DBAI
TECHNICAL REPORT

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

Towards Practical Feasibility of Core Computation in Data Exchange

DBAI-TR-2008-57

Reinhard Pichler, Vadim Savenkov

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18492
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI TECHNICAL REPORT
2008

TECHNISCHE UNIVERSITÄT WIEN

DBAI TECHNICAL REPORT

DBAI TECHNICAL REPORT DBAI-TR-2008-57, 2008

Towards Practical Feasibility of Core Computation in Data Exchange

Reinhard Pichler, Vadim Savenkov,[1]

Abstract. Data exchange is concerned with the transfer of data between databases with different schemas, whereby certain integrity constraints have to be observed. Given a source database, a target database fulfilling all the integrity constraints is called a "solution" to the data exchange problem.
In general, a source database admits a big number of solutions, which may significantly differ in size. The most compact one among the most general (universal) solutions is called the core. Fagin et al. gave convincing arguments that, in many cases, the core is the preferred solution to a data exchange problem. Moreover, Gottlob and Nash showed 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 aim of this paper is to address the principal obstacles to the practical feasibility of core computation and to make a big step forward towards the integration of core computation into data exchange systems.

[1]Technische Universität Wien mailto:{pichler|savenkov}@dbai.tuwien.ac.at

1. INTRODUCTION

Data exchange is concerned with the transfer of data between databases with different schemas. While data integration usually deals with query translation and query processing among multiple databases [10, 6], data exchange aims at actually materializing a target database stemming from some source database [4]. In order to make sure that the source data is accurately reflected by the target data, the materialization of the data in the target schema is governed by a set of source-to-target dependencies (STDs). Moreover, the target database may also impose additional integrity constraints, called target dependencies (TDs). Following [4, 5], we confine ourselves to relational schemas and to dependencies which may either be *tuple generating dependencies* (TGDs) or *equality generating dependencies* (EGD) [1].

The source schema **S** and the target schema **T** together with the set Σ_{st} of STDs and the set Σ_t of TDs constitute the *data exchange setting*. 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* to the data exchange problem. Typically, the number of possible solutions to a data exchange problem is infinite.

Example 1.1. *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. BasicUnit$(C) \rightarrow$ Course(Idc, C).
2. Tutorial$(C, T) \rightarrow$ Course(Idc, C), Tutor(Idt, T), Teaches(Idt, Itc).

and the TDs are given by the two TGDs:

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

Then the following instances are all valid solutions:

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

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

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

A natural requirement (proposed in [4]) 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 1.1 is not universal, since there exists no homomorphism $h\colon J' \rightarrow J$. Indeed, a homomorphism maps any constant onto itself and, therefore, the fact Course('java','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* [5], which is the most compact one. For instance, solution J_c in Example 1.1 is a core.

Fagin et al. [5] gave convincing arguments that, in many cases, the core should be the database to be materialized. Moreover, Gottlob and Nash [8] showed 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 like, e.g., Clio [9]. This is mainly due to the following counter-arguments which have been put forward against core computation: (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 of the algorithm in [8] exists. (2) The core computation looks like a separate technology which cannot be easily integrated into existing database technology.

The principal aim of this paper is to make a big step forward towards the integration of core computation into data exchange systems. The starting point of our work is the FindCore algorithm developed by Gottlob and Nash [8]. 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. As we shall point out in Section 5, the simulation of EGDs by TGDs, in general, causes a significant loss of performance. Moreover, there are other data exchange semantics [11] that favor the materialization of *canonical* universal solutions (for a definition, see Section 2) rather than cores. Hence, the core computation should be treated as an optional service and strictly separated from the process of finding a solution.

Results. The main contribution of this work is twofold:

(1) We present an enhanced version of the FindCore algorithm. The most significant advantage of our algorithm (which we shall refer to as FindCore^E) 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 the 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 gives the implementation a lot of flexibility and avoids rebuilding functionality which is provided by any RDBMS anyway. Moreover, 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 of core computation.

Structure of the paper. This paper is organized as follows. In Section 2, we recall some basic notions as well as the FindCore algorithm. The FindCore^E algorithm is presented in Section 3. In Section 4, we outline a prototype implementation. First experimental results are presented and discussed in Section 5. We conclude with Section 6.

2. PRELIMINARIES

2.1 Basic concepts of data exchange

Data exchange problem. A *schema* $\sigma = \{R_1, \ldots, R_n\}$ is a set of relation symbols R_i each of a fixed arity. An *instance* over a schema σ consists of a relation for each relation symbol in σ, s.t. both have the same arity. We only consider finite instances. By slight abuse of notation, we sometimes identify a relation with its relation symbol (and vice versa).

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 also write $\vec{x}$ for a tuple $(x_1, x_2, \ldots, x_n)$ and if $x_i \in X$, for every i, then we also write $\vec{x} \in X$ instead of $\vec{x} \in X^n$. Likewise, we write $r \in \vec{x}$ if $r = x_i$ for some i.

Let $\mathbf{S} = \{S_1, \ldots, S_n\}$ and $\mathbf{T} = \{T_1, \ldots, T_m\}$ be schemas with no relation symbols in common. We call $\mathbf{S}$ the *source schema* and $\mathbf{T}$ the *target schema*. We write $\langle \mathbf{S}, \mathbf{T} \rangle$ to denote the schema $\{S_1, \ldots, S_n, T_1, \ldots, T_m\}$. Instances over $\mathbf{S}$ (resp. $\mathbf{T}$) are called *source instances* (resp. *target instances*). If I is a source instance and J a target instance, then their combination $\langle I, J \rangle$ is an instance of the schema $\langle \mathbf{S}, \mathbf{T} \rangle$.

Embedded dependencies [3] over a schema σ are first-order formulae of the form

$$\forall \vec{x}\, (\phi(\vec{x}\,) \rightarrow \exists \vec{y}\ \psi(\vec{x}, \vec{y}\,))$$

where *premise* ϕ and *conclusion* ψ are conjunctions of atomic formulas with relational symbols from σ or equalities, s.t. all variables in $\vec{x}$ actually do occur in $\phi(\vec{x}\,)$. Throughout this paper, we shall omit the universal quantifiers. By convention, all variables occurring in the premise are universally quantified (over the entire formula). Moreover, we shall often also omit the existential quantifiers, unless we want to emphasize them. By convention, all variables occurring in the conclusion only are existentially quantified over the conclusion. We shall thus use the notations $\phi(\vec{x}\,) \rightarrow \psi(\vec{x}, \vec{y}\,)$ and $\phi(\vec{x}\,) \rightarrow \exists \vec{y}\ \psi(\vec{x}, \vec{y}\,)$ interchangeably for the above formula.

