Towards Practical Feasibility of Core Computation in Data Exchange*

Reinhard Pichler and Vadim Savenkov

Vienna University of Technology

Abstract. Core computation in data exchange is concerned with materializing the minimal target database for a given source database. Gottlob and Nash have recently shown that the core can be computed in polynomial time under very general conditions. Nevertheless, core computation has not yet been incorporated into existing data exchange tools. The principal aim of this paper is to make a big step forward towards the practical feasibility of core computation in data exchange by developing an improved algorithm and by presenting a prototype implementation of our new algorithm.

1 Introduction

Data exchange is concerned with the transfer of data between databases with different schemas. This transfer should be performed so that the source-to-target dependencies (STDs) establishing a mapping between the two schemas are satisfied. Moreover, the target database may also impose additional integrity constraints, called target dependencies (TDs). As STDs and TDs, we consider so-called *embedded dependencies* [1], which are first-order formulae of the form $\forall \boldsymbol{x}\,(\phi(\boldsymbol{x}) \rightarrow \exists \boldsymbol{y}\ \psi(\boldsymbol{x}, \boldsymbol{y}))$ where ϕ and ψ are conjunctions of atomic formulas or equalities, and all variables in $\boldsymbol{x}$ do occur in $\phi(\boldsymbol{x})$. Throughout this paper, we shall omit the universal quantifiers. By convention, all variables occurring in the premise are universally quantified. Moreover, we shall often also omit the existential quantifiers. By convention, all variables occurring in the conclusion only are existentially quantified over the conclusion. We shall thus use the notations $\phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y})$ and $\phi(\boldsymbol{x}) \rightarrow \exists \boldsymbol{y}\ \psi(\boldsymbol{x}, \boldsymbol{y})$ interchangeably for the above formula.

The source schema $\mathbf{S}$ and the target schema $\mathbf{T}$ together with the set Σ_{st} of STDs and the set Σ_t of TDs constitute the *data exchange setting* $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$. Following [2,3], we consider dependencies of the following forms: Each STD is a *tuple generating dependency* (TGD) [4] of the form $\phi_{\mathbf{S}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$, where $\phi_{\mathbf{S}}(\boldsymbol{x})$ is a conjunction of atomic formulas over $\mathbf{S}$ and $\psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$ is a conjunction of atomic formulas over $\mathbf{T}$. Each TD is either a TGD, of the form $\phi_{\mathbf{T}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$ or an *equality generating dependency* (EGD) [4] of the form $\phi_{\mathbf{T}}(\boldsymbol{x}) \rightarrow (x_i = x_j)$. In these dependencies, $\phi_{\mathbf{T}}(\boldsymbol{x})$ and $\psi_{\mathbf{T}}(\boldsymbol{x}, \boldsymbol{y})$ are conjunctions of atomic formulas over $\mathbf{T}$, and x_i, x_j are among the variables in $\boldsymbol{x}$. An important special case of

* This work was supported by the Austrian Science Fund (FWF), project P20704-N18.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 62–78, 2008.

TGDs are *full TGDs*, which have no (existentially quantified) variables $\boldsymbol{y}$, i.e. we have $\phi_{\mathbf{S}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x})$ and $\phi_{\mathbf{T}}(\boldsymbol{x}) \rightarrow \psi_{\mathbf{T}}(\boldsymbol{x})$, respectively.

The *data exchange problem* for a data exchange setting $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ is the task of constructing a target instance J for a given source instance I, s.t. all STDs Σ_{st} and TDs Σ_t are satisfied. Such a J is called a *solution*. Typically, the number of possible solutions to a data exchange problem is infinite.

Example 1. Suppose that the source instance consists of two relations Tutorial(course, tutor): {('java', 'Yves')} and BasicUnit(course): {'java'}. Moreover, let the target schema have four relation symbols NeedsLab(id_tutor,lab), Tutor(idt,tutor), Teaches(id_tutor, id_course) and Course(idc,course). Now suppose that we have the following STDs:

1. $\mathsf{BasicUnit}(C) \rightarrow \mathsf{Course}(Idc, C)$.
2. $\mathsf{Tutorial}(C, T) \rightarrow \mathsf{Course}(Idc, C), \mathsf{Tutor}(Idt, T), \mathsf{Teaches}(Idt, Itc)$.

and suppose that the TDs are given by the two TGDs:

3. $\mathsf{Course}(Idc, C) \rightarrow \mathsf{Tutor}(Idt, T), \mathsf{Teaches}(Idt, Idc)$.
4. $\mathsf{Teaches}(Idt, Idc) \rightarrow \mathsf{NeedsLab}(Idt, L)$.

Then the following instances are all valid solutions:

$J = \{\mathsf{Course}(C_1, \text{'java'}), \mathsf{Tutor}(T_2,N), \mathsf{Teaches}(T_2,C_1), \mathsf{NeedsLab}(T_2,L_2),$
$\mathsf{Course}(C_2, \text{'java'}), \mathsf{Tutor}(T_1,\text{'Yves'}), \mathsf{Teaches}(T_1,C_2), \mathsf{NeedsLab}(T_1,L_1)\},$

$J_c = \{\mathsf{Course}(C_1,\text{'java'}), \mathsf{Tutor}(T_1,\text{'Yves'}), \mathsf{Teaches}(T_1,C_1), \mathsf{NeedsLab}(T_1,L_1)\},$

$J' = \{\mathsf{Course}(\text{'java'},\text{'java'}), \mathsf{Tutor}(T_1,\text{'Yves'}), \mathsf{Teaches}(T_1,\text{'java'}), \mathsf{NeedsLab}(T_1,L_1)\}$

A natural requirement (proposed in [2]) on the solutions is *universality*, that is, there should be a homomorphism from the materialized solution to any other possible solution. Note that J' in Example 1 is not universal, since there exists no homomorphism $h\colon J' \rightarrow J$. Indeed, a homomorphism maps any constant onto itself; thus, the fact $\mathsf{Course}(\text{'java'},\text{'java'})$ cannot be mapped onto a fact in J.

In general, a data exchange problem has several universal solutions, which may significantly differ in size. However, there is – up to isomorphism – one particular, universal solution, called the *core* [3], which is the most compact one. For instance, solution J_c in Example 1 is a core.

Fagin et al. [3] gave convincing arguments that the core should be the database to be materialized. In general, computing the core of a graph or a structure is NP-complete [5]. However, Gottlob and Nash [6] showed that the core of the target database in data exchange can be computed in polynomial time under very general conditions. Despite this favorable complexity result, core computation has not yet been incorporated into existing data exchange tools. This is mainly due to the following reasons: (1) Despite the theoretical tractability of core computation, we are still far away from a practically efficient implementation of core computation. In fact, no implementation at all has been reported so far. (2) The core computation looks like a separate technology which cannot be easily integrated into existing database technology.

Results. The main contribution of this work is twofold:

(1) We present an enhanced version of the FindCore algorithm, which we shall refer to as FindCore^E. One of the specifics of FindCore is that EGDs in the target dependencies are simulated by TGDs. As a consequence, the core computation becomes an integral part of finding any solution to the data exchange problem. The most significant advantage of our FindCore^E algorithm is that it avoids the simulation of EGDs by TGDs. The activities of solving the data exchange problem and of computing the core are thus fully uncoupled. The core computation can then be considered as an optional add-on feature of data exchange which may be omitted or deferred to a later time (e.g., to periods of low database user activity). Moreover, the direct treatment of EGDs leads to a performance improvement of an order of magnitude. Another order of magnitude can be gained by approximating the core. Our experimental results suggest that the partial execution of the core computation may already yield a very good approximation to the core. Since all intermediate instances computed by our FindCore^E algorithm are universal solutions, one may stop the core computation at any time and content oneself with an approximation to the core.

