL

WebRefs

title	url	author
DB Teaching Materials	http://www....	C. Date

Auths

id	name
a1	E. F. Codd
a2	C. Date

Target: TRefs

title	year	publishedIn	author
A Relational Model...	1970	CACM	E. F. Codd
DB Teaching Materials	1985	http://www...	C. Date
The SQL92 Standard	1992	ANSI Stds	NULL
DB Teaching Materials	NULL	http://www...	C. Date
A Relational Model...	1970	CACM	NULL
DB Teaching Materials	1985	http://www...	NULL
NULL	NULL	NULL	E. F. Codd
NULL	NULL	NULL	C. Date

Figure 2: Instances for the References Scenario

[2]Note that the generation of mapping m_1 requires an extension of the algorithms described in [18, 12].

references that do not have authors, like "*The SQL92 Standard*". Similarly, mapping m_2 is needed in order to copy names of authors for which there are no references (none in our example). Finally, mapping m_4 copies tuples in *WebRefs*.

Given a source instance, executing the tgds amounts to running the standard *chase* algorithm on the source instance to obtain an instance of the target called a *canonical universal solution* [9]; note that a natural way to chase the dependencies is to execute them as SQL statements in the DBMS.

These expressions materialize the target instance in Figure 2. While this instance satisfies the tgds, still it contains many redundant tuples, those with a gray background. As shown in [12], for large source instances the amount of redundancy in the target may be very large, thus impairing the efficiency of the exchange and the query answering process. This has motivated several practical proposals [8, 12, 7] towards the goal of removing such redundant data. Unfortunately, these proposals are applicable only in some cases and do not represent a general solution to the problem.

Data exchange research [11] has introduced the notion of *core* solutions as "optimal" solutions for a data exchange problem. Consider for example tuples t_1 = *(null, null, null, E.F.Codd)* and t_2 = *(A Relational Model..., 1970, CACM, E.F.Codd)* in Figure 2. The fact that t_1 is redundant with respect to t_2 can be formalized by saying that there is an *homomorphism* from t_1 to t_2. A *homomorphism*, in this context, is a mapping of values that transforms the nulls of t_1 into the constants of t_2, and therefore t_1 itself into t_2. This means that the solution in Figure 2 has an *endomorphism*, i.e., a homomorphism into a sub-instance – the one obtained by removing t_1. The *core* [11] is the smallest among the solutions for a given source instance that has homomorphisms into all other solutions. The core of the solution in Figure 2 is in fact the portion of the *TRefs* table with a white background.

A possible approach to the generation of the core for a relational data exchange problem is to generate a canonical solution by chasing the tgds, and then to apply a post-processing algorithm for core identification. Several polynomial algorithms have been identified to this end [11, 13]. These algorithms provide a very general solution to the problem of computing core solutions for a data exchange setting. Also, an implementation of the core-computation algorithm in [13] has been developed [20], thus making a significant step towards the goal of integrating core computations in schema mapping systems.

However, experience with these algorithms shows that, although polynomial, they require very high computing times since they look for all possible endomorphisms among tuples in the canonical solution. As a consequence, they hardly scale to large mapping scenarios. Our goal is to introduce a core computation algorithm that lends itself to a more efficient implementation as an executable script and that scales well to large databases. To this end, in the following sections we introduce two key ideas: the notion of *homomorphism among formulas* and the use of *negation* to rewrite tgds.

Subsumption and Rewriting. The first intuition is that it is possible to analyze the set of formulas in order to recognize when two tgds may generate redundant tuples in the target. This happens when it is possible to find a homomorphism between the right-hand sides of the two tgds. Consider tgds m_2 and m_3 above; with an abuse of notation, we consider the two formulas as sets of tuples, with existentially quantified variables that correspond to nulls; it can be seen that the conclusion $TRefs(T, Y, P, n)$ of m_2 can be mapped into the conclusion $TRefs(t, y, p, n)$ of m_3 by the following mapping of variables: $T \to t$, $Y \to y$, $P \to p$; in this case, we say that m_3 *subsumes* m_2; similarly, m_3 also subsumes m_1 and m_4. This gives us a nice necessary condition to intercept possible redundancy (i.e., possible endomorphisms among tuples in the canonical solution). Note that the condition is merely a necessary one, since the actual generation of endomorphisms among facts depends on values coming from the source. Note also that we are checking for the presence of homomorphisms among formulas, i.e., conclusions of tgds, and not among instance tuples; since the number of tgds is typically much smaller than the size of an instance, this task can be carried out quickly.

A second important intuition is that, whenever we identify two tgds m, m' such that m subsumes m', we may prevent the generation of redundant tuples in the target instance by executing them according to the following strategy: (i) generate target tuples for m, the "more informative" mapping; (ii) for m', generate only those tuples that actually add some new content to the target. To make these ideas more explicit, we may rewrite the original tgds as follows (universally quantified variables have been omitted since they should be clear from the context):

$$m'_3.\ Refs(t,y,p,i) \wedge Auths(i,n) \to TRefs(t,y,p,n)$$
$$m'_1.\ Refs(t,y,p,i) \wedge \neg(Refs(t,y,p,i) \wedge Auths(i,n)) \to \exists N\colon TRefs(t,y,p,N)$$
$$m'_2.\ Auths(i,n) \wedge \neg(Refs(t,y,p,i) \wedge Auths(i,n)) \wedge \neg(WebRefs(t,p,n)) \to \exists X,Y,Z\colon TRefs(X,Y,Z,n)$$
$$m'_4.\ WebRefs(t,p,n) \wedge \neg(Refs(t,y,p,i) \wedge Auths(i,n)) \to \exists Y\colon TRefs(t,Y,p,n)$$

Once we have rewritten the original tgds in this form, we can easily generate an executable transformation under the form of relational algebra expressions. Here, negations become difference operators; in this simple case, nulls can be generated by outer-union operators, $\cup_*$, that have the semantics of the `insert into` SQL statement:[3]

$$m'_3 : TRefs = \pi_{t,y,p,n}(Refs \bowtie Auths)$$
$$m'_1 : \quad \cup_*(\pi_{t,y,p}(Refs) - \pi_{t,y,p}(Refs \bowtie Auths))$$
$$m'_2 : \quad \cup_*(\pi_n(Auths) - \pi_n(Refs \bowtie Auths) - \pi_a(WebRefs))$$
$$m'_4 : \quad \cup_*(\pi_{t,p,n}(WebRefs) - \pi_{t,p,n}(Refs \bowtie Auths))$$

The algebraic expressions above can be easily implemented in an executable script, say in SQL or XQuery, to be run in any database engine. As a consequence, there is a noticeable gain in efficiency with respect to the algorithms for core computation proposed in [11, 13, 20].

Despite the fact that this example looks pretty simple, it captures a quite common scenario. However, removing redundancy from the target may be a much more involved process, as discussed in the following.

Coverages. Consider now the mapping scenario in Figure 3. The target has two tables, in which genes reference their protein via a foreign key. In the source we have data coming

[3]We omit the actual SQL code since it tends to be quite long. Note also that in the more general case Skolem functions are needed to properly generate nulls.

from two different biology databases. Data in the PDB tables comes from the *Protein Database*, which is organized in a way that is similar to the target. On the contrary, the EMBL table contains data from the popular *EMBL* repository; there, tuples need to be partitioned into a gene and a protein tuple. In this process, we need to "invent" a value to be used as a key-foreign key pair for the target. This is usually done using a *Skolem function* [18]. This transformation

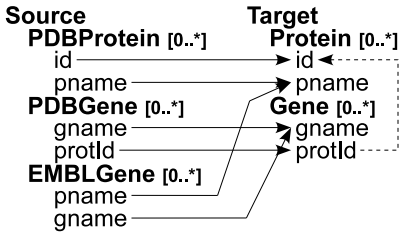

Figure 3: Genes

can be expressed using the following tgds:

m_1. $PDBProtein(i, p) \rightarrow Protein(i, p)$
m_2. $PDBGene(g, i) \rightarrow Gene(g, i)$
m_3. $EMBLGene(p, g) \rightarrow \exists N$: $Gene(g, N) \wedge Protein(N, p)$

Sample instances are in Figure 4. It can be seen that the canonical solution contains a smaller endomorphic image – the core – since the tuples (*14-A, N2*) and (*N2, 14-A-antigen*), where *N2* was invented during the chase, can be mapped to the tuples (*14-A, p1*) and (*p1, 14-A-antigen*). In fact, if we look at the right-hand sides of tgds, we see that there is a homomorphism from the right-hand side of m_3, $\{Gene(g, N), Protein(N, p)\}$, into the right-hand sides of m_1 and m_2, $\{Gene(g, i), Protein(i, p)\}$: it suffices to map N into i. However, this homomorphism is a more complex one with respect to those in the previous example. There, we were mapping the conclusion of one tgd into the conclusion of another. We call this form of homomorphism a *coverage* of m_3 by m_1 and m_2. We may rewrite the original tgds as follows

PDBProtein

id	pname
p1	14-A-antigen
p2	ACC synthase

PDBGene

gname	protId
14-A	p1
16-ACC	p2

EMBLGene

gname	pname
14-A	14-A-antigen
15-B	alpha-precursor

Protein (Target)

id	pname
p1	14-A-antigen
p2	ACC synthase
N1	alpha precursor
N2	14-A-antigen

Gene (Target)

gname	proteinId
14-A	p1
16-ACC	p2
15-B	*N1*
14-A	*N2*

Figure 4: Instances for the genes example

to obtain the core:

m'_1. $PDBProtein(i, p) \rightarrow Protein(i, p)$
m'_2. $PDBGene(g, i) \rightarrow Gene(g, i)$
m'_3. $EMBLGene(p, g) \wedge \neg(PDBGene(g, i) \wedge PDBProtein(i, p))$
$\rightarrow \exists N\ \ Gene(g, N) \wedge Protein(N, p)$

From the algebraic viewpoint, mapping m'_3 above requires to generate in *Gene* and *Protein* tuples based on the following expression:

$$EMBLGene - \pi_{p,g}(PDBGene \bowtie PDBProtein)$$

In the process, we also need to generate the appropriate Skolem functions to correlate tuples in *Gene* with the corresponding tuples in *Protein*. A key difference with respect to subsumptions is that there can be a much larger number of possible rewritings for a tgd like m_3, and therefore a larger number of additional joins and differences to compute. This is due to the fact that, in order to discover coverages, we need to look for homomorphisms of every single atom into other atoms appearing in right-hand sides of the tgds, and then combine them in all possible ways to obtain the rewritings. To give an example, suppose the source also contains tables *XProtein*, *XGene* that write tuples to *Protein* and *Gene*; then, we might have to rewrite m_3 by adding the negation of four different joins: (*i*) *PDBProtein* and *PDBGene*; (*ii*) *XProtein*, *XGene*; (*iii*) *PDBProtein* and *XGene*; (*iv*) *XProtein* and *PDBGene*. This obviously increases the time needed to execute the exchange.