Let Σ be a set of dependencies and I an instance. We write $I \models \Sigma$ to denote that the instance I satisfies Σ.

In the context of data exchange, we are mainly dealing with *source-to-target dependencies* (STDs) and *target dependencies* (TDs). In STDs, the premise may only use relation symbols from the source schema while the conclusion may only use relation symbols from the target schema. In TDs, both the premise and the conclusion may only use relation symbols from the target schema. Note that source dependencies may be important for deriving STDs (see [12]). However, they play no direct role in data exchange, where we take the source instance to be given.

A *data exchange setting* is given by a quadruple $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ consisting of the source schema $\mathbf{S}$, the target schema $\mathbf{T}$, the set of STDs Σ_{st} and the set of TDs Σ_t. The *data exchange problem* associated with this setting is the following: Given a (ground) source instance I, find a target instance J, s.t. $\langle I, J \rangle \models \Sigma_{st}$ and $J \models \Sigma_t$. Such a J is called a *solution for* I or, simply, a *solution* if I is clear from the context.

TGDs and EGDs. Following [4, 5], we consider dependencies in Σ_{st} and Σ_t of the following forms: Each STD in Σ_{st} is a *tuple generating dependency* (TGD) of the form

$$\phi_{\mathbf{S}}(\vec{x}\,) \rightarrow \psi_{\mathbf{T}}(\vec{x}, \vec{y}\,)$$

where $\phi_{\mathbf{S}}(\vec{x}\,)$ is a conjunction of atomic formulas over $\mathbf{S}$ and $\psi_{\mathbf{T}}(\vec{x}, \vec{y}\,)$ is a conjunction of atomic formulas over $\mathbf{T}$. Each TD in Σ_t is either a TGD, of the form

$$\phi_{\mathbf{T}}(\vec{x}\,) \rightarrow \psi_{\mathbf{T}}(\vec{x}, \vec{y}\,)$$

or an *equality generating dependency* (EGD) of the form

$$\phi_{\mathbf{T}}(\vec{x}\,) \rightarrow (x_i = x_j).$$

In these dependencies, $\phi_{\mathbf{T}}(\vec{x}\,)$ and $\psi_{\mathbf{T}}(\vec{x}, \vec{y}\,)$ are conjunctions of atomic formulas over $\mathbf{T}$, and x_i, x_j are among the variables in $\vec{x}$. The special case of a TGD without (existentially quantified) variables $\vec{y}$ is called a *full TGD*, i.e. we have $\phi_{\mathbf{S}}(\vec{x}\,) \rightarrow \psi_{\mathbf{T}}(\vec{x}\,)$ and $\phi_{\mathbf{T}}(\vec{x}\,) \rightarrow \psi_{\mathbf{T}}(\vec{x}\,)$, respectively.

Chase. The data exchange problem can be solved by the *chase* [1], a sequence of steps, each enforcing a single constraint within some limited set of tuples. More precisely, let Σ contain a TGD $\tau: \phi(\vec{x}\,) \rightarrow \psi(\vec{x}, \vec{y}\,)$, s.t. $I \models \phi(\vec{a}\,)$ for some assignment $\vec{a}$ on $\vec{x}$ and $I \not\models \exists \vec{y} \psi(\vec{a}, \vec{y}\,)$. Then we have to extend I with facts corresponding to $\psi(\vec{a}, \vec{z}\,)$, where the elements of $\vec{z}$ are fresh labeled nulls. Likewise, suppose that Σ contains an EGD $\tau: \phi(\vec{x}\,) \rightarrow x_i = x_j$, s.t. $I \models \phi(\vec{a}\,)$ for some assignment $\vec{a}$ on $\vec{x}$. 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 to guarantee the termination of the chase is that the set of TGDs be *weakly acyclic* (see [2, 4]). 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 $\phi(\vec{x}\,) \rightarrow \psi(\vec{x}, \vec{y}\,)$ be a TGD in Σ and suppose that some variable $x \in \vec{x}$ 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 $\psi(\vec{x}, \vec{y}\,)$ or (2) x occurs in some other field T^k in $\psi(\vec{x}, \vec{y}\,)$ and there is a variable $y \in \vec{y}$ in the field S^j in $\psi(\vec{x}, \vec{y}\,)$. 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, but not vice versa. In summary, we only 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 below shows the dependency graph for the target TGDs in Example 1.1, where special edges are marked with *. Clearly, this graph has no cycle containing a special edge (actually, it contains no cycle at all). Hence, these TGDs are weakly acyclic.

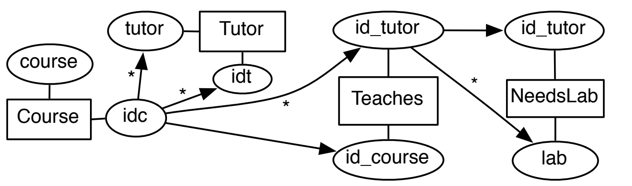


Figure 1: Dependency graph.

Universal solutions and core. Let I,I' be instances. A *homomorphism* $h: I \rightarrow I'$ is a mapping $\mathrm{dom}(I) \rightarrow \mathrm{dom}(I')$, s.t. (1) whenever $R(\vec{x}\,) \in I$, then $R(h(\vec{x}\,)) \in I'$, and (2) for every constant c, $h(c) = c$. An *endomorphism* is a homomorphism $I \rightarrow I$, and a *retraction* 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), i.e., if it "shrinks" the domain, so to speak. 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 an arbitrary 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 [4].

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 $\tau\colon \phi(\vec{x}\,) \to \psi(\vec{x}, \vec{y}\,)$ in Σ, we define the *width* of τ to be $|\vec{x}|$, and the *height* as $|\vec{y}|$. The width (resp. the height) of Σ is the maximal width (resp. height) of the dependencies in Σ.