(2) We also report on a proof-of-concept implementation of our enhanced algorithm. It is built on top of a relational database system and mimics data exchange-specific features by automatically generated views and SQL queries. This shows that the integration of core computation into existing database technology is clearly feasible. The lessons learned from the experiments with this implementation yield important hints concerning future improvements.

Due to lack of space, most proofs are sketched or even omitted in this paper. For full proofs, we refer to [7].

2 Preliminaries

2.1 Basic Notions

Schemas and instances. A *schema* $\sigma = \{R_1, \ldots, R_n\}$ is a set of relation symbols R_i with fixed arities. An *instance* over a schema σ consists of a relation for each relation symbol in σ, s.t. both have the same arity. Tuples of the relations may contain two types of *terms*: *constants* and *variables*. The latter are also called *labeled nulls*. Two labeled nulls are equal iff they have the same label. For every instance J, we write $\mathrm{dom}(J)$, $\mathrm{var}(J)$, and $\mathrm{const}(J)$ to denote the set of terms, variables, and constants, respectively, of J. Clearly, $\mathrm{dom}(J) = \mathrm{var}(J) \cup \mathrm{const}(J)$ and $\mathrm{var}(J) \cap \mathrm{const}(J) = \emptyset$. If a tuple $(x_1, x_2, \ldots, x_n)$ belongs to the relation R, we say that J contains the *fact* $R(x_1, x_2, \ldots, x_n)$. We write $\boldsymbol{x}$ for a tuple $(x_1, x_2, \ldots, x_n)$ and if $x_i \in X$, for every i, then we also write $\boldsymbol{x} \in X$ instead of $\boldsymbol{x} \in X^n$. Likewise, we write $r \in \boldsymbol{x}$ if $r = x_i$ for some i. Let Σ be an arbitrary set of dependencies and J an instance. We write $J \models \Sigma$ to denote that the instance J satisfies Σ. In a data exchange setting $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$, the source schema $\mathbf{S}$ and the target schema $\mathbf{T}$ have no relation symbols in common. In a source instance I, no variables are allowed, i.e., $\mathrm{dom}(I) = \mathrm{const}(I)$.

Chase. The data exchange problem can be solved by the *chase* [4], which iteratively introduces new facts or equates terms until all desired dependencies are fulfilled. More precisely, let Σ contain a TGD $\tau\colon \phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x},\boldsymbol{y})$, s.t. $I \models \phi(\boldsymbol{a})$ for some assignment $\boldsymbol{a}$ on $\boldsymbol{x}$ and $I \not\models \exists \boldsymbol{y}\psi(\boldsymbol{a},\boldsymbol{y})$. Then we have to extend I with facts corresponding to $\psi(\boldsymbol{a},\boldsymbol{z})$, where the elements of $\boldsymbol{z}$ are fresh labeled nulls. Likewise, suppose that Σ contains an EGD $\tau\colon \phi(\boldsymbol{x}) \rightarrow x_i = x_j$, s.t. $I \models \phi(\boldsymbol{a})$ for some assignment $\boldsymbol{a}$ on $\boldsymbol{x}$. This EGD enforces the equality $a_i = a_j$. We thus choose a variable v among a_i, a_j and replace *every occurrence* of v in I by the other term; if $a_i, a_j \in \mathrm{const}(I)$ and $a_i \neq a_j$, the chase halts with *failure*. The result of chasing I with dependencies Σ is denoted as I^{Σ}.

A sufficient condition for the termination of the chase is that the TGDs be *weakly acyclic* (see [8,2]). This property is formalized as follows. For a dependency set Σ, construct a *dependency graph* G^D whose vertices are *fields* R^i where i denotes a position (an "attribute") of relation R. Let $\phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x},\boldsymbol{y})$ be a TGD in Σ and suppose that some variable $x \in \boldsymbol{x}$ occurs in the field R^i. Then the edge $\left(R^i, S^j\right)$ is present in G^D if either (1) x also occurs in the field S^j in $\psi(\boldsymbol{x},\boldsymbol{y})$ or (2) x occurs in some other field T^k in $\psi(\boldsymbol{x},\boldsymbol{y})$ and there is a variable $y \in \boldsymbol{y}$ in the field S^j in $\psi(\boldsymbol{x},\boldsymbol{y})$. Edges resulting from rule (2) are called *special*.

A set of TGDs is *weakly acyclic* if there is no cycle containing a special edge. Obviously, the set of STDs is always weakly acyclic, since the dependency graph contains only edges from fields in the source schema to fields in the target schema. We thus consider data exchange settings $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ where Σ_{st} is a set of TGDs and Σ_t is a set of EGDs and *weakly acyclic* TGDs.

Figure 1 shows the dependency graph for the *target* TGDs in Example 1. Special edges are marked with *. Source-to-target TGDs are omitted, since they can never produce a cycle. Figure 1 contains two kinds of vertices: the ovals, labelled by attributes of target relations, are the actual vertices of the dependency graph. The rectangles, labelled by relation names, were inserted to improve the readability. Rather than adding the relation names to the attributes in the labels of the oval vertices, we have connected each attribute to the rectangular vertex with the corresponding relation name. Clearly, this dependency graph has no cycle containing a special edge. Hence, these TGDs are weakly acyclic.

Universal solutions and core. Let I,I' be instances. A *homomorphism* $h\colon I \rightarrow I'$ is a mapping $\mathrm{dom}(I) \rightarrow \mathrm{dom}(I')$, s.t. (1) whenever $R(\boldsymbol{x}) \in I$, then $R(h(\boldsymbol{x})) \in I'$, and (2) for every constant c, $h(c) = c$. An *endomorphism* is a homomorphism $I \rightarrow I$, and a *retraction* r is an idempotent endomorphism, i.e. $r \circ r = r$. An endomorphism or a retraction is *proper* if it is not surjective (for finite instances, this is equivalent to being not injective). The image $r(I)$ under a retraction r is called a *retract* of I. An instance is called a *core* if it has no proper retractions. A core C of an instance I is a retract of I, s.t. C is a core. Cores of an instance I are unique up to isomorphism. We can therefore speak about *the core* of I.

Consider a data exchange setting where Σ_{st} is a set of TGDs and Σ_t is a set of EGDs and weakly acyclic TGDs. Then the solution to a source instance S can be computed as follows: We start off with the instance $(S, \emptyset)$, i.e., the source instance is S and the target instance is initially empty. Chasing $(S, \emptyset)$

with Σ_{st} yields the instance (S, T), where T is called the *preuniversal instance*. This chase always succeeds since Σ_{st} contains no EGDs. Then T is chased with Σ_t. This chase may fail because of the EGDs in Σ_t. If the chase succeeds, then we end up with $U = T^{\Sigma_t}$, which is referred to as the *canonical universal solution*. Both T and U can be computed in polynomial time w.r.t. the size of the source instance [2].