We emphasize that this form of complex subsumption could be reduced to a simple subsumption if the source database contained a foreign-key constraint from *PDBGene* to *PDBProtein*; in this case, only two tgds would be necessary. In our experiments, simple subsumptions were much more frequent than complex coverages. Moreover, even in those cases in which coverage rewritings were necessary, the database engine performed very well.

Handling Self-Joins. Special care must be devoted to tgds containing *self-joins* in the conclusion, i.e., tgds in which the same relation symbols occurs more than once in the right-hand side. One example of this kind is the "self-join" scenario in STMark [1], or the "RS" scenario in [11]; in this section we shall refer to a simplified version of the latter, in which the source schema contains a single relation R, the target schema a single relation S, and a single tgd is given:

$$m_1.\ R(a, b) \rightarrow \exists x_1, x_2 : S(a, b, x_1) \wedge S(b, x_2, x_1)$$

Assume table R contains a single tuple: $R(1, 1)$; by chasing m_1, we generate two tuples in the target: $S(1, 1, N1)$, $S(1, N2, N1)$. It is easy to see that this set has a proper endomorphism, and therefore its core corresponds to the single tuple $S(1, 1, N1)$.

Even though the example is quite simple, eliminating this kind of redundancy in more complex scenarios can be rather tricky, and therefore requires a more subtle treatment. Intuitively, the techniques discussed above are of little help, since, regardless of how we rewrite the premise of the tgd, on a tuple $R(1, 1)$ the chase will either generate two tuples or none of them. As a consequence, we introduce a more sophisticate treatment of these cases.

Let us first note that in order to handle tgds like the one above, the mapping generation system had to be extended with several new primitives with respect to those offered by [18, 12], which cannot express scenarios with self-joins. We extend the primitives offered by the mapping system as follows: (*i*) we introduce the possibility of duplicating sets in the source and in the target; to handle the tgd above, we duplicate the S table in the target to obtain two different copies, S^1, S^2; (*ii*) we give users full control over joins in the sources, in addition to those corresponding to foreign key constraints; using this feature, users can specify arbitrary join paths, like the join on the third attribute of S^1 and S^2.

Based on this, we notice that the core computation can be carried-on in a clean way by adopting a two-step process. As a first step, we rewrite the original tgd using duplications as follows:

$$m_1.\ R(a, b) \rightarrow \exists x_1, x_2 : S^1(a, b, x_1) \wedge S^2(b, x_2, x_1)$$

By doing this, we "isolate" the tuples in S^1 from those in S^2. Then, we construct a second exchange to copy tuples of S^1 and S^2 into S, respectively. However, we can more easily rewrite the tgds in the second exchange in order to remove redundant tuples. In our example, on the source tuple $R(1,1)$ the first exchange generates tuples $S^1(1,1,N1)$ and $S^2(1,N2,N1)$; the second exchange discards the second tuple and generates the core. The process is sketched in Figure 5. These ideas are made more precise in the following sections.

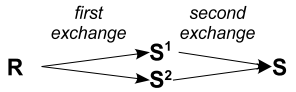

Figure 5: The Double Exchange

3. PRELIMINARIES

In the following sections we will mainly make reference to relational settings, since most of the results in the literature refer to the relational model. However, our algorithms extend to the nested case, as it will be discussed in Section 8.

Data Model We fix two disjoint sets: a set of *constants*, CONST, a set of *labeled nulls*, VAR. 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 $\mathbf{S}$. We allow to express *key constraints* and *foreign key constraints* over a schema, defined as usual. In the following, we will 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.

Given an instance I, we shall denote by *const(I)* the set of constants occurring in I, and by *var(I)* the set of labeled nulls in I. *dom(I)*, its *active domain*, will be *const(I)*$\cup$*var(I)*.

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 $\langle \mathbf{S}, \mathbf{T} \rangle$.

Dependencies Given two schemas, $\mathbf{S}$ and $\mathbf{T}$, an *embedded dependency* [3] is a first-order formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{x}, \overline{y}))$, where $\overline{x}$ and $\overline{y}$ are vectors of variables, $\phi(\overline{x})$ is a conjunction of atomic formulas such that all variables in $\overline{x}$ appear in it, and $\psi(\overline{x}, \overline{y})$ is a conjunction of atomic formulas. $\phi(\overline{x})$ and $\psi(\overline{x}, \overline{y})$ may contain equations of the form $v_i = v_j$, where v_i and v_j are variables.

An embedded dependency is a *tuple generating dependency* if $\phi(\overline{x})$ and $\psi(\overline{x}, \overline{y})$ only contain relational atoms. It is an *equality generating dependency (egd)* if $\psi(\overline{x}, \overline{y})$ contains only equations. A tgd is called a *source-to-target tgd* if $\phi(\overline{x})$ is a formula over $\mathbf{S}$ and $\psi(\overline{x}, \overline{y})$ over $\mathbf{T}$. It is a *target tgd* if both $\phi(\overline{x})$ and $\psi(\overline{x}, \overline{y})$ are formulas over $\mathbf{T}$.

Homomorphisms and Chase 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.

Dependencies are executed using the classical chase procedure. Given an instance $\langle I, J \rangle$, during the chase a tgd $\phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{x}, \overline{y}))$ is fired by a *value assignment* a, that is, an homomorphism from $\phi(\overline{x})$ into I, such that there is no extension of a that maps $\phi(\overline{x}) \cup \psi(\overline{x}, \overline{y})$ into $\langle I, J \rangle$. To fire the tgd a is extended to $\psi(\overline{x}, \overline{y})$ by assigning to each variable in $\overline{y}$ a fresh null, and then adding the new facts to J.

Data Exchange Setting A *data exchange setting* is a quadruple $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$, where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema, Σ_{st} is a set of source-to-target tgds, and Σ_t is a set of target dependencies that may contain tgds and egds. Associated with such a setting is the following *data exchange problem*: given an instance I of the source schema $\mathbf{S}$, find a finite target instance J such that I and J satisfy Σ_{st} and J satisfies Σ_t. In the case in which the set of target dependencies Σ_t is empty, we will use the notation $(\mathbf{S}, \mathbf{T}, \Sigma_{st})$.

Given a data exchange setting $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ and a source instance I, a *universal solution* [9] for I is a solution J such that, for every other solution J' there is a homomorphism $h : J \rightarrow J'$. The *core* [11] of a universal solution J, $\mathbf{C}$, is a subinstance of J such that there is a homomorphism from J to $\mathbf{C}$, but there is no homomorphism from J to a proper subinstance of $\mathbf{C}$.

4. TGD GENERATION

Before getting into the details of the tgd rewriting algorithm, let us give a quick overview of how the input tgds are generated by the system. Note that, as an alternative, the user may decide to load a set of pre-defined tgds provided as logical formulas encoded in a fixed textual format.

The tgd generation algorithm we describe here is a generalization of the *basic mapping* generation algorithm introduced in [18]. The input to the algorithm is a *mapping scenario*, i.e., an abstract specification of the mapping between source and target. In order to achieve a greater expressive power, we enrich the primitives for specifying scenarios. More specifically, given a source schema $\mathbf{S}$ and a target $\mathbf{T}$, a *mapping scenario* is specified as follows:
(i) two (possibly empty) sets of *duplications* of the sets in $\mathbf{S}$ and in $\mathbf{T}$; each duplication of a set R corresponds to adding to the data source a new set named R^i, for some i, that is an exact *copy* of R;
(ii) two (possibly empty) sets of *join constraints* over $\mathbf{S}$ and over $\mathbf{T}$; each join constraint specifies that the system needs to chase a join between two sets; foreign key constraints also generate join constraints;
(iii) a set of *value correspondences*, or *lines*; for the sake of simplicity in this paper we concentrate on 1 : 1 correspondences of the form $A_S \rightarrow A_T$.[4]

[4]In its general form, a correspondence maps n source attributes into a target attribute via a *transformation function*; moreover, it can have an attached filter that states under which conditions

The tgd generation algorithm is made of several steps. As a first step, duplications are processed; for each duplication of a set R in the source (target, respectively), a new set R^i is added to the source (target, respectively). Then, the algorithm finds all sets in the source and in the target schema; this corresponds, in the terminology of [18], to finding *primary paths*.

The next step is concerned with generating views over the source and the target. Views are a generalization of *logical relations* in [18] and are the building blocks for tgds. Each view is an algebraic expression over sets in the data source. Let us now restrict our attention to the source (views in the target are generated in a similar way).

The set of views, $\mathcal{V}_{init}$, is initialized as follows: for each set R a view R is generated. This initial set of views is then processed in order to chase join constraints and assemble complex views; intuitively, chasing a join constraint from set R to set R' means to build a view that corresponds to the join of R and R'. As such, each join constraint can be seen as an operator that takes a set of existing views and transforms them into a new set, possibly adding new views or changing the input ones. Join constraints can be *mandatory* or *non mandatory*; intuitively, a mandatory join constraint states that two sets must either appear together in a view, or not appear at all.