Our main topic here is the core computation, which is essentially a search for appropriate homomorphisms. It was shown in [5], that the key complexity factor when searching for homomorphisms is the *block size*, which is defined as follows: The *Gaifman graph* $\mathcal{G}(I)$ of an instance I is an undirected graph whose vertices are 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. In [5], the following important results concerning the block size were proved:

THEOREM 2.1. [5] *Let A and B be instances, and suppose that* $\text{blocksize}(A) \le c$ *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 $O(|A| \cdot |B|^c)$.*

PROOF. (SKETCH) The crucial observation is that, in order to search for a homomorphism $h\colon A \to B$, we may search for homomorphisms from every block of A onto B separately. Note that A has $\le |A|$ blocks, each containing $\le c$ variables. Hence, from each block of A we have to consider $\le |B|^c$ possible mappings onto B. $\square$

THEOREM 2.2. [5] *If Σ_{st} is a set of STDs of height e, S is ground, and $(S, T) = (S, \emptyset)^{\Sigma_{st}}$, then* $\text{blocksize}(T) \le e$.

Sibling, parent, ancestor. Consider the chase of the preuniversal instance T with TDs Σ_t and suppose that $\vec{y}$ is a tuple of variables created by enforcing a TGD $\phi(\vec{x}\,) \to \psi(\vec{x}, \vec{y}\,)$ in Σ_t, s.t. the precondition $\phi(\vec{x}\,)$ was satisfied with a tuple $\vec{a}$. Then the elements of $\vec{y}$ are *siblings* of each other; every variable of $\vec{a}$ is a *parent* of every element of $\vec{y}$; 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 [8]. 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 $E(x, y)$ thus turning every EGD into a TGD.
2. Add the following *equality* constraints:
   - $E(x, y) \to E(y, x)$
   - $E(x, y), E(y, z) \to E(x, z)$
   - $R(x_1, \ldots, x_k) \to E(x_i, x_i)$ for every $R \in \tau$ and $i \in \{1, 2, \ldots, k\}$ where k is the arity of R
3. Add the following consistency constraints:
   - $R(x_1, \ldots, x_k), E(x_i, y) \to R(x_1, \ldots, y, \ldots, x_k)$ 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 $\Sigma_{st}$ to obtain $(S,T) := (S,\emptyset)^{\Sigma_{st}}$;
(2)  Compute $\bar{\Sigma}_t$ from $\Sigma_t$;
(3)  Chase T with $\bar{\Sigma}_t$ (using a nice order) to get $U := T^{\bar{\Sigma}_t}$;
(4)  for each $x \in \text{var}(U)$, $y \in \text{dom}(U)$, $x \neq y$ do
(5)      Compute $T_{xy}$;
(6)      Look for $h\colon T_{xy} \to U$ 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 [8] 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 iteratively computing a succession of nested retracts. This is motivated by the fact that retractions have the following favorable properties: (1) embedded dependencies are closed under retractions and (2) any proper endomorphism can be effectively transformed into a retraction [8]:

THEOREM 2.3. [8] *Let $r\colon A \to A$ 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 2.4. [8] *Given an endomorphism $h\colon A \to A$ such that $h(x) = h(y)$ for some $x, y \in \text{dom}(A)$, there is a proper retraction r on A s.t. $r(x) = r(y)$. Such a retraction can be found in time $O(|\text{dom}(A)|^2)$.*

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 2.4 to turn h into a retraction r in step (9) and replace U by $r(U)$ in step (10). By Theorem 2.3 we know that Σ_{st} and $\bar{\Sigma}_t$ are still satisfied.

Searching for proper endomorphisms. At every step of the descent to a 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 2.1, 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 [8]:

LEMMA 2.1. [8] *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 2.5. (LIFTING) [8] *Let T^Σ be a universal solution of a data exchange problem obtained by chasing a pre-universal 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 auxiliary predicate E is used to simulate 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 and the core computation should halt with failure. 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 [8] 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. In total, we thus have

THEOREM 2.6. *[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 *correctly computes the core of a canonical universal solution in time $O(|\mathrm{dom}(S)|^b)$ for some b that depends only on $\Sigma_{st} \cup \Sigma_t$.*

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.:

- compute an instance T_{xy}
- search for a non-injective homomorphism $h \colon T_{xy} \to U$
- lift h to a proper endomorphism $h' \colon U \to U$
- construct a proper retraction r from h'.

Actually, the construction of a retraction r via Theorem 2.4 and the closure of embedded dependencies w.r.t. retractions according to Theorem 2.3 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, e.g.:

- 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 will need to define T_{xy} differently.
- 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 pre-universal instance T. EGDs have a positive effect in that they eliminate variables, thus reducing the size of a single block. Conversely, EGDs may also have a negative effect in that they may merge different blocks of the preuniversal instance T. Hence, without further measures, this would destroy the tractability of the search for a homomorphism $h \colon T_{xy} \to U$.
- Since we have to define T_{xy} differently from Section 2.2, also the lifting of $h \colon T_{xy} \to U$ to a proper endomorphism $h' \colon U \to U$ will have 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 and of the required modifications w.r.t. Section 2.2 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 fresh variable t.

These changes neither have an effect on the chase nor on the core computation (apart from increasing the variable domains of target instances), as no rules rely on values in the added columns. It is immediate that a fact $R(x_1, x_2, \ldots, x_n)$ s 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 null, 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 3.1. *Let $J = \{S(id_{S1}, x_1, y_1)\}$ be a preuniversal instance and consider the following target dependencies:*

1. $S(id_S, x, y) \rightarrow P(id_P, y, z)$
2. $P(id_P, y, z) \rightarrow Q(id_Q, y, v)$,

yielding the canonical universal solution J^Σ shown in Figure 2: $J^\Sigma = \{S(id_{S1}, x_1, y_1), P(id_{P1}, y_1, z_1), Q(id_{Q1}, y_1, v_1)\}$.

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. The remaining two positions of the facts id_{P1} and id_{Q1} are native.

The origin positions of the variables are as follows: $(id_{S1}, 1)$ for x_1, $(id_{S1}, 2)$ for y_1, $(id_{P1}, 2)$ for z_1, and $(id_{Q1}, 2)$ for v_1.