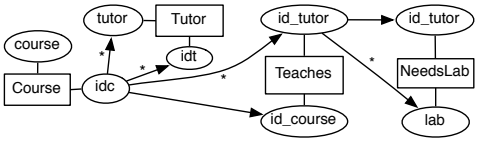

Fig. 1. Dependency graph

Depth, height, width, blocks. Let Σ be a set of dependencies with dependency graph G^D. The *depth* of a field R^j of a relation symbol R is the maximal number of special edges in any path of G^D that ends in R^j. The depth of Σ is the maximal depth of any field in Σ. Given a dependency $\tau : \phi(\boldsymbol{x}) \to \psi(\boldsymbol{x}, \boldsymbol{y})$ in Σ, we define the *width* of τ to be $|\boldsymbol{x}|$, and the *height* as $|\boldsymbol{y}|$. The width (resp. the height) of Σ is the maximal width (resp. height) of the dependencies in Σ.

Core computation is essentially a search for appropriate homomorphisms, whose key complexity factor is the *block size* [3]. It is defined as follows: The *Gaifman graph* $\mathcal{G}(I)$ of an instance I is an undirected graph whose vertices are the variables of I and, whenever two variables v_1 and v_2 share a tuple in I, there is an edge (v_1, v_2) in $\mathcal{G}(I)$. A *block* is a connected component of $\mathcal{G}(I)$. Every variable v of I belongs exactly to one block, denoted as $\text{block}(v, I)$. The *block size* of instance I is the maximal number of variables in any of its blocks.

Theorem 1. [3] *Let A and B be instances, and suppose that* $\text{blocksize}(A) \leq c$ *holds. Then the check if a homomorphism $h\colon A \to B$ exists and, if so, the computation of h can both be done in time $O(|A| \cdot |B|^c)$.*

Theorem 2. [3] *If Σ_{st} is a set of STDs of height e, S is ground, and $(S, T) = (S, \emptyset)^{\Sigma_{st}}$, then* $\text{blocksize}(T) \leq e$.

Sibling, parent, ancestor. Consider the chase of the preuniversal instance T with TDs Σ_t and suppose that $\boldsymbol{y}$ is a tuple of variables created by enforcing a TGD $\phi(\boldsymbol{x}) \to \psi(\boldsymbol{x}, \boldsymbol{y})$ in Σ_t, s.t. the precondition $\phi(\boldsymbol{x})$ was satisfied with a tuple $\boldsymbol{a}$. Then the elements of $\boldsymbol{y}$ are *siblings* of each other; every variable of $\boldsymbol{a}$ is a *parent* of every element of $\boldsymbol{y}$; and the *ancestor* relation is the transitive closure of the parent relation.

2.2 Core Computation with FindCore

In this section, we recall the FindCore algorithm of [6]. To this end, we briefly explain the main ideas underlying the steps (1) – (11) of this algorithm.

The chase. FindCore starts in (1) with the computation of the preuniversal instance. But then, rather than directly computing the canonical universal solution by chasing T with Σ_t, the EGDs in Σ_t are simulated by TGDs. Hence,

in (2), the set Σ_t of EGDs and TGDs over the signature τ is transformed into the set $\bar{\Sigma}_t$ of TGDs over the signature $\tau \cup E$, where E (encoding equality) is a binary relation not present in τ. The transformation proceeds as follows:

1. Replace all equations $x = y$ with $E(x, y)$, turning every EGD into a TGD.
2. Add *equality* constraints (symmetry, transitivity, reflexivity): (i) $E(x, y) \rightarrow E(y, x)$; (ii) $E(x, y), E(y, z) \rightarrow E(x, z)$; and (iii) $R(x_1, \ldots, x_k) \rightarrow E(x_i, x_i)$ for every $R \in \tau$ and $i \in \{1, 2, \ldots, k\}$ where k is the arity of R.
3. Add *consistency* constraints: $R(x_1, \ldots, x_k), E(x_i, y) \rightarrow R(x_1, \ldots, y, \ldots, x_k)$ for every $R \in \tau$ and $i \in \{1, 2, \ldots, k\}$.

Procedure FindCore
Input: Source ground instance S
Output: Core of a universal solution for S

```
(1)  Chase (S,∅) with Σ_st
     to obtain (S,T) := (S,∅)^Σst;
(2)  Compute Σ̄_t from Σ_t;
(3)  Chase T with Σ̄_t (using a nice order)
     to get U := T^Σ̄t;
(4)  for each x ∈ var(U), y ∈ dom(U), x ≠ y do
(5)      Compute T_xy;
(6)      Look for h: T_xy → U s.t. h(x) = h(y);
(7)      if there is such h then
(8)          Extend h to an endomorphism h' on U;
(9)          Transform h' into a retraction r;
(10)         Set U := r(U);
(11) return U.
```

Even if Σ_t was weakly acyclic, $\bar{\Sigma}_t$ may possibly not be so. Hence, a special *nice chase order* is defined in [6] which ensures termination of the chase by $\bar{\Sigma}_t$. It should be noted that U computed in (3) is not a universal solution since, in general, the EGDs of Σ_t are not satisfied. Their enforcement happens as part of the core computation.

Retractions. The FindCore algorithm computes the core by computing nested retracts. This is motivated by the following properties of retractions: (1) embedded dependencies are closed under retractions and (2) any proper endomorphism can be efficiently transformed into a retraction [6]:

Theorem 3. [6] *Let $r\colon A \rightarrow A$ be a retraction with $B = r(A)$ and let Σ be a set of embedded dependencies. If $A \models \Sigma$, then $B \models \Sigma$.*

Theorem 4. [6] *Given an endomorphism $h\colon A \rightarrow A$ such that $h(x) = h(y)$ for some $x, y \in \mathrm{dom}(A)$, there is a proper retraction r on A s.t. $r(x) = r(y)$. Such a retraction can be found in time $O(|\mathrm{dom}(A)|^2)$.*

Note that U after step (3) clearly satisfies the dependencies Σ_{st} and $\bar{\Sigma}_t$. Steps (4) – (8), which will be explained below, search for a proper endomorphism h on U. If this search is successful, we use Theorem 4 to turn h into a retraction r in step (9) and replace U by $r(U)$ in step (10). By Theorem 3 we know that Σ_{st} and $\bar{\Sigma}_t$ are still satisfied.