Once views have been generated for the source and the target schema, it is possible to produce a number of candidate tgds. We say that a source view v *covers* a value correspondence $A_S \rightarrow A_T$ if A_S is an attribute of a set that appears in v; similarly for target views. We generate a candidate tgd for each pair made of a source view and a target view that covers at least one correspondence. The source view generates the left-hand side of the tgd, the target view the right-hand side; lines are used to generate universally quantified variables in the tgd; for each attribute in the target view that is not covered by a line, we add an existentially quantified variable.

5. TGD REWRITING

We are now ready to introduce the rewriting algorithm. We concentrate on data exchange settings expressed as a set of source-to-target tgds, i.e., we do not consider target tgds and egds. Target constraints are used to express key and foreign key constraints on the target. With respect to target tgds, we assume that the source-to-target tgds have been rewritten in order to incorporate any target tgds corresponding to foreign key constraints. In [10] it is proven that it is always possible to rewrite a data exchange setting with a set of *weakly acyclic* [9] target tgds into a setting with no target tgds such that the cores of the two settings coincide, provided that the target tgds satisfy a *boundedness* property. With respect to key constraints, they can be enforced in the final SQL script after the core for the source-to-target tgds has been generated.[5]

the correspondence must be applied; our system handles the most general form of correspondences; it also handles *constant* lines. It is possible to extend the algorithms presented in this paper to handle the most general form of correspondence; this would be important in order to incorporate *conditional tgds* [6]; while the extension is rather straightforward for constants appearing in tgd premises, it is more elaborate for constants in tgd conclusions, and is therefore left to future work.

[5]The description of the algorithm is out of the scope of this paper.

A key contribution of this paper is the definition of a rewriting algorithm that takes as input a set of source-to-target tgds Σ and rewrites them into a new set of constraints Σ' with the nice property that, given a source instance I, the canonical solution for Σ' on I coincides with the core of Σ on I.

We make the assumption that the set Σ is *source-based*. A tgd $\phi(\overline{x}) \rightarrow \exists\overline{y}(\psi(\overline{x},\overline{y}))$ is *source-based* if: (i) the left-hand side $\phi(\overline{x})$ is not empty; (ii) the vector of universally quantified variables $\overline{x}$ is not empty; (iii) at least one of the variables in $\overline{x}$ appears in the right hand side $\psi(\overline{x},\overline{y})$.

This definition, while restricting the variety of tgds handled by the algorithm, captures the notion of a "useful" tgd in a schema mapping scenario. In fact, note that tgds in which the left-hand side is empty or it contains no universally quantified variables – like, for example $\rightarrow \exists X, Y : T(X, Y)$, or $\forall a : S(a) \rightarrow \exists X, Y : R(X, Y) \wedge S(Y, X)$ – would generate target tuples made exclusively of nulls, which are hardly useful in practical cases.

Besides requiring that tgds are source-based, without loss of generality we also require that the input tgds are in *in normal form*, i.e., each tgd uses distinct variables, and no tgd can be decomposed in two different tgds having the same left-hand side. To formalize this second notion, let us introduce the *Gaifman graph* of a formula as the undirected graph in which each variable in the formula is a node, and there is an edge between v_1 and v_2 if v_1 and v_2 occur in the same atom. The *dual Gaifman graph* of a formula is an undirected graph in which nodes are atoms, and there is an edge between atoms $R_i(\overline{x}_i,\overline{y}_i)$ and $R_j(\overline{x}_j,\overline{y}_j)$ if there is some existential variable y_k occurring in both atoms.

Definition: A set of tgds Σ is in *normal form* if: (i) 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; (ii) for each tgd, the dual Gaifman graph of atoms is connected.

If the input set of tgds is not in normal form, it is always possible to preliminarily rewrite them to obtain an input in normal form.[6]

5.1 Formula Homomorphisms

An important intuition behind the algorithm is that by looking at homomorphisms between tgd conclusions, we may identify when firing one tgd may lead to the generation of "redundant" tuples in the target. To formalize this idea, we introduce the notion of *formula homomorphism*, which is reminiscent of the notion of *containment mapping* used in [16]. We find it useful to define homomorphisms among variable occurrences, and not among variables.

Definition: Given an atom $R(A_1 : v_1, \ldots, A_k : v_k)$ in a formula $\psi(\overline{x},\overline{y})$, a *variable occurrence* is a pair $R.A_i : v_i$. We denote by $occ(\psi(\overline{x},\overline{y}))$ the set of variable occurrences in $\psi(\overline{x},\overline{y})$. A variable occurrence $R.A_i : v_i \in occ(\psi(\overline{x},\overline{y}))$ is a *universal occurrence* if v_i is a universally quantified variable; it is a *Skolem occurrence* if v_i is an existentially quantified variable that occurs more than once in $\psi(\overline{x},\overline{y})$; it is a *pure null occurrence* if v_i is an existentially quantified variable that occurs only once in $\psi(\overline{x},\overline{y})$.

Intuitively, the term "pure null" is used to denote those variables that generate labeled nulls that can be safely re-

[6]In case the dual Gaifman graph of a tgd is not connected, we generate a set of tgds with the same premise, one for each connected component in the dual Gaifman graph.

placed with ordinary null values in the final instance. There is a precise hierarchy in terms of information content associated with each variable occurrence. More specifically, we say that a variable occurrence o_2 is *more informative* than variable occurrence o_1 if one of the following holds: (i) o_2 is universal, and o_1 is not; (ii) o_2 is a Skolem occurrence and o_1 is a pure null.

Definition: Given two formulas, $\psi_1(\overline{x}_1, \overline{y}_1)$, $\psi_2(\overline{x}_2, \overline{y}_2)$, a *variable substitution*, h, is an injective mapping from the set $occ(\psi_1(\overline{x}_1, \overline{y}_1))$ to $occ(\psi_2(\overline{x}_2, \overline{y}_2))$ that maps universal occurrences into universal occurrences. In the following we shall refer to the variable occurrence $h(R.A_i : x_i)$ by the syntax $A_i : h_{R.A_i}(x_i)$.

Definition: Given two sets of atoms $\mathcal{R}_1$, $\mathcal{R}_2$, a *formula homomorphism* is a variable substitution h such that, for each atom $R(A_1 : v_1, \ldots, A_k : v_k) \in \mathcal{R}_1$, it is the case that: (i) $R(A_1 : h_{R.A_1}(v_1), \ldots, A_k : h_{R.A_k}(v_k)) \in \mathcal{R}_2$; (ii) for each pair of existential occurrences $R_i.A_j : v$, $R'_i.A'_j : v$ in $\mathcal{R}_1$ it is the case that either $h_{R_i.A_j}(v)$ and $h_{R'_i.A'_j}(v)$ are both universal or $h_{R_i.A_j}(v) = h_{R'_i.A'_j}(v)$.

Given a set of tgds $\Sigma_{ST} = \{\phi_i(\overline{x}_i) \rightarrow \exists \overline{y}_i(\psi_i(\overline{x}_i, \overline{y}_i)), i = 1, \ldots, n\}$, a *simple formula endomorphism* is a formula homomorphism from $\psi_i(\overline{x}_i, \overline{y}_i)$ to $\psi_j(\overline{x}_j, \overline{y}_j)$, for some $i, j \in \{1, \ldots, n\}$. A *formula endomorphism* is a formula homomorphism from $\bigcup_{i=1}^{n} \psi_i(\overline{x}_i, \overline{y}_i)$ to $\bigcup_{i=1}^{n} \psi_i(\overline{x}_i, \overline{y}_i) - \{\psi_j(\overline{x}_j, \overline{y}_j)\}$ for some $j \in \{1, \ldots, n\}$.

Definition: A formula homomorphism is said to be *proper* if either the size of $\mathcal{R}_2$ is greater than the size of $\mathcal{R}_1$ or there exists at least one occurrence $R.A_i : v_i$ in $\mathcal{R}_1$ such that $h_{R.A_i}(v_i)$ is more informative than $R.A_i : v_i$.

To give an example, consider the following tgds. Suppose relation W has three attributes, A, B, C:

$$m_1.\ A(x_1) \rightarrow \exists Y_0, Y_1 : W(x_1, Y_0, Y_1)$$
$$m_2.\ B(x_2, x_3) \rightarrow \exists Y_2 : W(x_2, x_3, Y_2)$$
$$m_3.\ C(x_4) \rightarrow \exists Y_3, Y_4 : W(x_4, Y_3, Y_4),\ V(Y_4)$$

There are two different formula homomorphisms: (i) the first maps the right-hand side of m_1 into the rhs of m_2: $W.A : x_1 \rightarrow W.A : x_2, W.B : Y_0 \rightarrow W.B : x_3, W.C : Y_1 \rightarrow W.C : Y_2$; (ii) the second maps the rhs of m_1 into the rhs of m_3: $W.A : x_1 \rightarrow W.A : x_4, W.B : Y_0 \rightarrow W.B : Y_3, W.C : Y_1 \rightarrow W.C : Y_4$. Both homomorphisms are proper.

Note that every standard homomorphism h on the variables of a formula induces a formula homomorphism h that associates with each occurrence of a variable v the same value $h(v)$. The study of formula endomorphisms provides nice necessary conditions for the presence of endomorphisms in the solutions of an exchange problem.