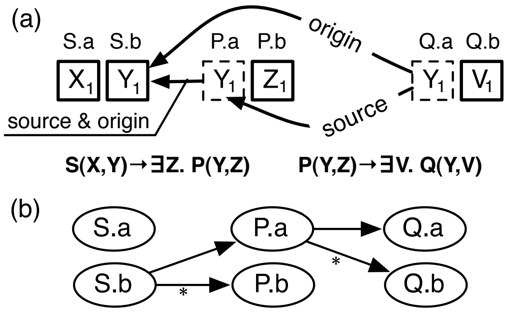


Figure 2: Positions of the instance J^Σ (foreign positions are dashed) (a) and the dependency graph of Σ (b).

LEMMA 3.1. *Let I be an instance. Moreover, let p a position in I and o_p its origin position. Then p and o_p always contain the same term.*

PROOF. If p is native to its fact, then $p = o_p$ by definition. Hence, in this case, p and o_p trivially hold the same term.

Otherwise, let $p \neq o_p$. Then there exists a chain $p_0, p_1, \ldots, p_n$ of positions, s.t. p_{i-1} is the source position of p_i for every $i \in \{1, \ldots, n\}$ and $p_0 = o_p$ and $p_n = p$. We proceed by induction on i: Of course, p_0 always contains the same term as o_p, since $p_0 = o_p$. Now suppose that, at any stage of the chase, p_{i-1} contains the same term as o_p. By definition, p_{i-1} is the source position of p_i, i.e.: When p_i is created by firing a TGD, then the term contained in p_{i-1} is *copied* to p_i. Hence, p_i will always contain the same term as p_{i-1}, no matter which EGDs are applied in the course of the chase. Thus, by the induction hypothesis, it will always contain the same term as o_p. □

Normalization of TGDs. Let $\tau\colon \phi(\vec{x}\,) \rightarrow \psi(\vec{x}, \vec{y}\,)$ be a non-full TGD, i.e., $\vec{y}$ is non-empty. Then we can set up the *Gaifman graph* $\mathcal{G}(\tau)$ of the atoms in the conclusion $\psi(\vec{x}, \vec{y})$, considering only the new variables $\vec{y}$, i.e., $\mathcal{G}(\tau)$ contains as vertices the variables in $\vec{y}$. Moreover, two variables y_i and y_j are adjacent (by slight abuse of notation, we identify vertices and variables), if they jointly occur in some atom of $\psi(\vec{x}, \vec{y})$. Let $\mathcal{G}(\tau)$ contain the connected components $\vec{y}_1, \ldots, \vec{y}_n$. Then the conclusion $\psi(\vec{x}, \vec{y}\,)$ is of the form

$$\psi(\vec{x}, \vec{y}\,) = \psi_0(\vec{x}\,) \wedge \psi_1(\vec{x}, \vec{y}_1) \wedge \cdots \wedge \psi_n(\vec{x}, \vec{y}_n),$$

where the subformula $\psi_0(\vec{x}\,)$ contains all atoms of $\psi(\vec{x}, \vec{y}\,)$ without variables from $\vec{y}$ and each subformula $\psi_i(\vec{x}, \vec{y}_i)$ contains exactly the atoms of $\psi(\vec{x}, \vec{y}\,)$ containing at least one variable from the connected component $\vec{y}_i$.

Now let the full TGD τ_0 be defined as $\tau_0\colon \phi(\vec{x}\,) \rightarrow \psi_0(\vec{x}\,)$ and let the non-full TGDs τ_i with $i \in \{1, \ldots, n\}$ be defined as $\tau_i\colon \phi(\vec{x}\,) \rightarrow \psi_i(\vec{x}, \vec{y}_i)$. 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 TGDs 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. Such target dependencies have the following property, which plays an important role in the proof of Theorem 3.1.

LEMMA 3.2. *Let the preuniversal instance J be chased with the set Σ_t of TDs in* normal form. *Suppose that at some step in the chase, the non-full TGD $\tau : \phi(\vec{x}\,) \rightarrow \psi(\vec{x}, \vec{y}\,)$ fires. Then τ introduces a new fact for every atom in the conclusion $\psi(\vec{x}, \vec{y}\,)$. More precisely, suppose that τ fires with the assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$. Then all atoms in $\psi(\vec{a}, \vec{z}\,)$ are newly created by this chase step.*

PROOF. Let J' denote the instance prior to this chase step. The TGD τ is only fired if it introduces at least one new fact. Let $\rho(\vec{a}, \vec{z}\,)$ denote the subformula of $\psi(\vec{a}, \vec{z}\,)$, s.t. all atoms in $\rho(\vec{a}, \vec{z}\,)$ are newly created by this chase step, while all atoms in the remaining subformula $\rho'(\vec{a}, \vec{z}\,)$ of $\psi(\vec{a}, \vec{z}\,)$ already exist in J'. We have to show that $\rho(\vec{a}, \vec{z}\,)$ comprises all atoms of $\psi(\vec{a}, \vec{z}\,)$.

Suppose to the contrary that $\rho(\vec{a}, \vec{z}\,)$ is a proper subformula of $\psi(\vec{a}, \vec{z}\,)$. Since this application of τ creates new facts for every atom in $\rho(\vec{a}, \vec{z}\,)$, the assignment $\vec{z}$ instantiates all variables in $\vec{y}$ occurring in $\rho(\vec{a}, \vec{z}\,)$ to fresh nulls. By the normalization of τ, the Gaifman graph $\mathcal{G}(\tau)$ has exactly 1 connected component. Hence, there exists at least one atom A in $\rho'(\vec{a}, \vec{y}\,)$, s.t. A shares with $\rho(\vec{a}, \vec{y}\,)$ a variable from $\vec{y}$. Hence, the atom $A[\vec{y} \leftarrow \vec{z}\,]$ in $\rho'(\vec{a}, \vec{z}\,)$ contains at least one fresh null. But this contradicts the assumption that $A[\vec{y} \leftarrow \vec{z}\,]$ already existed in J'. □

EXAMPLE 3.2. *Consider the non-full TGD*

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

Then τ 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 sense above.

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 $\tau : \phi(\vec{x}\,) \rightarrow \psi(\vec{x}, \vec{y}\,)$ is enforced, i.e.: $I \models \phi(\vec{a}\,)$ for some assignment $\vec{a}$ on $\vec{x}$ and $I \not\models \exists\vec{y}\psi(\vec{a}, \vec{y}\,)$, s.t. the facts corresponding to $\psi(\vec{a}, \vec{z}\,)$, where the elements of $\vec{z}$ are fresh labeled nulls, are added. Let t be a fact introduced by this chase step, i.e., t is an atom of $\psi(\vec{a}, \vec{z}\,)$. Then all other facts introduced by the same chase step (i.e., by Lemma 3.2, all other atoms of $\psi(\vec{a}, \vec{z}\,)$) are the *siblings* of t. Given a fact t, its *parent* set 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 facts.