Searching for proper endomorphisms. At every step of the descent to the core, the FindCore algorithm attempts to find a proper endomorphism for the current instance U in the steps (5) – (8) of the algorithm. Given a variable x and another domain element y, we try to find an endomorphism which equates x and y. However, by Theorem 1, the time needed to find an appropriate homomorphism may be exponential w.r.t. the block size. The key idea in FindCore

is, therefore, to split the search for a proper endomorphism into two steps: For given x and y, there exists an instance T_{xy} (defined below) whose block size is bounded by a constant depending only on $\Sigma_{st} \cup \Sigma_t$. So we first search for a homomorphism $h\colon T_{xy} \to U$ with $h(x) = h(y)$; and then h is extended to a homomorphism $h\colon U \to U$, s.t. $h(x) = h(y)$ still holds. Hence, h is still non-injective and, thus, h is a *proper* endomorphism, since we only consider finite instances. The properties of T_{xy} and the existence of an extension h' of h are governed by the following results from [6]:

Lemma 1. [6] *For every weakly acyclic set Σ of TGDs, instance T and $x, y \in dom(T^\Sigma)$, there exist constants b, c which depend only on Σ and an instance T_{xy} satisfying* (1) $x, y \in \mathrm{dom}(T_{xy})$, (2) $T \subseteq T_{xy} \subseteq T^\Sigma$, (3) $\mathrm{dom}(T_{xy})$ *is closed under parents and siblings, and* (4) $|\mathrm{dom}(T_{xy})| \leq |\mathrm{dom}(T)| + b$. *Moreover, T_{xy} can be computed in time $O(|\mathrm{dom}(T)|^c)$.*

Theorem 5. (Lifting) [6] *Let T^Σ be a universal solution of a data exchange problem obtained by chasing a preuniversal instance T with the weakly acyclic set Σ of target TGDs. If B and W are instances such that:* (1) $B \models \Sigma$, (2) $T \subseteq W \subseteq T^\Sigma$, *and* (3) $\mathrm{dom}(W)$ *is closed under ancestors and siblings, then any homomorphism $h\colon W \to B$ can be extended in time $O(|\mathrm{dom}(T)|^b)$ to a homomorphism $h'\colon T^\Sigma \to B$ where b depends only on Σ.*

Summary. Recall that the predicate E simulates equality. Hence, if step (3) of the algorithm generates a fact $E(a_i, a_j)$ with $a_i \neq a_j$ then the data exchange problem has no solution. Otherwise, the loop in steps (4) – (10) tries to successively shrink $\mathrm{dom}(U)$. When no further shrinking is possible, then the core is reached. In fact, it is proved in [6] that such a minimal instance U resulting from FindCore indeed satisfies all the EGDs. Hence, U minus all auxiliary facts with leading symbol E constitutes the core of a universal solution. Moreover, it is proved in [6] that the entire computation fits into $O(|\mathrm{dom}(S)|^b)$ time for some constant b which depends only on the dependencies $\Sigma_{st} \cup \Sigma_t$.

3 Enhanced Core Computation

The crucial point of our enhanced algorithm FindCore^E is the *direct* treatment of the EGDs, rather than simulating them by TGDs. Hence, our algorithm produces the canonical universal solution U first (or detects that no solution exists), and then successively minimizes U to the core. On the surface, our FindCore^E algorithm proceeds exactly as the FindCore algorithm from Section 2.2 algorithm, i.e.: `(i)` compute an instance T_{xy}; `(ii)` search for a non-injective homomorphism $h\colon T_{xy} \to U$; `(iii)` lift h to a proper endomorphism $h'\colon U \to U$; and `(iv)` construct a proper retraction r from h'.

Actually, the construction of a retraction r via Theorem 4 and the closure of embedded dependencies w.r.t. retractions according to Theorem 3 are not affected by the application of the EGDs. In contrast, the first 3 steps above require significant adaptations in order to cope with EGDs:

(i) T_{xy} in Section 2.2 is obtained by considering only a small portion of the target chase, thus producing a subinstance of U. Now that EGDs are involved, the domain of U may no longer contain all elements that were present in T or in some intermediate result of the chase. Hence, we need to define T_{xy} differently.

(ii) The computational cost of the search for a homomorphism $h\colon T_{xy} \to U$ depends on the block size of T_{xy} which in turn depends on the block size of the preuniversal instance T. EGDs have a positive effect in that they eliminate variables, thus reducing the size of a single block. However, EGDs may also merge different blocks of T. Hence, without further measures, this would destroy the tractability of the search for a homomorphism $h\colon T_{xy} \to U$.

(iii) Since T_{xy} is defined differently from Section 2.2, also the lifting of $h\colon T_{xy} \to U$ to a proper endomorphism $h'\colon U \to U$ has to be modified. Moreover, it will turn out that a completely new approach is needed to prove the correctness of this lifting. The details of the FindCore^E algorithm are worked out below.

Introduction of an id. Chasing with EGDs results in the substitution of variables. Hence, the application of an EGD to an instance J produces a syntactically different instance J'. However, we find it convenient to regard the instance J' after enforcement of an EGD as a *new version* of the instance J rather than as a completely new instance. In other words, the substitution of a variable produces new versions of facts that have held that variable, but the facts themselves persist. We formalize this idea as follows: Given a data exchange setting $S = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$, we define an *id-aware data exchange setting* S^{id} by augmenting each relation $R \in \mathbf{T}$ with an additional *id* field inserted at position 0. Hence, in the atoms of the conclusions of STDs and in all atoms occurring in TDs, we have to add a unique existentially-quantified variable at position 0. For example, the source-to-target TGD $\tau\colon S(x) \to R(x, y)$ is transformed into $\tau^{id}\colon S(x) \to R^{id}(t, x, y)$ for some fresh variable t.

These changes neither have an effect on the chase nor on the core computation, as no rules rely on values in the added columns. It is immediate that a fact $R(x_1, x_2, \ldots, x_n)$ is present in the target instance at some phase of solving the original data exchange problem iff the fact $R^{id}(id, x_1, x_2, \ldots, x_n)$ is present at the same phase of solving its *id-aware* version. In fact, this modification does not even need to be implemented – we just introduce it to allow the discussion about facts in an unambiguous way.

During the chase, every fact of the target instance is assigned a unique *id* variable, which is never substituted by an EGD. We can therefore identify a fact with this variable: (1) if $R^{id}(t_1, x_1, \ldots, x_n)$ is a fact of a target instance $\mathbf{T}$, then we refer to it as *fact* t_1; (2) we define equality on facts as equality between their *id* terms: $R^{id}(t_1, x_1, \ldots, x_n) = R^{id}(t_2, y_1, \ldots, y_n)$ iff $t_1 = t_2$.

We also define a *position* by means of the *id* of a fact plus a positive integer indicating the place of this position inside the fact. Thus, if J is an instance and $R(id_R, x_1, x_2, \ldots, x_n)$ is an *id-aware* version of $R(x_1, \ldots, x_n) \in J$, then we say that the term x_i occurs at the position (id_R, i) in J.

Source position and origin. By the above considerations, facts and positions in an *id-aware data exchange setting*, persist in the instance once they have

been created – in spite of possible modifications of the variables. New facts and, therefore, new positions in the target instance are introduced by TGDs. If a position $p = (id_R, i)$ occurring in the fact $R(id_R, x_1, \ldots, x_n)$ was created to hold a fresh variable, we call p *native* to its fact id_R. Otherwise, if an already existing variable was copied from some position p' in the premise of the TGD to p, then we say that p is *foreign* to its fact id_R. Moreover, we call p' the *source position* of p. Note that there may be multiple choices for a source position. For instance, in the case of the TGD $R(y, x) \wedge S(x) \rightarrow P(x)$: a term of $P/1$ may be copied either from $R/2$ or from $S/1$. Any possibility can be taken in such a case: the choice is *don't care non-deterministic*.

Of course, a source position may itself be foreign to its fact. Tracing the chain of source positions back until we reach a native position leads to the notion of *origin position*, which we define recursively as follows: If a position $p = (id_R, i)$ is native to the fact $R(id_R, x_1, \ldots, x_n)$, then its origin position is p itself. Otherwise, if p is foreign, then the origin of p is the origin of a *source position* of p.

The fact holding the origin position of p is referred to as the *origin fact of the position* p. Finally, we define the *origin fact of a variable* x, denoted as $Origin_x$, as the origin fact of one of the positions where it was first introduced (again in a don't care non-deterministic way).