THEOREM 5.1 (NECESSARY CONDITION). *Given a data exchange setting $(\boldsymbol{S}, \boldsymbol{T}, \Sigma_{ST})$, suppose Σ_{ST} is a set of source-based tgds in normal form. Given an instance I of $\boldsymbol{S}$, call J a universal solution for I. If J contains a proper endomorphism, then $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$ contains a proper formula endomorphism.*

Typically, the canonical solution contains a proper endomorphism into its core. It is useful, for application purposes, to classify data exchange scenarios in various categories, based on the complexity of core identification. To do this, as discussed in Section 2, special care needs to be devoted to those tgds m in which the same relation symbol appears more than once in the conclusion. In this case we say that m *contains self-joins* in tgd conclusions.

(i) a *subsumption scenario* is a data exchange scenario in which Σ_{ST} may only contain simple endomorphisms, and no tgd contains self-joins in tgd conclusions.

(ii) a *coverage scenario* is a scenario in which Σ_{ST} may contain arbitrary endomorphisms, but no tgd contains self-joins in tgd conclusions.

(iii) a *general scenario* is a scenario in which Σ_{ST} may contain tgds with arbitrary self-joins.

In the following sections, we introduce the rewriting for each of these categories.

5.2 Subsumption Scenarios

Definition: Given two tgds m_1, m_2, whenever there is a simple homomorphism h from $\psi_1(\overline{x}_1, \overline{y}_1)$ to $\psi_2(\overline{x}_2, \overline{y}_2)$, we say that m_2 *subsumes* m_1, in symbols $m_1 \preceq m_2$. If h is proper, we say that m_2 *properly subsumes* m_1, in symbols $m_1 \prec m_2$.

Subsumptions are very frequent and can be handled efficiently. One example is the references scenario in Section 2. There, as discussed, the only endomorphisms in the right-hand sides of tgds are simple endomorphisms that map an entire tgd conclusion into another conclusion. Then, it may be the case that the two tgds are instantiated with value assignments a, a' and produce two sets of facts $\psi(\overline{a}, \overline{b})$ and $\psi_\prime(\overline{a_\prime}, \overline{b}_\prime)$ such there is an endomorphism that maps $\psi(\overline{a}, \overline{b})$ into $\psi_\prime(\overline{a_\prime}, \overline{b}_\prime)$. In these cases, whenever m_2 subsumes m_1, we rewrite m_1 by adding to the its left-hand side the negation of the left-hand side of m_2; this prevents the generation of redundant tuples.

Note that a set of tgds may contain both proper and non-proper subsumptions. However, only proper ones introduce actual redundancy in the final instance; non-proper subsumptions generate tuples that are identical up to the renaming of nulls and therefore are filtered-out by the semantics of the chase. As a consequence, for performance purposes it is convenient to concentrate on proper subsumptions.

We can now introduce the rewriting of the original set of source-to-target tgds Σ into a new set of tgds, Σ', as follows.

Definition: For each $m = \phi(\overline{x}) \rightarrow \exists \overline{y}(\psi(\overline{x}, \overline{y}))$ in Σ, add to Σ' a new tgd $m_{subs} = \phi_\prime(\overline{x}_\prime) \rightarrow \exists \overline{y}_\prime(\psi_\prime(\overline{x}_\prime, \overline{y}_\prime))$, obtained by rewriting m as follows:
(i) initialize $m_{subs} = m$;
(ii) for each tgd $m_s = \phi_s(\overline{x}_s) \rightarrow \exists \overline{y}_s(\psi_s(\overline{x}_s, \overline{y}_s))$ in Σ such that $m \prec m_s$, call h the homomorphism of m into m_s; add to $\phi_\prime(\overline{x}_\prime)$ a negated sub-formula $\wedge\neg(\gamma_s)$, where γ_s is obtained as follows:
$(ii.a)$ initialize $\gamma_s = \phi_s(\overline{x}_s)$;
$(ii.b)$ for each pair of existential occurrences $R_i.A_j : v$, $R'_i.A'_j : v$ in $\psi(\overline{x}, \overline{y})$ such that $h_{R_i.A_j}(v)$ and $h_{R'_i.A'_j}(v)$ are both universal, add to γ_s an equation of the form $h_{R_i.A_j}(v) = h_{R'_i.A'_j}(v)$;
$(ii.c)$ for each universal position $A_i : x_i$ in $\psi(\overline{x}, \overline{y})$, add to γ_s an equation of the form $x_i = h_{R.A_i}(x_i)$. Intuitively, the latter equations correspond to computing differences among instances of the two formulas.

Consider again the W example in the previous paragraph. The tgds in normal form are reported below. Based on the

proper subsumptions, we can rewrite mapping m_1 as follows:

$$m_1'.\ A(x_1) \wedge \neg(B(x_2, x_3) \wedge x_1 = x_2) \\ \wedge\neg(C(x_4) \wedge x_1 = x_4) \rightarrow \exists Y_0, Y_1\ \ W(x_1, Y_0, Y_1)$$

By looking at the logical expressions for the rewritten tgds it can be seen how we have introduced negation. Results that have been proven for data exchange with positive tgds extend to tgds with *safe negation* [14]. To make negation safe, we assume that during the chase universally quantified variables range over the active domain of the source database. This is reasonable since – as it was discussed in Section 2 – the rewritten tgds will be translated into a relational algebra expression.

5.3 Coverage Scenarios

Consider now the case in which the tgds contain endomorphisms that are not simple subsumptions; recall that we are still assuming the tgds contain no self-joins in their conclusions. Consider the genes example in Section 2. Tgd m_3 in that example states that the target must contain two tuples, one in the *Gene* table and one in the *Protein* table that join on the *protein* attribute. However, this constraint do not necessarily must be satisfied by inventing a new value. In fact, there might be tuples generated by m_1 and m_2 that satisfy the constraint imposed by m_3. Informally speaking, a *coverage* for the conclusion of a tgd is a set of atoms from other tgds that might represent alternative ways of satisfying the same constraint.

Definition: Assume that, for tgd $m = \phi(\overline{x}) \rightarrow \exists\overline{y}(\psi(\overline{x}, \overline{y}))$, there is an endomorphism $h : \bigcup_i \psi_i(\overline{x}_i, \overline{y}_i) \rightarrow \bigcup_i \psi_i(\overline{x}_i, \overline{y}_i) - \{\psi(\overline{x}, \overline{y})\}$. Call $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$ a minimal set of formulas such that h maps each atom $R_i(\ldots)$ in $\psi(\overline{x}, \overline{y})$ into some atom $R_i(\ldots)$ of $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$ a *coverage* of m; note that if i equals 1 the coverage becomes a subsumption.

The rewriting algorithm for coverages is made slightly more complicated by the fact that proper join conditions must in general be added among coverage premises.

Definition: For each $m = \phi(\overline{x}) \rightarrow \exists\overline{y}(\psi(\overline{x}, \overline{y}))$ in Σ, add to Σ' a new tgd $m_{cov} = \phi_\prime(\overline{x}_\prime) \rightarrow \exists\overline{y}_\prime(\psi_\prime(\overline{x}_\prime, \overline{y}_\prime))$, obtained as follows:
(i) initialize $m_{cov} = m_{subs}$, as defined above;
(ii) for each coverage $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$ of m, call h the homomorphism of $\psi(\overline{x}, \overline{y})$ into $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$; add to $\phi_\prime(\overline{x}_\prime)$ a negated sub-formula $\wedge\neg(\gamma_c)$, where γ_c is obtained as follows:
(iia) initialize $\gamma_c = \bigwedge_i \phi_i(\overline{x}_i)$;
(iib) for each universal position $A_i : x_i$ in $\psi(\overline{x}, \overline{y})$, add to γ_c an equation of the form $x_i = h_{R.A_i}(x_i)$
(iic) for each existentially quantified variable y in $\psi(\overline{x}, \overline{y})$, and any pair of positions $A_i : y$, $A_j : y$ such that $h_{R.A_i}(y)$ and $h_{R.A_j}(y)$ are universal variables, add to γ_c an equation of the form $h_{R.A_i}(y) = h_{R.A_j}(y)$.

To see how the rewriting works, consider the following example (existentially quantified variables are omitted since they should be clear from the context):

$$m_1.\ A(a_1, b_1, c_1) \rightarrow R(a_1, N_{10}) \wedge S(N_{10}, N_{11}) \wedge T(N_{11}, b_1, c_1)$$
$$m_2.\ B(a_2, b_2) \rightarrow R(a_2, b_2)$$
$$m_3.\ F^1(a_3, b_3) \wedge F^2(b_3, c_3) \rightarrow S(a_3, c_3)$$
$$m_4.\ D(a_4, b_4) \rightarrow T(a_4, b_4, N_4)$$
$$m_5.\ E(a_5, b_5) \rightarrow R(a_5, N_{50}) \wedge S(N_{50}, N_{51}) \wedge T(N_{51}, b_5, N_{52})$$

Consider tgd m_5. It is subsumed by m_1. It is also covered by $\{R(a_2, b_2),\ S(a_3, c_3),\ T(a_4, b_4, N_4)\}$, by homomorphism: $\{R.1 : a_5 \rightarrow R.1 : a_2,\ R.2 : N_{50} \rightarrow R.2 : b_2,\ S.1 : N_{50} \rightarrow S.1 : a_3,\ S.2 : N_{51} \rightarrow S.2 : c_3,\ T.1 : N_{51} \rightarrow T.1 : a_4,\ T.2 : b_5 \rightarrow T.2 : b_4\ T.3 : N_{52} \rightarrow T.3 : N_4\}$. Based on this, we rewrite tgd m_5 as follows:

$$m_5'.\ E(a_5, b_5) \wedge \neg(A(a_1, b_1, c_1) \wedge a_5 = a_1 \wedge b_5 = b_1) \\ \wedge\neg(B(a_2, b_2) \wedge F^1(a_3, b_3) \wedge F^2(b_3, c_3) \wedge D(a_4, b_4) \\ \wedge\ b_2 = a_3 \wedge c_3 = a_4 \wedge a_5 = a_2 \wedge b_5 = b_4) \\ \rightarrow R(a_5, N_{50}) \wedge S(N_{50}, N_{51}) \wedge T(N_{51}, b_5, N_{52})$$