Example 3.3. *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 3.1, 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 [5]. 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 3.1, 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: $\forall c \in \mathrm{const}(J)\ c = [c]$. For $u, v \in \mathrm{dom}(J)$, 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 $\forall u \in \mathrm{dom}(J)\ u = [u]$ holds. The following property of $[\cdot]$ is immediate:

Proposition 3.1. *The mapping $[\cdot]\colon J \rightarrow J'$ is a homomorphism.*

We are now ready to prove the main results underlying the FindCore^E algorithm, i.e.: Definition of T_{xy} (Lemma 3.3), search for a homomorphism $h\colon T_{xy} \rightarrow U$ (Lemma 3.4 and Theorem 3.3), and lifting a homomorphism $h\colon T_{xy} \rightarrow U$ to a non-injective homomorphism $T^{\Sigma_{st}} \rightarrow U$ (Theorem 3.1).

Lemma 3.3. *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,*
4. *$|\mathrm{dom}(T_{xy})| \leq |\mathrm{dom}(T)| + b$.*

Moreover, T_{xy} can be computed in time $O(|\mathrm{dom}(T)|^c)$.

Proof. Let d denote the depth of Σ_t. Given variable x, let the set F_x (= the "family" of x) denote the set of facts obtained as follows: At the deepest level j (with $j \leq d$), F_x contains $Origin_x$ and all siblings thereof. On the next higher level $j - 1$, F_x contains all parents of facts on level j plus all siblings thereof. This procedure is continued until the top level is reached. Thus, F_x contains $Origin_x$ and is closed under the parent and sibling relations. Then the set $T_{xy} := T^{\Sigma_{st}} \cup F_x \cup F_y$ satisfies the conditions 1–3.

The desired upper bound on the domain size of F_x and, therefore, of T_{xy} is obtained as follows: On every level, every fact has at most constantly many siblings with at most constantly many variables, where this constant only depends on Σ. Likewise, for the transition from one level to the next higher one, we observe that every fact has at most constantly many foreign positions, each with at most constantly many parents. Hence, since also the depth d of Σ_t is a constant of Σ, T_{xy} contains only constantly many facts in addition to the facts of T, and each new fact introduces only constantly many new variables. Note that EGDs cannot augment the domain size of any set of facts, since they result only in replacements of some variable u with some already present term v at all occurrences of u. Finally, the polynomial upper bound on the computation time needed to construct T_{xy} is clear, once we have the bound on the facts of T_{xy}. $\square$

Having a homomorphism $h\colon T_{xy} \rightarrow U$, we want to extend it to a homomorphism $h'\colon T^{\Sigma_{st}} \rightarrow U$, analogously to Theorem 2.5. However, compared with Lemma 2.1, 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 [8, Theorem 7]. At any rate, the following theorem shows that also in our case, the lifting can be performed efficiently by essentially the same procedure as described in [8].

Theorem 3.1. (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 $h\colon W \rightarrow B$ can be transformed in time $O(|\mathrm{dom}(T)|^b)$ into a homomorphism $h'\colon T^{\Sigma_t} \rightarrow B$, 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 $\mathrm{dom}(T)$ which are not in $\mathrm{dom}(W)$, because of the EGDs. Hence, $\forall x \in \mathrm{dom}(T) \setminus \mathrm{dom}(W)\colon x \neq [x]$, and $\forall x \in \mathrm{dom}(T) \cap \mathrm{dom}(W)\colon x = [x]$.

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 $h_s\colon T_s \rightarrow B$ is *consistent with* h if $\forall x \in \mathrm{dom}(h_s)$, such that $[x] \in \mathrm{dom}(h)$, $h_s(x) = h([x])$ 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 (see Appendix A).

In order to actually construct the homomorphism $h' = h_n$, we may thus simply replay the chase and construct h_s for every $s \in \{0, \dots, n\}$. 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'. $\square$

Even though the proof of THEOREM 3.1 directly yields an algorithm for transforming a homomorphism $h\colon W \to B$ to an appropriate homomorphism $h'\colon T^{\Sigma_t} \to B$ in polynomial time, it is slightly unsatisfactory. In fact, as intermediate steps, it may process variables which are not present any more in $\mathrm{dom}(T_s)$. Naturally, it would be desirable to skip such unnecessary steps. We therefore propose the following simplified procedure EXTEND, which allows us to literally *extend* h to $h'\colon T^{\Sigma_t} \to B$ starting with W and considering only the variables present in T^{Σ_t}. The correctness of the procedure EXTEND is the subject of the following theorem.

Procedure Extend

Input: Canonical universal solution T^{Σ_t}
Input: Subinstance $W \subseteq T^{\Sigma_t}$ closed under parents and siblings, s.t. W contains all facts of T
Input: Homomorphism $h\colon W \to B$ with $B \models \Sigma$
Output: Homomorphism $h'\colon T^{\Sigma_t} \to B$ such that $\forall x \in \mathrm{dom}(W)\ h'(x) = h(x)$

```
(1)  Set $h' := h$;
(2)  while exists a fact $A \in T^{\Sigma_t} \setminus W$, s.t. $Parents(A) \subseteq W$
(3)      Set $P := Parents(A)$
(4)      Set $S := \{A\} \cup Siblings(A)$
(5)      Find homomorphism $g\colon S \cup P \to B$,
           such that $\forall x \in \mathrm{dom}(g) \cap \mathrm{dom}(h')\colon g(x) = h'(x)$;
(6)      Set $h' := h' \cup g$;
(7)      Set $W := W \cup S$;
(8)  return $h'$.
```

THEOREM 3.2. *Let T, T^{Σ_t}, B, W, and $h\colon W \to B$ be as in Theorem 3.1. Then the procedure* EXTEND *extends h to a homomorphism $h'\colon T^{\Sigma} \to B$.*

PROOF. Let W_j with $j \geq 1$ denote the set W when the while-loop in the EXTEND procedure is entered for the j-th time. It can be shown by induction on j (see Appendix B) that W_j fulfills the following properties: $W_j \subseteq T^{\Sigma_t}$, W_j contains all facts from T, W_j is closed under parents and siblings, and $h_j\colon W_j \to B$ is a homomorphism s.t. $\forall x \in \mathrm{dom}(W)\colon h'(x) = h(x)$ holds.