Example 2. Let $J = \{S(id_{S1}, x_1, y_1)\}$ be a preuniversal instance, and consider the TDs $\{S(id_S, x, y) \rightarrow P(id_P, y, z); P(id_P, y, z) \rightarrow Q(id_Q, y, v)\}$ yielding the canonical solution $J^\Sigma = \{S(id_{S1}, x_1, y_1), P(id_{P1}, y_1, z_1), Q(id_{Q1}, y_1, v_1)\}$ in Figure 2. Every position of J is native, being created by the source-to-target chase, which never copies labeled nulls. Thus the origin positions of $(id_{S1}, 1)$ and $(id_{S1}, 2)$ are these positions themselves. The latter is also the origin position for the two foreign positions $(id_{P1}, 1)$ and $(id_{Q1}, 1)$, introduced by the target chase (foreign positions are dashed in the figure). The remaining two positions of the facts id_{P1} and id_{Q1} are native. The origin positions of the variables are: $(id_{S1}, 1)$ for x_1, $(id_{S1}, 2)$ for y_1, $(id_{P1}, 2)$ for z_1, and $(id_{Q1}, 2)$ for v_1.

Lemma 2. *Let I be an instance. Moreover, let p be a position in I and o_p its origin position. Then p and o_p always contain the same term.*

Normalization of TGDs. Let $\tau\colon \phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y})$ be a non-full TGD, i.e., $\boldsymbol{y}$ is non-empty. Then we can set up the *Gaifman graph* $\mathcal{G}(\tau)$ of the atoms in the conclusion $\psi(\boldsymbol{x}, \boldsymbol{y})$, considering only the new variables $\boldsymbol{y}$, i.e., $\mathcal{G}(\tau)$ contains as vertices the variables in $\boldsymbol{y}$. Moreover, two variables y_i and y_j are adjacent

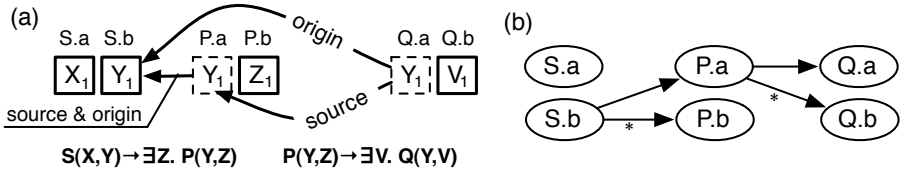

Fig. 2. Positions of the instance J^Σ (a) and the dependency graph of Σ (b)

(by slight abuse of notation, we identify vertices and variables), if they jointly occur in some atom of $\psi(\boldsymbol{x}, \boldsymbol{y})$. Let $\mathcal{G}(\tau)$ contain the connected components $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$. Then the conclusion is of the form $\psi(\boldsymbol{x}, \boldsymbol{y}) = \psi_0(\boldsymbol{x}) \wedge \psi_1(\boldsymbol{x}, \boldsymbol{y}_1) \wedge \cdots \wedge \psi_n(\boldsymbol{x}, \boldsymbol{y}_n)$, where the subformula $\psi_0(\boldsymbol{x})$ contains all atoms of $\psi(\boldsymbol{x}, \boldsymbol{y})$ without variables from $\boldsymbol{y}$ and each subformula $\psi_i(\boldsymbol{x}, \boldsymbol{y}_i)$ contains exactly the atoms of $\psi(\boldsymbol{x}, \boldsymbol{y})$ containing at least one variable from the connected component $\boldsymbol{y}_i$.

Now let the full TGD τ_0 be defined as $\tau_0 \colon \phi(\boldsymbol{x}) \rightarrow \psi_0(\boldsymbol{x})$ and let the non-full TGDs τ_i with $i \in \{1, \ldots, n\}$ be defined as $\tau_i \colon \phi(\boldsymbol{x}) \rightarrow \psi_i(\boldsymbol{x}, \boldsymbol{y}_i)$. Then τ is clearly logically equivalent to the conjunction $\tau_0 \wedge \tau_1 \wedge \cdots \wedge \tau_n$. Hence, τ in the set Σ_t of target dependencies may be replaced by $\tau_0, \tau_1, \ldots, \tau_n$.

We say that Σ_t is in *normal form* if every TGD τ in Σ_t is either full or its Gaifman graph $\mathcal{G}(\tau)$ has exactly 1 connected component. By the above considerations, we will henceforth assume w.l.o.g., that Σ_t is in normal form.

Example 3. The non-full TGD $\tau \colon S(x, y) \rightarrow \exists z, v(P(x, z) \wedge R(x, y) \wedge Q(y, v))$ is logically equivalent to the conjunction of the three TGDs: $\tau_0 \colon S(x, y) \rightarrow R(x, y)$, $\tau_1 \colon S(x, y) \rightarrow \exists z\ P(x, z)$, and $\tau_2 \colon S(x, y) \rightarrow \exists v\ Q(y, v)$. Clearly, these dependencies τ_0, τ_1, and τ_2 are normalized in the above sense.

Extension of the parent and sibling relation to facts. Let I be an instance after the j^{th} chase step and suppose that in the next chase step, the *non-full* TGD $\tau \colon \phi(\boldsymbol{x}) \rightarrow \psi(\boldsymbol{x}, \boldsymbol{y})$ is enforced, i.e.: $I \models \phi(\boldsymbol{a})$ for some assignment $\boldsymbol{a}$ on $\boldsymbol{x}$ and $I \not\models \exists \boldsymbol{y} \psi(\boldsymbol{a}, \boldsymbol{y})$, s.t. the facts corresponding to $\psi(\boldsymbol{a}, \boldsymbol{z})$, where the elements of $\boldsymbol{z}$ are fresh labeled nulls, are added. Let t be a fact introduced by this chase step, i.e., t is an atom of $\psi(\boldsymbol{a}, \boldsymbol{z})$. Then all other facts introduced by the same chase step (i.e., all other atoms of $\psi(\boldsymbol{a}, \boldsymbol{z})$) are the *siblings* of t. The *parent* set of a fact t consists of the origin facts for any foreign position in t or in any of its siblings. The *ancestor* relation on facts is the transitive closure of the parent relation. This definition of siblings and parents implies that facts introducing no fresh nulls (since we are assuming the above normal form, these are the facts created by a full TGD) can be neither parents nor siblings.

Recall that we identify facts by their ids rather than by their concrete values. Hence, any substitutions of nulls that happen in the course of the chase do not change the set of siblings, the set of parents, or the set of ancestors of a fact.

Example 4. Let us revisit the two TGDs $S(id_S, x, y) \rightarrow P(id_P, y, z)$ and $P(id_P, y, z) \rightarrow Q(id_Q, y, v)$ from Example 2, see also Figure 2. Although the creation of the atom $Q(y_1, v_1)$ was triggered by the atom $P(y_1, z_1)$, the only parent of $Q(y_1, v_1)$ is the origin fact of y_1, namely $S(x_1, y_1)$.