It is possible to prove the following result:

THEOREM 5.2 (CORE COMPUTATION). *Given a data exchange setting $(\boldsymbol{S}, \boldsymbol{T}, \Sigma_{ST})$, suppose Σ_{ST} is a set of source-based tgds in normal form that do not contain self-joins in tgd conclusions. Call Σ'_{ST} the set of coverage rewritings of Σ_{ST}. Given an instance I of $\boldsymbol{S}$, call J, J' the canonical solutions of Σ_{ST} and Σ'_{ST} for I. Then J' is the core of J.*

The proof is based on the fact that, whenever two tgds m_1, m_2 in Σ_{ST} are fired to generate an endomorphism, several homomorphisms must be in place. Call $\overline{a_1}, \overline{a_2}$ the variable assignments used to fire m_1, m_2; suppose there is an homomorphism h from $\psi_1(\overline{a_1}, \overline{b}_1)$ to $\psi_2(\overline{a_2}, \overline{b}_2)$. Then, by Theorem 5.1, we know that there must be a formula homomorphism h' from $\psi_1(\overline{x}_1, \overline{y}_1)$ to $\psi_2(\overline{x}_2, \overline{y}_2)$, and therefore a rewriting of m_1 in which the premise of m_2 is negated. By composing the various homomorphism it is possible to show that the rewriting of m_1 will not be fired on assignment a_1. Therefore, the endomorphism will not be present in J'.

6. REWRITING TGDS WITH SELF-JOINS

The most general scenario is the one in which one relation symbol may appear more than once in the right-hand side of a tgd. This introduces a significant difference in the way redundant tuples may be generated in the target, and therefore increases the complexity of core identification.

There are two reasons for which the rewriting algorithm introduced above does not generate the core. Note that the algorithm removes redundant tuples by preventing a tgd to be fired for some value assignment. Therefore, it prevents redundancy that comes from instantiations of different tgds, but it does not control redundant tuples generated within an instantiation of a single tgd. In fact, if a tgd writes two or more tuples at a time into a relation R, solutions may still contain unnecessary tuples. As a consequence, we need to rework the algorithm in a way that, for a given instantiation of a tgd, we can intercept every single tuple added to the target by firing the tgd, and remove the unnecessary ones. In light of this, our solution to this problem is to adopt a two-step process, i.e., to perform a double exchange.

6.1 The Double Exchange

Given a set of source-to-target tgds, Σ_{ST} over $\mathbf{S}$ and $\mathbf{T}$, as a first step we normalize the input tgds; we also introduce suitable duplications of the target sets in order to remove self-joins. A *duplicate* of a set R is an exact copy named R^i of R. By doing this, we introduce a new, intermediate schema, $\mathbf{T'}$, obtained from $\mathbf{T}$. Then, we produce a new set of tgds $\Sigma_{ST'}$ over $\mathbf{S}$ and $\mathbf{T'}$ that do not contain self-joins.

Definition: Given a mapping scenario $(\mathbf{S}, \mathbf{T}, \Sigma_{ST})$ where Σ_{ST} contains self-joins in tgd conclusions, the *intermediate scenario* $(\mathbf{S}, \mathbf{T'}, \Sigma_{ST'})$ is obtained as follows: for each tgd m in Σ_{ST} add a tgd m' to $\Sigma_{ST'}$ such that m' has the same

premise as m and for each target atom $R(\overline{x}, \overline{y})$ in m, m' contains a target atom $R^i(\overline{x}, \overline{y})$, where R^i is a fresh duplicate of R.

To give an example, consider the *RS* example in [11]. The original tgds are reported below:

$$m_1.\ R(a,b,c,d) \rightarrow \exists x_1,x_2,x_3,x_4,x_5 : S(x_5,b,x_1,x_2,a) \wedge S(x_5,c,x_3,x_4,a) \wedge S(d,c,x_3,x_4,b)$$
$$m_2.\ R(a,b,c,d) \rightarrow \exists x_1,x_2,x_3,x_4,x_5 : S(d,a,a,x_1,b) \wedge S(x_5,a,a,x_1,a) \wedge S(x_5,c,x_2,x_3,x_4)$$

In that case, $\Sigma_{ST'}$ will be as follows (variables have been renamed to normalize the tgds):

$$m'_1.\ R(a,b,c,d) \rightarrow \exists x_1,x_2,x_3,x_4,x_5 : S^1(x_5,b,x_1,x_2,a) \wedge S^2(x_5,c,x_3,x_4,a) \wedge S^3(d,c,x_3,x_4,b)$$
$$m'_2.\ R(e,f,g,h) \rightarrow \exists y_1,y_2,y_3,y_4,y_5 : S^4(h,e,e,y_1,f) \wedge S^5(y_5,e,e,y_1,e) \wedge S^6(y_5,g,y_2,y_3,y_4)$$

We execute this ST' exchange by applying the rewritings discussed in the previous sections. This yields an instance of **T'** that needs to be further processed in order to generate the final target instance. To do this, we need to execute a second exchange from **T'** to **T**. This second exchange is constructed in such a way to generate the core. The overall process is shown in Figure 6. Note that, while we describe

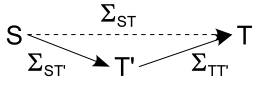

Figure 6: Double Exchange

our algorithm as a double exchange, in our SQL scripts we do not actually implement two exchanges, but only one exchange with a number of additional intermediate views to simplify the rewriting.

Remark The problem of core generation via executable scripts has been independently addressed in [21]. There the authors show that it is possible to handle tgds with self-joins using one exchange only.

6.2 Expansions

Although inspired by the same intuitions, the algorithm used to generate the second exchange is considerably more complex than the ones discussed before. The common intuition is that each of the original source-to-target tgds represents a constraint that must be satisfied by the final instance. However, due to the presence of duplicate symbols, there are in general many different ways of satisfying these constraints. To give an example, consider mapping m'_1 above: it states that the target must contain a number of tuples in S that satisfy the two joins in the tgd conclusion. It is important to note, however, that: (*i*) it is not necessarily true that these tuples must belong to the extent of S^1, S^2, S^3 – since these are pure artifacts introduced for the purpose of our algorithm – but they may also come from S^4 or S^5 or S^6; (*ii*) moreover, these tuples are not necessarily distinct, since there may be tuples that perform a self-join.

In light of these ideas, as a first step of our rewriting algorithm, we compute all *expansions* of the conclusions of the ST' tgds. Each expansion represents one of the possible ways to satisfy the constraint stated by a tgd. For each tgd $m_i \in \Sigma_{ST'}$, we call $\psi_i(\overline{x}_i, \overline{y}_i)$ a *base view*. Consider again tgd m'_1 above; the constraint stated by its base view may obviously be satisfied by copying to the target one atom in S^1, one in S^2 and one in S^3. This corresponds to the *base expansion* of the view, i.e., the expansion that corresponds with the base view itself:

$$e_{11}.S^1(x_5,b,x_1,x_2,a) \wedge S^2(x_5,c,x_3,x_4,a) \wedge S^3(d,c,x_3,x_4,b)$$

However, there are also other ways to satisfy the constraint. One way is to use only one tuple from S^2 and one from S^3, the first one in join with itself on the first attribute – i.e., S^2 is used to "cover" the S^1 atom; this may work as long as it does not conflict with the constants generated in the target by the base view; in our example, the values generated by the S^2 atom must be consistent with those that would be generated by the S^1 atom we are eliminating. We write this second expansion as follows:

$$e_{12}.\ S^2(x_5,c,x_3,x_4,a) \wedge S^3(d,c,x_3,x_4,b) \wedge (S^1(x_5,b,x_1,x_2,a) \wedge b = c)$$

It is possible to see that – from the algebraic viewpoint – the formula requires to compute a join between S^2 and S^3, and then an *intersection* with the content of S^1. This is even more apparent if we look at another possible extension, the one that replaces the three atoms with a single covering atom from S^4 in join with itself:

$$e_{13}.\ S^4(h,e,e,y_1,f) \wedge S^4(h',e',e',y_1,f') \wedge h = h' \wedge (S^1(x_5,b,x_1,x_2,a) \wedge S^2(x_5,c,x_3,x_4,a) \wedge S^3(d,c,x_3,x_4,b) \wedge e = b \wedge f = a \wedge e' = c \wedge f' = a \wedge h' = d \wedge e' = c \wedge f' = b)$$

In algebraic terms, expansion e_{13} corresponds to computing the join $S^4 \bowtie S^4$ and then taking the intersection on the appropriate attributes with the base view, i.e., $S^1 \bowtie S^2 \bowtie S^3$.

A similar approach can be used for tgd m'_2 above. In this case, besides the base expansion, it is possible to see that also the following expansion is derived – S^4 covers S^5 and S^3 covers S^6, the join is on the universal variables d and h:

$$e_{21}.\ S^4(h,e,e,y_1,f) \wedge S^3(d,c,x_3,x_4,b) \wedge h = d \wedge (S^5(y_5,e,e,y_1,e) \wedge S^6(y_5,g,y_2,y_3,y_4) \wedge f = e \wedge g = c)$$

As a first step of the rewriting, for each ST' tgd, we take the conclusion, and compute all possible expansions, including the base expansion. The algorithm to generate expansions is very similar to the one to compute coverages described in the first section, with several important differences. In particular, we need to extend the notion of homomorphism in such a way that atoms corresponding to duplicates of the same set can be matched.

Definition: We say that two sets R and R' are *equal up to duplications* if they are equal, or one is a duplicate of the other, or both are duplicates of the same set. Given two sets of atoms $\mathcal{R}_1$, $\mathcal{R}_2$, an *extended formula homomorphism*, h, is defined as a formula homomorphism, with the variant that h is required to map each atom $R(A_1 : v_1, \ldots, A_k : v_k) \in \mathcal{R}_1$ into an atom $R'(A_1 : h_{R.A_1}(v_1), \ldots, A_k : h_{R.A_k}(v_k)) \in \mathcal{R}_2$ such that R and R' are not necessarily the same symbol but are equal up to duplications.