Clearly, for every j, the transition from W_j to W_{j+1} corresponds to the application of a non-full TGD in the course of the target chase. Hence, the number of iterations of the while-loop is bounded by the length n of the chase. $\square$

The only ingredient missing for our FINDCORE^E algorithm is an efficient search for a homomorphism $h\colon T_{xy} \to U$ with $U \subseteq T^{\Sigma_t}$. By the construction of T_{xy} according to Lemma 3.3, 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 2.1, 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 [5].

DEFINITION 3.1. *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.*

The original *Rigidity Lemma* was formulated for sets of target dependencies consisting of EGDs only. A close inspection of the proof in [5] reveals that it remains valid when TGDs are added.

LEMMA 3.4. *(*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.*

PROOF. (SKETCH) (cf. [5]) Unifications performed while chasing EGDs are logically forced, i.e., given the formula $\tau\colon \phi \to x = y$ where ϕ is a *diagram* of the instance J (that is, the conjunction of all facts in J, where all domain elements of J are now treated as first-order variables), $\Sigma_t \models \tau$ holds. Moreover, since J' satisfies Σ_t, it follows that J' satisfies τ.

Assume that x and y are nulls in different blocks of J with $x \backsim y$. Moreover, let h be an arbitrary homomorphism on J'. We have to show that then x is rigid, i.e.: $h([x]) = [x]$.

We construct a valuation V for the terms of ϕ as follows: Let $V(z) = [z]$ if z occurs in the block B of x and $V(z) = h([z])$ otherwise. Let $R(u_1, \dots, u_n)$ be a fact in J (and, therefore, a conjunct in ϕ). Then the fact $R([u_1], \dots, [u_n])$ is in J' by the definition of $[\cdot]$. Moreover, it can be shown (by exactly the same arguments as in [5]), that $V(u_i) = h([u_i])$ holds for every element $u_i \in \mathrm{dom}(J)$. Hence, $R(V(u_1), \dots, V(u_n)) = R(h([u_1]), \dots, h([u_n]))$. The latter tuple is contained in J', since h is an endomorphism. Hence, V is a valid assignment for ϕ in J'. Thus, $V(x) = V(y)$, since J' satisfies τ. Now $V(x) = h([x])$ and $V(y) = [y]$ by definition of V. So $h([x]) = V(x) = V(y) = [y]$. By $x \backsim y$, we have $[x] = [y]$ and, therefore, in total $h([x]) = [y] = [x]$. $\square$

Next, we formalize the idea of considering the blocks of J when searching for a homomorphism of J'.

DEFINITION 3.2. *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 3.3. *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 3.3. 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 which depends only on $\Sigma = \Sigma_{st} \cup \Sigma_t$.*

PROOF. First, we prove that the rigid variables of T^{Σ_t} are also rigid in T_{xy}. Assume to the contrary that $x \in \mathrm{var}(T_{xy})$ is rigid in T^{Σ_t} and that there exists a homomorphism $h\colon T_{xy} \to U$ s.t. $h(x) \neq x$. By Theorem 3.1, h can be transformed into an endomorphism $h'\colon T^{\Sigma} \to U$, s.t. $\forall x \in \mathrm{dom}(h)\colon h(x) = h'(x)$. Thus, we get $h'(x) = h(x) \neq x$, which contradicts the assumption that x is rigid in T^{Σ}.

Hence, the search for a homomorphism $h\colon T_{xy} \to U$ proceeds by checking all possible homomorphisms on the non-rigid blocks of T_{xy} individually. This is justified by the following observation: Let $B_1, \dots, B_n$ denote the non-rigid blocks of T_{xy}. Moreover, for every $i \in \{1, \dots, n\}$, let $h_i\colon B_i \to$

U be a homomorphism. Then the mapping $h\colon T_{xy} \to U$ defined as follows is well-defined and a homomorphism: For every $z \in B_i$, we set $h(z) := h_i(z)$ and for all z outside all B_i (i.e, z is rigid), we set $h(z) := [z]$.

Recall from Lemma 3.3 that T_{xy} has only constantly many variables in addition to T. By Theorem 2.2, the block size of T depends only on Σ_{st}. Hence, also the non-rigid block size of T_{xy} is bounded by a constant depending only on Σ. In principle, we thus get, analogously to Theorem 2.1, the upper bound $O(n \cdot |dom(U)|^c)$, where n is the number of (non-rigid) blocks. However, we are dealing with the situation that U is a retract of T^{Σ_t}, i.e., we already have a retraction $r : T^{\Sigma} \to U$. Hence, in order to search for a homomorphism h with $h(x) = h(y)$ it suffices to inspect the blocks containing x and y and to set $h(z) = r(z)$ for the variables of all other blocks. This allows us to eliminate the factor n from the above upper bound, and the claim of the theorem follows immediately. □

Procedure FindCore^E

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

```
(1)  Chase (S,∅) with $\Sigma_{st}$ to obtain $(S,T) := (S,\emptyset)^{\Sigma_{st}}$;
(2)  Chase T with $\Sigma_t$ to obtain $U := T^{\Sigma_t}$;
(3)  for each $x \in$ var($U$), $y \in$ 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 $U$
               by calling the procedure EXTEND;
(8)          Transform $h'$ into a retraction $r$;
(9)          Set $U := r(U)$;
(10) return U.
```

Putting all these pieces together, we get the FINDCORE^E algorithm. It has basically the same overall structure as the FINDCORE algorithm of [8], 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 3.3, step (5) is based on Lemma 3.4 and Theorem 3.3, and step (7) is based on Theorems 3.1 and 3.2. Analogously to Theorem 2.6, we thus get

THEOREM 3.4. *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 $O(|\text{dom}(S)|^b)$ for some b that depends only on $\Sigma_{st} \cup \Sigma_t$.*

In other words, the asymptotic worst-case behavior of the two algorithms FINDCORE and FINDCORE^E is very similar. In particular, 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 [7] 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. Nevertheless we shall illustrate below by means of experimental results that our new approach may clearly outperform the previous one under realistic assumptions.