Some useful notation. To reason about the effects of EGDs, it is convenient to introduce some additional notation, following [3]. Let J be a canonical preuniversal instance and J' the canonical universal solution, resulting from chasing J with a set of target dependencies Σ_t. Moreover, suppose that u is a term which either exists in the domain of J or which is introduced in the course of the chase. Then we write $[u]$ to denote the term to which u is mapped by the chase. More precisely, let $t = S(u_1, u_2, \ldots, u_s)$ be an arbitrary fact, which either exists in J or which is introduced by the chase. Then the same fact t in J' has the form

$S([u_1], [u_2], \ldots, [u_s])$. By Lemma 2, every $[u_i]$ is well-defined, since it corresponds to the term produced by the chase in the corresponding origin position. For any set Σ_t of TDs, constants are mapped onto themselves: $\forall c \in \text{const}(J)\ c = [c]$. For $u, v \in \text{dom}(J)$, we write $u \sim v$ if $[u] = [v]$, i.e. two terms have the same image in J'. If Σ_t contains no EGDs, then $u = [u]$ holds for all $u \in \text{dom}(J)$. Clearly, the mapping $[\cdot]\colon J \to J'$ is a homomorphism.

The following lemma is the basis for constructing a homomorphism $h'\colon T^{\Sigma_{st}} \to U$, analogously to Theorem 5 by extending a homomorphism $h\colon T_{xy} \to U$.

Lemma 3. *For every weakly acyclic set Σ_t of TGDs and EGDs, instance T, and $x, y \in dom(T^{\Sigma_t})$, there exist constants b, c which depend only on $\Sigma = \Sigma_{st} \cup \Sigma_t$ and an instance T_{xy} satisfying* (1) *$Origin_x, Origin_y \subseteq T_{xy}$,* (2) *all facts of T are in T_{xy}, and $T_{xy} \subseteq T^{\Sigma_t}$,* (3) *$T_{xy}$ is closed under parents and siblings over facts, and* (4) *$|\text{dom}(T_{xy})| \leq |\text{dom}(T)| + b$. Moreover, T_{xy} can be computed in time $O(|\text{dom}(T)|^c)$.*

Compared with Lemma 1, we had to redefine the set T_{xy}. Moreover, the unification of variables caused by EGDs in the chase invalidates some essential assumptions in the proof of the corresponding result in [6, Theorem 7]. At any rate, also in our case, the lifting can be performed efficiently:

Theorem 6. *(*Lifting*) Let T^{Σ_t} be a universal solution of a data exchange problem obtained by chasing a preuniversal instance T with the weakly acyclic set Σ_t of TGDs and EGDs. If B and W are instances such that:* (1) *$B \models \Sigma$ with $\Sigma = \Sigma_{st} \cup \Sigma_t$;* (2) *all facts of T are in W (i.e. W contains facts with the same* ids*) and $W \subseteq T^{\Sigma_t}$, and* (3) *W is closed under ancestors and siblings (over facts), then any homomorphism $h\colon W \to B$ can be transformed in time $O(|\text{dom}(T)|^b)$ into a homomorphism $h'\colon T^{\Sigma_t} \to B$, s.t. $\forall x \in dom(h)\colon h(x) = h'(x)$, where b depends only on Σ.*

Proof. Although every fact of T is in W, there may of course be variables in $\text{dom}(T)$ which are not in $\text{dom}(W)$, because of the EGDs. Hence, $\forall x \in \text{dom}(T) \setminus \text{dom}(W)\colon x \neq [x]$, and $\forall x \in \text{dom}(T) \cap \text{dom}(W)\colon x = [x]$.

Suppose that the chase of a preuniversal instance T with Σ_t has length n. Then we write T_s with $0 \leq s \leq n$ to denote the result after step s of the chase. In particular, we have $T_0 = T$ and $T_n = T^{\Sigma_t}$. For every s, we say that a homomorphism $h_s\colon T_s \to B$ is *consistent with h* if $\forall x \in \text{dom}(h_s)$, such that $[x] \in \text{dom}(h)$, $h_s(x) = h([x])$ holds. We claim that for every $s \in \{0, \ldots, n\}$, such a homomorphism h_s consistent with h exists. Then $h' = h_n$ is the desired homomorphism. This claim can be proved by induction on s.

In order to actually construct the homomorphism $h' = h_n$, we may thus simply replay the chase and construct h_s for every $s \in \{0, \ldots, n\}$. The length n of the chase is polynomially bounded (cf. Section 2.1). The action required to construct h_s from h_{s-1} fits into polynomial time as well. We thus get the desired upper bound on the time needed for the construction of h'. □

The only ingredient missing for our FindCore^E algorithm is an efficient search for a homomorphism $h\colon T_{xy} \to U$ with $U \subseteq T^{\Sigma_t}$.

Procedure FindCore$^{\mathbf{E}}$

Input: Source ground instance S
Output: Core of a universal solution for S

(1) Chase (S,∅) with Σ_{st} to obtain $(S,T) := (S,\emptyset)^{\Sigma_{st}}$;
(2) Chase T with Σ_t to obtain $U := T^{\Sigma_t}$;
(3) **for** each $x \in \mathrm{var}(U)$, $y \in \mathrm{dom}(U)$, $x \neq y$ **do**
(4) | Compute T_{xy};
(5) | Look for $h\colon T_{xy} \to U$ s.t. $h(x) = h(y)$;
(6) | **if** there is such h **then**
(7) | | Extend h to an endomorphism h' on U;
(8) | | Transform h' into a retraction r;
(9) | | Set $U := r(U)$;
(10) **return** U.

By the construction of T_{xy} according to Lemma 3, the domain size of T_{xy} as well as the number of facts in it are only by a constant larger than those of the corresponding preuniversal instance T. By Theorem 1, the complexity of searching for a homomorphism is determined by the block size. The problem with EGDs in the target chase is that they may destroy the block structure of T by equating variables from different blocks of T. However, we show below that the search for a homomorphism on T_{xy} may still use the blocks of $T^{\Sigma_{st}}$ computed *before* the target chase. To achieve this, we adapt the *Rigidity Lemma* from [3]. The original *Rigidity Lemma* was formulated for sets of target dependencies consisting of EGDs only. A close inspection of the proof in [3] reveals that it remains valid when TGDs are added.

Definition 1. *Let K be an instance whose elements are constants and nulls. Let y be some element of K. We say that y is* rigid *if $h(y) = y$ for every endomorphism h on K. In particular, all constants of K are rigid.*

Lemma 4. *(*Rigidity*) Assume a data exchange setting where Σ_{st} is a set of TGDs and Σ_t is a set of EGDs* and TGDs. *Let J be the canonical preuniversal instance and let $J' = J^{\Sigma_t}$ be the canonical universal instance. Let x and y be nulls of J s.t. $x \backsim y$ (i.e., $[x] = [y]$) and s.t. $[x]$ is a nonrigid null of J'. Then x and y are in the same block of J.*

Next, we formalize the idea of considering the blocks of the preuniversal instance when searching for a homomorphism on the universal instance.