Note that, in terms of complexity, another important difference is that in order to generate expansions we do not need to exclusively use atoms in other tgds, but may reuse atoms from the tgd itself. Also, the same atom may be used multiple times in an expansion. Call $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$ the union

of all atoms in the conclusions of $\Sigma_{ST'}$. To compute its expansions, if the base view has size k, we consider all *multisets* of size k or less of atoms in $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$. If one atom occurs more than once in a multiset, we assume that variables are properly renamed to distinguish the various occurrences.

Definition: Given a base view $\psi(\overline{x}, \overline{y})$ of size k, a multiset $\mathcal{R}$ of atoms in $\bigcup_i \psi_i(\overline{x}_i, \overline{y}_i)$ of size k or less, and an extended formula homomorphism h from $\psi(\overline{x}, \overline{y})$ to $\mathcal{R}$, an *expansion* $e_{\mathcal{R},h}$ is a logical formula of the form $c \wedge i$, where:
(i) c – the *coverage formula* – is constructed as follows:
(ia) initialize $c = \mathcal{R}$;
(ib) for each existentially quantified variable y in $\psi(\overline{x}, \overline{y})$, and any pair of positions $A_i : y$, $A_j : y$ such that $h_{R.A_i}(y)$ and $h_{R.A_j}(y)$ are universal variables, add to c an equation of the form $h_{R.A_i}(y) = h_{R.A_j}(y)$.
(ii) i – the *intersection formula* – is constructed as follows:
(iia) initialize $i = \psi(\overline{x}, \overline{y})$;
(iib) for each universal position $A_i : x_i$ in $\psi(\overline{x}, \overline{y})$, add to i an equation of the form $x_i = h_{R.A_i}(x_i)$.

Note that for base expansions the intersection part can be removed. It can be seen that the number of coverages may significantly increase when the number of self-joins increase.[7] In the RS example our algorithm finds 10 expansions of the two base views, 6 for the conclusion of tgd m_1' and 4 for the conclusion of tgd m_2'.

6.3 T'T Tgds

Expansions represent all possible ways in which the original constraints may be satisfied. Our idea is to use expansions as premises for the T'T tgds that actually write to the target. The intuition is pretty simple: for each expansion e we generate a tgd. The tgd premise is the expansion itself, e. The tgd conclusion is the formula e_T, obtained from e by replacing all duplicate symbols by the original one. To give an example, consider expansion e_{12} above. It generates a tgd like the following:

$$S^2(x_5, c, x_3, x_4, a) \wedge S^3(d, c, x_3, x_4, b) \\ \wedge (S^1(x_5, b, x_1, x_2, a) \wedge b = c) \rightarrow \exists N_3, N_4, N_5 : \\ \rightarrow S(N_5, c, N_3, N_4, a) \wedge S(d, c, N_3, N_4, b)$$

Before actually executing these tgds, two preliminary steps are needed. As a first step, we need to normalize the tgds, since conclusions are not necessarily normalized. Second, as we already did in the first exchange, we need to suitably rewrite the tgds in order to prevent the generation of redundant tuples.

6.4 T'T Rewriting

To generate the core, we now need to identify which expansions may generate redundancy in the target. In essence, we look for subsumptions among expansions, in two possible ways.

First, among all expansions of the same base view, we try to favor the *'most compact'* ones, i.e., those that generate less tuples in the target. To see an example, consider the source tuple $R(n, n, n, k)$; chasing the tuple using the base expansion e_{11} generates in the target three tuples: $S(N_5, n, N_1, N_2, n), S(N_5, n, N_3, N_4, n), S(k, n, N_3, N_4, n)$; if, however, we chase expansion e_{12}, we generate in the target only two tuples: $S(N_5, n, N_3, N_4, n), S(k, n, N_3, N_4, n)$; chasing e_{13} generates one single tuple that subsumes all of the tuples above: $S(k, n, n, N_1, n)$. We can easily identify this fact by finding an homomorphism from e_{11} to e_{12} and e_{13}, and an homomorphism from e_{12} into e_{13}. We rewrite expansions accordingly by adding negations as in the first exchange.

Definition: Given expansions $e = c \wedge i$ and $e' = c' \wedge i'$ of the same base view, we say that e' is *more compact* than e if there is a formula homomorphism h from the set of atoms $\mathcal{R}_c$ in c to the set of atoms $\mathcal{R}_{c'}$ in c' and either the size of $\mathcal{R}_{c'}$ is smaller than the size of $\mathcal{R}_c$ or there exists at least one occurrence $R.A_i : v_i$ in $\mathcal{R}_c$ such that $h_{R.A_i}(v_i)$ is more informative than $R.A_i : v_i$.

This definition is a generalization of the definition of a subsumption among tgds. Given expansion e, we generate a first rewriting of e, called e_{rew}, by adding to e the negation $\neg(e')$ of each expansion e' of the same base view that is more compact than e, with the appropriate equalities, as for any other subsumption. This means, for example, that expansion e_{12} above is rewritten into a new formula e_{12}^{rew} as follows:

$$e_{12}^{rew}.\ S^2(x_5, c, x_3, x_4, a) \wedge S^3(d, c, x_3, x_4, b) \\ \wedge (S^1(x_5, b, x_1, x_2, a) \wedge b = c) \\ \wedge \neg(S^4(h, e, e, y_1, f) \wedge h = h' \wedge S^4(h', e', e', y_1', f') \wedge \\ (S^1(x_5', b', x_1', x_2', a') \wedge S^2(x_5', c', x_3', x_4', a') \\ \wedge\ S^3(d', c', x_3', x_4', b') \\ \wedge\ e = b' \wedge f = a' \wedge e' = c' \wedge f' = a' \wedge h' = d' \wedge f' = b') \\ \wedge\ c = e \wedge a = f \wedge d = h' \wedge c = e' \wedge b = f')$$

After we have rewritten the original expansion in order to remove unnecessary tuples, we look among other expansions to favor those that generate *'more informative'* tuples in the target. To see an example, consider expansion e_{12} above: it is easy to see that – once we have removed tuples for which there are more compact expansions – we have to ensure that expansion e_{21} of the other tgd does not generate more informative tuples in the target.

Definition: Given expansions $e = c \wedge i$ and $e' = c' \wedge i'$, we say that e' is *more informative* than e if there is a proper homomorphism from the set of atoms $\mathcal{R}_c$ in c to the set of atoms $\mathcal{R}_{c'}$ in c'.

To summarize, to generate the final rewriting, we consider the premise, e, of each T'T tgd; then: (i) we first rewrite e into a new formula e^{rew} by adding the negation of all expansions e_i of the same base view such that e_i is more compact than e; (ii) we further rewrite e^{rew} into a new formula $e^{rew'}$ by adding the negation of e_j^{rew}, for all expansions e_j such that e_j is more informative than e. In the RS example our algorithm finds 21 subsumptions due to more compact expansions of the same base view, and 16 further subsumptions due to more informative expansions.

As a final step, we have to look for proper subsumptions among the normalized tgds to avoid that useless tuples are copied more than once to the target. For example, tuple $S(N_1, h, k, l, m)$ – where N_1 is not in join with other tuples, and therefore is a "pure" null – is redundant in presence of a tuple $S(N_2, h, k, l, m)$ or in the presence of $S(i, h, k, l, m)$. This yields our set of rewritten T'T tgds.

Also in this case it is possible to prove that chasing these rewritten tgds generates core solutions for the original ST tgds.

[7] Note that, as an optimization step, many expansions can be pruned out by reasoning on existential variables.

6.5 Skolem Functions

Our final goal is to implement the computation of cores via an executable script, for example in SQL. In this respect, great care is needed in order to properly invent labeled nulls. A common technique to do this is to use Skolem functions. A Skolem function is usually an uninterpreted term of the form $f_{sk}(v_1, v_2, \ldots, v_k)$, where each v_i is either a constant or a term itself.

An appropriate choice of Skolem functions is crucial in order to correctly reproduce in the final script the semantics of the chase. Recall that, given a tgd $\phi(\overline{x}) \rightarrow \exists\overline{y}(\psi(\overline{x}, \overline{y}))$ and a *value assignment* a, that is, an homomorphism from $\phi(\overline{x})$ into I, before firing the tgd the chase procedure checks that there is no extension of a that maps $\phi(\overline{x}) \cup \psi(\overline{x}, \overline{y})$ into the current solution. In essence, the chase prevents the generation of different instantiations of a tgd conclusion that are identical up to the renaming of nulls.

We treat Skolem functions as interpreted functions that encode their arguments as strings. We call a string generated by a Skolem function a *Skolem string*. Whenever a tgd is fired, existential variables in tgd conclusion are associated with a Skolem string; the Skolem string is then used to generate a unique (integer) value for the variable.

We may see the block of facts obtained by firing a tgd as a hypergraph in which facts are nodes and null values are labeled edges that connect the facts. Each null value that corresponds to an edge of this hypergraph requires an appropriate Skolem function. To correctly reproduce the desired semantics, the Skolem functions for a tgd m should be built is such a way that, if the same tgd or another tgd is fired and generates a block of facts that is identical to that generated by m up to nulls, the Skolem strings are identical. To implement this behavior in our scripts, we embed in the function a full description of the tgd instantiation, i.e., of the corresponding hypergraph. Consider for example the following tgd:

$$R(a, b, c) \rightarrow \exists N_0, N_1 : S(a, N_0),\ T(b, N_0, N_1),\ W(N_1)$$

The Skolem functions for N_0 and N_1 will have three arguments: (a) the sequence of facts generated by firing the tgd (existential variables omitted), i.e., an encoding of the graph nodes; (ii) the sequence of joins imposed by existential variables, i.e., an encoding of the graph edges; (iii) a reference to the specific variable for which the function is used. The actual functions would be as follows:

f_{sk}({S(A:a),T(A:b),W()},{S.B=T.B, T.C=W.A}, S.B=T.B)
f_{sk}({S(A:a),T(A:b),W()},{S.B=T.B, T.C=W.A}, T.C=W.A)

An important point here is that set elements must be encoded in lexicographic order, so that the functions generate appropriate values regardless of the order in which atoms appear in the tgd. This last requirement introduces further subtleties in the way exchanges with self-joins are handled. In fact, note that in tgds like the one above – in which all relation symbols in the conclusion are distinct – the order of set elements can be established at script generation time (they depend on relation names). If, on the contrary, the same atom may appear more than once in the conclusion, then functions of this form are allowed: f_{sk}({S(A:a),S(A:b)},{S.B=S.B}). It can be seen how facts must be reordered at execution time, based on the actual assignment of values to variables.

7. COMPLEXITY AND APPROXIMATIONS

A few comments are worth making here on the complexity of core computations. In fact, the three categories of scenarios discussed in the previous sections have considerably different complexity bounds. Recall that our goal is to execute the rewritten tgds under the form of SQL scrips; in the scripts, negated atoms give rise to difference operators. Generally speaking, differences are executed very efficiently by the DBMS under the form of sort-scans. However, the number of differences needed to filter out redundant tuples depends on the nature of the scenario.

As a first remark, let us note that subsumptions are nothing but particular forms of coverages; nevertheless, they deserve special attention since they are handled more efficiently than coverages. In a subsumption scenario the number of differences corresponds to the number of subsumptions. Consider the graph of the subsumption relation obtained by removing transitive edges. In the worst case – the graph is a path – there are $O(n^2)$ subsumptions. However, this is rather unlikely in real scenarios. Typically, the graph is broken into several smaller connected components, and the number of differences is linear in the number of tgds.

The worst-case complexity of the rewriting is higher for coverage scenarios, for two reasons. First, coverages always require to perform additional joins before computing the actual difference. Second, and more important, if we call k the number of atoms in a tgd, assume each atom can be mapped into n other atoms via homomorphisms; then we need to generate n^k different coverages, and therefore n^k differences.

This exponential bound on the number of coverages is not surprising. In fact, Gottlob and Nash have shown that the problem of computing core solutions is *fixed-parameter intractable*[13] wrt the size of the tgds (in fact, wrt the size of blocks), and therefore it is very unlikely that the exponential bound can be removed. We want to emphasize however that we are talking about expression complexity and not data complexity (the data complexity remains polynomial).

Despite this important difference in complexity between subsumptions and coverages, coverages can usually be handled quite efficiently. In brief, the exponential bound is reached only under rather unlikely conditions; to see why, recall that coverages tend to follow this pattern:

$$m_1 : A(a, b) \rightarrow R(a, b)$$
$$m_2 : B(a, b) \rightarrow S(a, b)$$
$$m_3 : C(a, b) \rightarrow \exists N : R(a, N), S(b, N)$$

Note that m_1 and m_2 write into the key–foreign key pair, while m_3 invents a value. Complexity may become an issue, here, only if the set of tgds contains a significant number of other tgds like m_1 and m_2 which write into R and S separately. This may happen only in those scenarios in which a very large number of different data sources with a poor design of foreign key relationships must be merged into the same target, which can hardly be considered a frequent case. In fact, in our experiments with both real-life scenarios and large randomly generated schemas, coverages have never been an issue.

Computing times are usually higher for scenarios with self-joins in tgd conclusions. In fact, the exponential bound is more severe in these cases. If we call n the number of atoms in tgd conclusions, since the construction of expansions requires to analyze all possible subsets of atoms in tgd con-

clusions,[8] a bound of 2^n is easily reached. Therefore, the number of joins, intersections and differences in the final SQL script may be very high. In fact, it is not difficult to design synthetic scenarios like the *RS* one discussed above that actually trigger the exponential explosion of rewritings.

However, in more realistic scenarios containing self-joins, the overhead is usually much lower. To understand why, let us note that expansions tend to increase when tgds are designed in such a way that it is possible for a tuple to perform a join with itself. In practice, this happens very seldom. Consider for example a *Person(name, father)* relation, in which children reference their father. It can be seen that no tuple in the *Person* table actually joins with itself. Similarly, in a *Gene(name, type, protein)* table, in which "synonym" genes refer to their "primary" gene via the protein attribute, since no gene is at the same time a synonym and a primary gene. In light of these ideas, we may

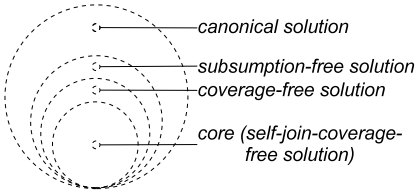

Figure 7: Containment of Solutions

say that, while it is true that the rewriting algorithm may generate expensive queries, this happens only in rather specific cases that hardly reflect practical scenarios. In practice, scalability is very good. In fact, we may say that the 90% of the complexity of the algorithm is needed to address a small minority of the cases. Our experiments confirm this intuition.

It is also worth noting that, when the complexity of the rewriting becomes high, our algorithms allows to produce several acceptable approximations of the core. In fact, the algorithm is modular in nature; when the core computation requires very high computing times and does not scale to large databases, the mapping designer may decide to discard the "full" rewriting, and select a "reduced" rewriting (i.e., a rewriting wrt to a subset of homomorphisms) to generate an *approximation* of the core more efficiently. This can be done by rewriting tgds with respect to subsumptions only or to subsumptions and coverages, as shown in Figure 7.

8. EXPERIMENTAL RESULTS

The algorithms introduced in the paper have been implemented in a working prototype written in Java. 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 the core even for large databases and complex scenarios. All experiments have been executed on a Intel Core 2 Duo machine with 2.4Ghz processor and 4 GB of RAM under Linux. The DBMS was PostgreSQL 8.3.

Computing Times. We start by comparing our algorithm with an implementation [20] of the core computation algorithm developed in [13], made available to us by the authors. In the following we will refer to this implementation as the "post-processing approach".

[8]In fact, all multisets.

We selected a set of seven experiments to compare execution times of the two approaches. The seven experiments include two scenarios with subsumptions, two with coverages, and three with self-joins in the target schema. The scenarios have been taken from the literature (two from [11], one from [22]), and from the STMark benchmark. Each test has been run with 10k, 100k, 250k, 500k, and 1M tuples in the source instance. On average we had 7 tables, with a minimum of 2 (for the *RS* example discussed in Section 6) and a maximum of 10.

A first evidence is that the post processing approach does not scale. We have been able to run experiments with 1k and 5k tuples, but starting at around 10k tuples the experiments took on average several hours. This result is not surprising, since these algorithms exhaustively look for endomorphisms in the canonical solution in order to remove variables (i.e, invented nulls). For instance, our first subsumption scenario with 5k tuples in the source generated 13500 variables in the target; the post-processing algorithm took on our machine running PostgreSQL around 7 hours to compute the final solution. It is interesting to note that in some cases the post processing algorithm finds the core after only one iteration (in the previous case, it took 3 hours), but the algorithm is not able to recognize this fact and stop the search. For all experiments, we fixed a timeout of 1 hour. If the experiment was not completed by that time, it was stopped. Since none of the scenarios we selected was executed in less than 1 hour we do not report computing times for the post-processing algorithm in our graphs. Execution times for the

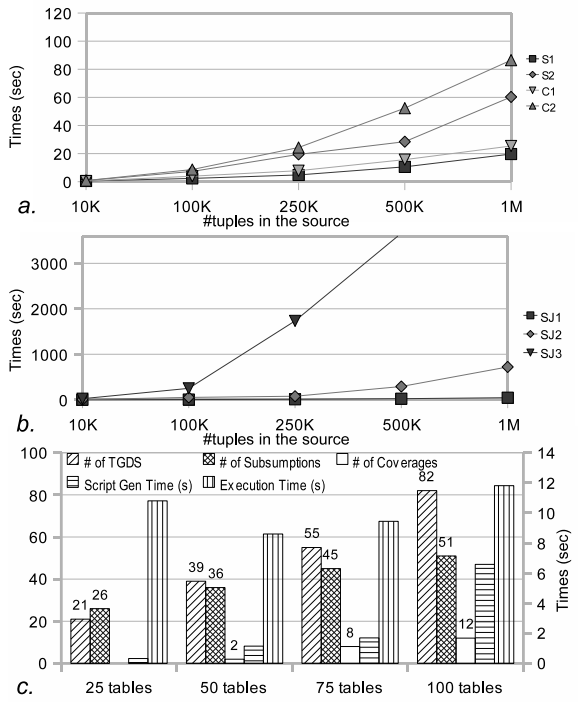

Figure 8: SQL Experiments

SQL scripts generated by our rewriting algorithms are reported in Figure 8. Figure 8.a shows executing times for the four scenarios that do not contain self-joins in the target; as it can be seen, execution times for all scenarios were below 2 minutes.

Figure 8.b reports times for the three self-join scenarios.

It can be seen that the *RS* example did not scale up to 1M tuples (computing the core for 500K tuples required 1 hour and 9 minutes). This is not surprising, given the exponential behavior discussed in the previous Section. However, the other two experiments with self-join – one from STMark and another from [22] – did scale nicely to 1M tuples.

Scalability on Large Scenarios. To test the scalability of our algorithm on schemas of large size we generated a set of synthetic scenarios using the scenario generator developed for the STMark benchmark. We generated four relational scenarios containing 20/50/75/100 tables, with an average join path length of 3, variance 1. Note that, to simulate real-application scenarios, we did not include self-joins. 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.