Definition 2. *We define the* non-rigid Gaifman graph $\mathcal{G}'(I)$ *of an instance I as the usual Gaifman graph but restricted to vertices corresponding to non-rigid variables. We define* non-rigid blocks *of an instance I as the connected components of the non-rigid Gaifman graph $\mathcal{G}'(I)$.*

Theorem 7. *Let T be a preuniversal instance obtained via the STDs Σ_{st}. Let Σ_t be a set of weakly acyclic TGDs and EGDs, and let U be a retract of T^{Σ_t}. Moreover, let $x, y \in \mathrm{dom}(T^{\Sigma_t})$ and let $T_{xy} \subseteq T^{\Sigma_t}$ be constructed according to Lemma 3. Then we can check if there exists a homomorphism $h\colon T_{xy} \to U$, s.t. $h(x) = h(y)$ in time $O(|dom(U)|^c)$ for some c depending only on $\Sigma = \Sigma_{st} \cup \Sigma_t$.*

Proof. First, it can be easily shown that the rigid variables of T^{Σ_t} are also rigid in T_{xy}. The key observation to achieve the $O(|dom(U)|^c)$ upper bound on the complexity is that the search for a homomorphism $h\colon T_{xy} \to U$ proceeds by inspecting the non-rigid blocks of T_{xy} individually. Moreover, since we already

have a retraction $r : T^{\Sigma} \to U$, we may search for a homomorphism h with $h(x) = h(y)$ by inspecting only the blocks containing x and y and to set $h(z) = r(z)$ for the variables of all other blocks. □

Putting all these pieces together, we get the FindCore^E algorithm. It has basically the same overall structure as the FindCore algorithm of [6], which we recalled in Section 2.2. Of course, the correctness of our algorithm and its polynomial time upper bound are now based on the new results proved in this section. In particular, step (4) is based on Lemma 3, step (5) is based on Lemma 4 and Theorem 7, and step (7) is based on Theorem 6.

Theorem 8. *Let $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ be a data exchange setting with STDs Σ_{st} and TDs Σ_t. Moreover, let S be a ground instance of the target schema $\mathbf{S}$. If this data exchange problem has a solution, then* FindCore^E *correctly computes the core of a canonical universal solution in time $O(|\mathrm{dom}(S)|^b)$ for some b that depends only on $\Sigma_{st} \cup \Sigma_t$.*

4 Implementation and Experimental Results

Implementation. We have implemented the FindCore^E algorithm presented in Section 3 in a prototype system. Its principal architecture is shown in Figure 3(a). For specifying data exchange scenarios, we use XML configuration files. The schema of the source and target database as well as the STDs and TDs are thus cleanly separated from the scenario-independent Java code. The XML configuration data is passed to the Java program, which uses XSLT templates to automatically generate those code parts which depend on the concrete scenario – in particular, the SQL-statements for managing the target database (creating tables and views, transferring data between tables etc.).

None of the common DBMSs to-date support labeled nulls. Therefore, to implement this feature, we had to augment every target relation (i.e., table) with additional columns, storing null labels. For instance, for a column `tutor` of the `Tutor` table, a column `tutor_var` is created to store the labels for nulls of `tutor`. To simulate homomorphisms, we use a table called `Map` storing variable mappings, and views that substitute labeled nulls in the data tables with their

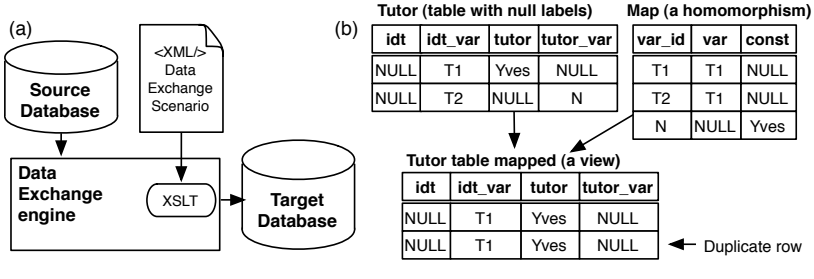

Fig. 3. Overview of the implementation (a) and modelling labeled nulls (b)

images given by a homomorphism. Figure 3(b) gives a flavor of what this part of the database looks like. The target database contains many more auxiliary tables for maintaining the relevant information of the core computation.

A great deal of the core computation is delegated to the target DBMS via SQL commands. For instance, the homomorphism computation in step 5 of FindCore^E is performed in the following way. Let a variable x and a term y be selected at step 3 of the algorithm, and let the set T_{xy} be computed at step 4. We want to build a homomorphism $h\colon T_{xy} \to U$, s.t. $h(x) = h(y)$. To do so, we need to inspect all possible mappings from the block of x and from the block of y. Each of these steps boils down to generating and executing a database query that fetches all possible substitutions for the variables in each block.

Example 5. Let us revisit the data exchange setting from Example 1. Suppose that the canonical universal solution is

$J = \{$`Course(`C_1`,'java'), Tutor(`T_2`,`N`), Teaches(`T_2`,`C_1`), NeedsLab(`T_2`,`L_2`),`
`Course(`C_2`,'java'), Tutor(`T_1`,'Yves'), Teaches(`T_1`,`C_2`), NeedsLab(`T_1`,`L_1`)}`

Suppose that we look for a proper endomorphism h' on J and suppose that step 4 of FindCore^E yields the set $T_{N,'\mathsf{Yves}'} = \{$`Tutor(`$T_2$`,`$N$`),Teaches(`$T_2$`,`$C_1$`), Course(`$C_1$`,'java')`$\}$. At step 5, a homomorphism $h\colon T_{xy} \to J$ (with $x = N$ and $y =' \mathsf{Yves}'$), s.t. $h(N) = '\mathsf{Yves}'$ has to be found. In the absence of EGDs, non-rigid blocks are the same as usual blocks, and the block of N in $T_{N,'\mathsf{Yves}'}$ is $\{N, T_2, C_1\}$. The following SQL query returns all possible instantiations of the variables $\{T_2, C_1\}$ compatible with the mapping $h(N) = '\mathsf{Yves}'$:

SELECT Tutor.idt_var **AS** T2, Course.idc_var **AS** C1 **FROM** Tutor **JOIN** Teaches **ON** Tutor.idt_var = Teaches.id_tutor_var **JOIN** Course **ON** Teaches.id_course_var = Course.idc_var **WHERE** Tutor.tutor='Yves' **AND** Course.course='java'

In our example, the result is $\{T_2 \leftarrow T_1,\ C_1 \leftarrow C_2\}$.

Experiments. We have run experiments with our prototype implementation on several scenarios with varying size of the schema (5–10 target relations), of the dependencies (5–15 constraints), and of the actual source data. Since there

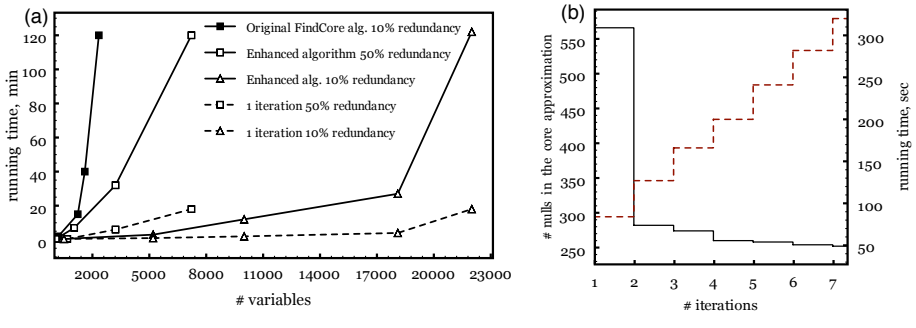

Fig. 4. Performance (a) and the progress (b) of core computation

are no established benchmarks for core computation algorithms, we constructed our own test cases by appropriately extending the data exchange scenario from Example 1. Typical runtimes are displayed in Figure 4. They were obtained by tests on a workstation running Suse Linux with 2 QuadCore processors (2.3 GHz) and 16 GB RAM. Oracle 11g was used as database system.