Figure 8.c summarizes the results. In the graph, we report several values. One is the number of tgds processed by the algorithm, with the number of subsumptions and coverages. Then, since we wanted to study how the tgd rewriting phase scales on large schemas, we measured the time needed to generate the SQL script. In all cases the algorithm was able to generate the SQL script in a few seconds. Finally, we report execution times in seconds for source databases of 100K tuples.

Nested Scenarios. All algorithms discussed in the previous sections are applicable to both flat and nested data. As it is common [18], the system adopts a nested relational model that can handle both relational and nested data sources (i.e, XML).

Note that data exchange research has so far concentrated on relational data. There is still no formal definition of a data exchange setting for nested data. Still, we compare the solutions produced by the system for nested scenarios with the ones generated by the basic [18] and the nested [12] mapping generation algorithms, which we have reimplemented in our prototype. We show that the rewriting algorithm invariably produces smaller solutions, without losing informative content.

For the first set of experiments we used two real data sets and a synthetic one. The first scenario maps a fragment of DBLP[9] to one of the Amalgam publication schemas[10]. The second scenario maps the Mondial database[11] to the CIA Factbook schema[12]. As a final scenario we used the *StatDB* scenario from [18] with synthetic random data. For each experiment we used three different input files with increasing size $(n, 2n, 4n)$.

Figure 9.a shows the percent reduction in the output size for our mappings compared to basic mappings (dashed line) and nested mappings. As output size, we measured the number of tuples, i.e., the number of sequence elements in the XML. Larger output files for the same scenario indicate more redundancy in the result. As expected, our approach outperformed basic mappings in all the examples. Nested mappings had mixed performance. In the first scenario they were able to compute a non-redundant solution. In the second scenario, they brought no benefits wrt basic mappings.

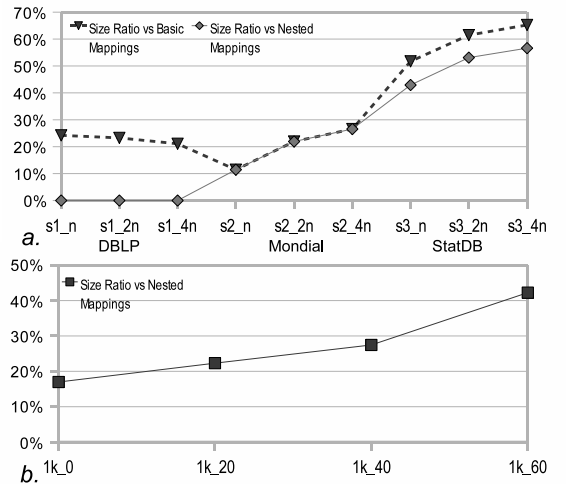

Figure 9: XML Experiments

Figure 9.b shows how the percent reduction changes with respect to the level of redundancy in the source data. We considered the statDB experiment, and generated several source instances of 1k tuples based on a pool of values of decreasing size. This generates different levels of redundancy (0/20/40/60%) in the source database. The reduction in the output size produced by the rewriting algorithm with respect to nested mappings increases almost linearly.

9. RELATED WORK

In this section we review some related works in the fields of schema mappings and data exchange.

The original schema mapping algorithm was introduced in [18] in the framework of the Clio project. The algorithm relies on a nested relational model to handle relational and XML data. The primary inputs are value correspondences and foreign key constraints on the two sources that are chased to build tableaux called logical relations; a tgd is produced for each source and target logical relations that cover at least one correspondence. Our tgd generation algorithm is a generalization of the basic mapping algorithm that captures a larger class of mappings, like self-joins [1] or those in [2]. Note that the need for explicit joins was first advocated in [19]; the duplication of symbols in the schemas has been first introduced in the MapForce commercial system (`www.altova.com/MapForce`).

The amount of redundancy generated by basic mappings has motivated a revision of the algorithm known as *nested mappings* [12]. Intuitively, whenever a tgd m_1 writes into an external target set R and a tgd m_2 writes into a set nested into R, it is possible to "merge" the two mappings by nesting m_2 into m_1. This reduces the amount of redundant tuples in the target. Unfortunately, nested mappings are applicable only in specific scenarios – essentially schema evolution problems in which the source and the target database have similar structures – and are not applicable in many of the examples discussed in this paper.

[9] http://dblp.uni-trier.de/xml
[10] http://www.cs.toronto.edu/~miller/amalgam
[11] http://www.dbis.informatik.uni-goettingen.de/Mondial
[12] https://www.cia.gov/library/publications/the-world-factbook

The notion of a core solution was first introduced in [11]; it represents a nice formalization of the notion of a "minimal" solution, since cores of finite structures arise in many areas of computer science (see, for example, [15]). Note that computing the core of an arbitrary instance is an intractable problem [11, 13]. However, we are not interested in computing cores for arbitrary instances, but rather for solutions of a data exchange problem; these show a number of regularities, so that polynomial-time algorithms exist.

In [11] the authors first introduce a polynomial greedy algorithm for core computation, and then a *blocks* algorithm. A block is a connected component in the Gaifman graph of nulls. The block algorithm looks at the nulls in J and computes the core of J by successively finding and applying a sequence of small *useful* endomorphisms; here, *useful* means that at least one null disappears. Only egds are allowed as target constraints.

The bounds are improved in [13]. The authors introduce various polynomial algorithms to compute cores in the presence of weakly-acyclic target tgds and arbitrary egds, that is, a more general framework than the one discussed in this paper. The authors prove two complexity bounds. Using an exhaustive enumeration algorithm they get an upper bound of $O(vm|dom(J)|^b)$, where v is the number of variables in J, m is the size of J, and b is the block size of J. There exist cases where a better bound can be achieved by relying on hypertree decomposition techniques. In such cases, the upper bound is $O(vm^{[b/2]+2})$, with special benefits if the target constraints of the data exchange scenario are LAV tgds. One of the algorithms introduced [13] has been revised and implemented in a working prototype [20]. The prototype uses a relational DBMS to chase tgds and egds, and a specialized engine to find endomorphisms and minimize the solution. Unfortunately, as discussed in Section 8, the technique does not scale to real size databases.

+Spicy is an evolution of the original Spicy mapping system [5], which was conceived as a platform to integrate schema matching and schema mappings, and represented one of the first attempt at the definition of a notion of *quality* for schema mappings.

10. CONCLUSIONS

We have introduced new algorithms for schema mappings that rely on the theoretical foundations of data exchange to generate optimal solutions.

From the theoretical viewpoint, it represents a step forward towards answering the following question: "is it possible to compute core solutions by using the chase ?" However, we believe that the main contribution of the paper is to show that, despite their intrinsic complexity, core solutions can be computed very efficiently in practical, real-life scenarios by using relational database engines.

+Spicy is the first mapping generation system that integrates a feasible implementation of a core computation algorithm into the mapping generation process. We believe that this represents a concrete advancement towards an explicit notion of quality for schema mapping systems.

Acknowledgments We would like to thank the anonymous reviewers for their comments that helped us to improve the presentation. Our gratitude goes also to Vadim Savenkov and Reinhard Pichler who made available to us an implementation of their post-processing core-computation algorithm, which proved very useful during the tests of the system. Finally, we are very grateful to Paolo Atzeni for all his comments and his advice.

11. REFERENCES

[1] B. Alexe, W. Tan, and Y. Velegrakis. Comparing and Evaluating Mapping Systems with STBenchmark. *Proc. of the VLDB Endowment*, 1(2):1468–1471, 2008.

[2] Y. An, A. Borgida, R. Miller, and J. Mylopoulos. A Semantic Approach to Discovering Schema Mapping Expressions. In *Proc. of ICDE*, pages 206–215, 2007.

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

[4] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting Context into Schema Matching. In *Proc. of VLDB*, pages 307–318. VLDB Endowment, 2006.

[5] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. Schema Mapping Verification: The Spicy Way. In *Proc. of EDBT*, pages 85 – 96, 2008.

[6] L. Bravo, W. Fan, and S. Ma. Extending Dependencies with Conditions. In *Proc. of VLDB*, pages 243–254, 2007.

[7] L. Cabibbo. On Keys, Foreign Keys and Nullable Attributes in Relational Mapping Systems. In *Proc. of EDBT*, pages 263–274, 2009.

[8] L. Chiticariu. Computing the Core in Data Exchange: Algorithmic Issues. MS Project Report, 2005. Unpublished manuscript.

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

[10] R. Fagin, P. Kolaitis, A. Nash, and L. Popa. Towards a Theory of Schema-Mapping Optimization. In *Proc. of ACM PODS*, pages 33–42, 2008.

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

[12] A. Fuxman, M. A. Hernández, C. T. Howard, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *Proc. of VLDB*, pages 67–78, 2006.

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

[14] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *Proc. of VLDB*, pages 675–686, 2007.

[15] P. Hell and J. Nešetřil. The Core of a Graph. *Discrete Mathematics*, 109(1-3):117–126, 1992.

[16] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[17] R. J. Miller, L. M. Haas, and M. A. Hernandez. Schema Mapping as Query Discovery. In *Proc. of VLDB*, pages 77–99, 2000.

[18] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *Proc. of VLDB*, pages 598–609, 2002.

[19] A. Raffio, D. Braga, S. Ceri, P. Papotti, and M. A. Hernández. Clip: a Visual Language for Explicit Schema Mappings. In *Proc. of ICDE*, pages 30–39, 2008.

[20] V. Savenkov and R. Pichler. Towards practical feasibility of core computation in data exchange. In *Proc. of LPAR*, pages 62–78, 2008.

[21] B. ten Cate, L. Chiticariu, P. Kolaitis, and W. C. Tan. Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries. Unpublished manuscript – http://arxiv.org/abs/0903.1953, March 2009.

[22] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data Driven Understanding and Refinement of Schema Mappings. In *Proc. of ACM SIGMOD*, pages 485–496, 2001.