We had to synthesize the scenarios ourselves since no benchmark for core computation exists currently, and we needed means to adjust the dependencies and the source data in order to manage redundancy in the target database. When the target EGDs were deliberately "badly designed", the canonical solution had about 50% more nulls than the core. In this case, our system handled only about 7,000 nulls in the target DB in 120 min (2nd solid curve from the left). In contrast, when the target EGDs were "carefully designed", the canonical solution had only 10% more nulls than the core. In this case, about 22,000 nulls were handled in similar time (3rd solid curve).

We have also implemented the FindCore algorithm of [6] in order to compare its performance with our algorithm. The left-most curve in Figure 4(a) corresponds to a run of FindCore on the "well-designed" data exchange problem. The runtime is comparable to FindCore^E in case of "badly designed" dependencies. Actually, this is not surprising: One of the principal advantages of the FindCore^E algorithm is that it enforces EGDs as part of the chase rather than in the course of the core computation. The negative effect of simulating the EGDs by TGDs is illustrated by the following simple example:

Example 6. Let $J = \{R(x,y), P(y,x)\}$ be a preuniversal instance, and let a single EGD $R(z,v), P(v,z) \rightarrow z = v$ constitute Σ_t. To simulate this EGD by TGDs in [6], the following set of dependencies $\bar{\Sigma}_t$ has to be constructed:

$$
\begin{array}{llll}
P(x,y) \rightarrow E(x,x) & E(x,y) \rightarrow E(y,x) & & \\
P(x,y) \rightarrow E(y,y) & E(x,y), E(y,z) \rightarrow E(x,z) & P(x,y), E(x,z) \rightarrow P(z,y) & \\
R(x,y) \rightarrow E(x,x) & R(x,y), E(x,z) \rightarrow R(z,y) & P(x,y), E(y,z) \rightarrow P(x,z) & \\
R(x,y) \rightarrow E(y,y) & R(x,y), E(y,z) \rightarrow R(x,z) & R(z,v), P(v,z) \rightarrow E(z,v) &
\end{array}
$$

where E is the auxiliary predicate representing equality. Chasing J with $\bar{\Sigma}_t$ (in a nice order), yields the instance $J^{\bar{\Sigma}_t} = \{R(x,y), R(x,x), R(y,x), R(y,y), P(y,x), P(y,y), P(x,y), P(x,x), E(x,x), E(x,y), E(y,x), E(y,y)\}$. The core computation applied to $J^{\bar{\Sigma}_t}$ produces the instance $\{R(x,x), P(x,x)\}$ or $\{R(y,y), P(y,y)\}$. On the other hand, if EGDs were directly enforced by the target chase, then the chase would end with the canonical universal solution $J^{\Sigma_t} = \{R(x,x), P(x,x)\}$.

Another interesting observation is that, in many cases, the result of applying just a few endomorphisms already leads to a significant elimination of *redundant* nulls (i.e., nulls present in the canonical solution but not in the core) from the target database and that further iterations of this procedure are much less effective concerning the number of nulls eliminated vs. time required. A typical situation is shown in Figure 4(b): The solid line shows the number of redundant nulls remaining after i iterations (i.e., i nested endomorphisms) while the dotted line shows the total time required for the first i iterations. To achieve this, we used several heuristics to choose the best homomorphisms. The following hints

proved quite useful: (i) prefer constants over variables, (ii) prefer terms already used as substitutions, and (iii) avoid mapping a variable onto itself.

Every intermediate database instance produced by FindCore^E is a universal solution to the data exchange problem. Hence, our prototype implementation also allows the user to restrict the number of nested endomorphisms to be constructed, thus computing an approximation of the core rather than the core itself. The dotted curves in Figure 4(a) corresponds to a "partial" core computation, with only 1 iteration of the while-loop in FindCore^E. In both scenarios, even a single endomorphism allowed us to eliminate over 85% of all redundant nulls.

Lessons learned. Our experiments have clearly revealed the importance of carefully designing target EGDs. In some sense, they play a similar role as the core computation in that they lead to an elimination of nulls. However, the EGDs do it much more efficiently. Another observation is that it is well worth considering to content oneself with an approximation of the core since, in general, a small number of iterations of our algorithm already leads to a significant reduction of nulls. Finally, the experience gained with our experiments gives us several hints for future performance improvements. We just give three examples: `(i)` Above all, further heuristics have to be incorporated concerning the search for an endomorphism which maps a labeled null onto some other domain element. So far, we have identified and implemented only the most straightforward, yet quite effective, rules. Apparently, additional measures are needed to further prune the search space. `(ii)` We have already mentioned the potential of approximating the core by a small number of endomorphisms. Again, we need further heuristics concerning the search for the most effective endomorphisms. `(iii)` Some phases of the search for an endomorphism allow for concurrent implementation. This potential of parallelization, which has not been exploited so far, clearly has to be leveraged in future versions of our implementation.

5 Conclusion

In this paper we have revisited the core computation in data exchange and we have come up with an enhanced version of the FindCore algorithm from [6], which avoids the simulation of EGDs by TGDs. The algorithms FindCore and FindCore^E look similar in structure and have essentially the same asymptotic worst-case behavior. More precisely, both algorithms are exponential w.r.t. some constant b which depends on the dependencies $\Sigma_{st} \cup \Sigma_t$ of the data exchange setting. Actually, in [9] it was shown that the core computation for a given target instance J is fixed-parameter intractable w.r.t. its block size. Hence, a significant reduction of the worst-case complexity is not likely to be achievable. At any rate, as we have discussed in Section 4, our new approach clearly outperforms the previous one under realistic assumptions.

We have also presented a prototype implementation of our algorithm, which delegates most of its work to the underlying DBMS via SQL. It has thus been demonstrated that core computation fits well into existing database technology and is clearly not a separate technology. Although the data exchange scenarios

tackled so far are not industrial size examples, we expect that there is ample space for performance improvements. The experience gained with our prototype gives valuable hints for directions of future work.

References

1. Fagin, R.: Horn clauses and database dependencies. J. ACM 29, 952–985 (1982)
2. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theor. Comput. Sci. 336, 89–124 (2005)
3. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. 30, 174–210 (2005)
4. Beeri, C., Vardi, M.Y.: A proof procedure for data dependencies. J. ACM 31, 718–741 (1984)
5. Hell, P., Nešetřil, J.: The core of a graph. Discrete Mathematics 109, 117–126 (1992)
6. Gottlob, G., Nash, A.: Data exchange: computing cores in polynomial time. In: Proc. PODS 2006, pp. 40–49. ACM Press, New York (2006)
7. Pichler, R., Savenkov, V.: Towards practical feasibility of core computation in data exchange. Technical Report DBAI-TR-2008-57, TU Vienna (2008), `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2008-57.pdf`
8. Deutsch, A., Tannen, V.: Reformulation of XML queries and constraints. In: Calvanese, D., Lenzerini, M., Motwani, R. (eds.) ICDT 2003. LNCS, vol. 2572, pp. 225–238. Springer, Heidelberg (2002)
9. Gottlob, G.: Computing cores for data exchange: new algorithms and practical solutions. In: Proc. PODS 2005, pp. 148–159. ACM Press, New York (2005)