4. IMPLEMENTATION

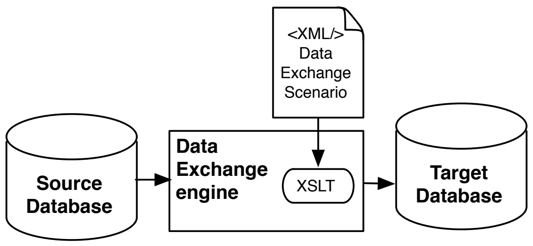
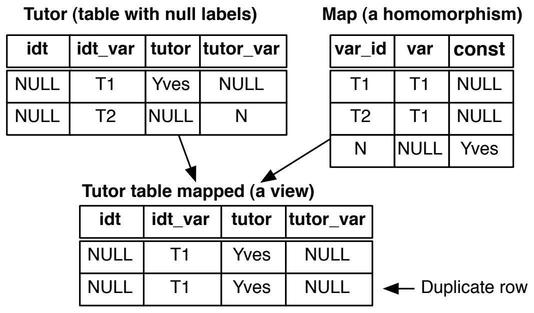


Figure 3: Overview of the implementation.



Tutor (table with null labels)

idt	idt_var	tutor	tutor_var
NULL	T1	Yves	NULL
NULL	T2	NULL	N

Map (a homomorphism)

var_id	var	const
T1	T1	NULL
T2	T1	NULL
N	NULL	Yves

Tutor table mapped (a view)

idt	idt_var	tutor	tutor_var
NULL	T1	Yves	NULL
NULL	T1	Yves	NULL

Duplicate row

Figure 4: Modelling labeled nulls.

We have implemented a prototype system based on the FINDCORE^E algorithm presented in Section 3, relying on a DBMS back-end. Its principal architecture is shown in Figure 3. This approach allowed us to delegate the storage and querying of relational data to the systems best suited for that and concentrate on the core computation itself. Currently, the implementation works with Oracle 11g as well as with the freely available HSQLDB and PostgreSQL. Of course, it can be easily adapted to any other RDBMS.

For specifying data exchange scenarios, we use XML configuration files. The schema of the source and target DB 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 images given by a homomorphism. Figure 4 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 like information on variables (e.g., are they rigid or not) and blocks of the preuniversal instance, information on sibling and parent relations, a log of non-full TGD applications (which is needed by the EXTEND procedure), etc.

A great deal of the core computation is delegated to the target DBMS via SQL commands. Profiling the test runs with our implementation shows that about 90% of the entire time is spent by the database system on SQL processing. Of course, the chase lends itself naturally to an SQL-realization, bearing in mind that the premise and conclusion of dependencies are basically conjunctive queries. But also the various steps of the FINDCORE^E algorithm make heavy use of SQL. 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. Extending the homomorphism h to an endomorphism h' requires finding images for the yet unmapped variables – consistent with the already found mappings. This task is also accomplished by a series of SQL commands.

EXAMPLE 4.1. *Let us revisit the data exchange setting from Example 1.1. Suppose that the canonical solution is*

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

Suppose that we look for a proper endomorphism h' on J. Step 4 of FINDCORE^E *might, e.g., yield the set $T_{N,\text{'Yves'}} =$* `{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 $y = \text{'Yves'}$), s.t. $h(N) = \text{'Yves'}$ has to be found. In the absence of EGDs, non-rigid blocks are the same as usual blocks, and the block of N in $T_{N,\text{'Yves'}}$ 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 $h(N) = \text{'Yves'}$:

```
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 $\{T_2 \leftarrow T_1,\ C_1 \leftarrow C_2\}$. In order to extend $h\colon T_{N,\text{'Yves'}} \to J$ with $\text{var}(T_{N,\text{'Yves'}}) = \{N, C_1, T_2\}$ to an endomorphism h' on J, we have to find images of one variable after the other in $J \setminus T_{N,\text{'Yves'}}$. For instance, the following SQL query finds an image for variable L_2 (generated by the non-full TGD #4) consistent with the previously found mappings for N, C_1, T_2:

```
SELECT NeedsLab.lab_var AS L2
FROM NeedsLab JOIN Teaches ON
       NeedsLab.id_tutor_var = Teaches.id_tutor_var
WHERE Teaches.id_tutor_var = 'T1' AND
       Teaches.id_course_var = 'C2'
```

The query returns L_1, as expected, i.e., $h(L_2) = L_1$.

At every iteration, the algorithm tries to find an endomorphism, that would map a variable on some other term. Since all the variables are distributed among the facts by the chase, we may analyze the dependencies to prune impossible substitutions, e.g., in our running example, it makes no sense to try to unify a variable from the `id_tutor` column with any term from `id_course`. We capture this with the notion of *field partitions*, i.e., sets of fields that possibly share terms. Two fields f_1 and f_2 belong to the same partition, if there is

1. a variable shared between f_1 in the premise and f_2 in the conclusion of the same TGD,
2. a variable shared by f_1 and f_2 in the conclusion of a TGD,
3. an EGD unifying two variables occurring at fields f_1 and f_2 in its premise.

Back to the EXAMPLE 1.1, the target field partitions are `{Course.course}`, `{Tutor.tutor}`, `{NeedsLab.lab}`, `{Course.idc, Teaches.id_course}` and `{Tutor.idt, Teaches.id_tutor, NeedsLab.id_tutor}`.

5. EXPERIMENTS AND DISCUSSION

So far, neither the core computation nor labelled nulls are featured in any data integration tool resp. DBMS and, to the best of our knowledge, no established benchmark for testing such a functionality exists. To conduct our experiments, we synthesized several test cases reflecting common schema transformations: normalization/denormalization and enforcement of additional functional and inclusion dependencies. Our focus was on testing the effectiveness of the minimization phase of FINDCORE^E. Adding or omitting certain functional and inclusion dependencies, synthesizing nearly duplicate tuples in the source database, and rule ordering in the chase allowed us to vary the minimization effort for the core computation from mere checking the optimality of the instance to removing approximately half of the tuples generated by the chase. Some details on the test scenarios are given in Appendix C.

We have run experiments with our prototype implementation on several scenarios with varying size of the schema, the dependencies and the actual source data. Typical runtimes for the core computation are displayed in Figure 5. 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.

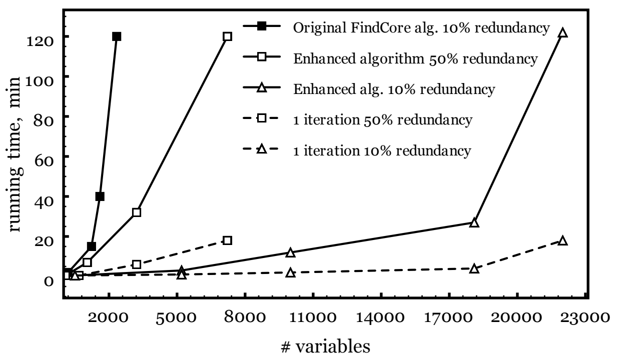


Figure 5: Performance of core computation.

In a setting where the canonical solution had about 50% more nulls than the core, our system handled only about 7,000 nulls in the target DB in 120 min (2nd solid curve from the left). In contrast, about 22,000 nulls were handled in similar time (3rd solid curve) when the canonical solution was only 10% larger than the core.

We have also implemented the FINDCORE algorithm of [8] in order to compare its performance with our algorithm. The left-most curve in Figure 5 corresponds to a run of FINDCORE on the "core-friendly" data exchange problem. The runtime is comparable to (in fact, worse than) the case when the target instance has five times more redundancy, but the extended algorithm is applied. Actually, this is not surprising: The negative effect of simulating the EGDs by TGDs is illustrated by the following simple example:

EXAMPLE 5.1. *Let $J = \{R(x, y), P(y, x)\}$ be a preuniversal instance, and a single EGD $R(z, v), P(v, z) \to z = v$ constitute Σ_t. In order to simulate this EGD by TGDs, the following set of dependencies $\bar{\Sigma}_t$ has to be constructed according to the algorithm in [8]:*

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

where E is an auxiliary predicate representing equality.

Chasing J with $\bar{\Sigma}_t$ (in a nice order), yields the instance

$$J^{\bar{\Sigma}_t} = \{R(x,y), R(x,x), R(y,x), R(y,y), P(y,x), P(y,y), P(x,y), P(x,x), E(x,x), E(x,y), E(y,x), E(y,y)\}.$$

Note that, if a fact contains k occurrences of any of the two terms that have to be unified (in our case, the variables x and y), then the chase produces 2^k variants of this fact.

The core computation applied to $J^{\bar{\Sigma}_t}$ will produce either the solution $\{R(x,x), P(x,x)\}$ or $\{R(y,y), P(y,y)\}$.

On the other hand, if EGDs are directly enforced by the target chase, then the chase ends 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 small number of 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 with respect to the number of nulls eliminated vs. time required. A typical situation is shown in Figure 6: 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:

- Prefer constants over variables.
- Prefer terms already used as substitutions.
- Avoid mapping a variable on itself.

As was already mentioned in Section 3, every intermediate database instance of the FindCore^E algorithm 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 5 correspond 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.

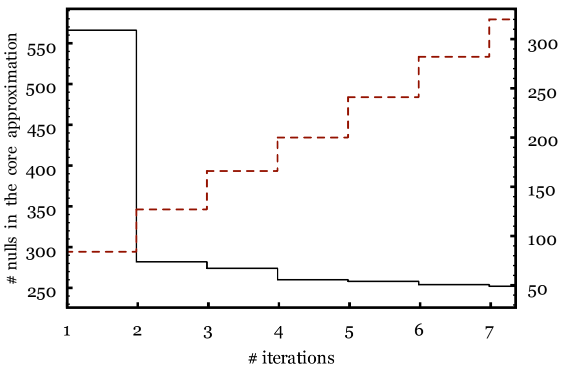


Figure 6: Progress of core computation.

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 four 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. Moreover, it would be desirable to add an estimation of the redundancy in the instance, measuring the remaining "distance" to the core.

(iii) Some phases of the endomorphism search 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.

(iv) Profiling has revealed that currently most of the execution time (about 90%) is spent in the RDBMS when executing the SQL-commands. So far, no efforts of database tuning or SQL tuning (like de-normalization of auxiliary structures) have been made. This is clearly required next.

6. 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 [8], 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 (see Theorem 2.6 and 3.4). Nevertheless, there are some fundamental differences between them, as has been detailed in Section 5. In particular, our approach allows us to strictly separate the search for a solution of a data exchange problem from the core computation and to consider the latter as an optional service. Moreover, the direct treatment of EGDs has led to a performance improvement of an order of magnitude. Another order of magnitude can be gained by contenting ourselves with an approximation to the core, which has been made possible with our new approach.

We have also presented a prototype implementation of our algorithm, which delegates most of its work to the underlying RDBMS 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.

7. REFERENCES

[1] C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.

[2] A. Deutsch and V. Tannen. Reformulation of xml queries and constraints. In *Proc. ICDT'03*, volume 2572 of *LNCS*, pages 225–241. Springer, 2002.

[3] R. Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, 1982.

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

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

[6] G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. On reconciling data exchange, data integration, and peer data management. In *Proc. PODS'07*, pages 133–142. ACM, 2007.

[7] G. Gottlob. Computing cores for data exchange: new algorithms and practical solutions. In *Proc. PODS'05*, pages 148–159. ACM Press, 2005.

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

[9] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *Proc. SIGMOD'05*, pages 805–810. ACM, 2005.

[10] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246. ACM, 2002.

[11] L. Libkin. Data exchange and incomplete information. In *Proc. PODS'06*, pages 60–69. ACM Press, 2006.

[12] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *Proc. VLDB'02*, pages 598–609. Morgan Kaufmann, 2002.

APPENDIX

A. PROOF OF THEOREM 3.1

THEOREM 3.1 *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 $O(|\mathrm{dom}(T)|^b)$ 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. Suppose that the chase of a preuniversal instance T with Σ_t has length n. Then we write T_s with $0 \le s \le n$ 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 $\forall x \in \mathrm{dom}(h_s)$, such that $[x] \in \mathrm{dom}(h)$: $h_s(x) = h([x])$ holds. We claim that for every $s \in \{0, \dots, n\}$, such a homomorphism h_s consistent with h exists. Then $h' = h_n$ is the desired homomorphism. The only part missing in the proof in Section 3 was a proof of this claim by induction on s.

[induction begin.] We define $h_0\colon T = T_0 \to B$ by setting $h_0(x) = h([x])$ for all $x \in \mathrm{dom}(T)$. Then h_0 is consistent with h by definition. By condition 2 of the theorem, all facts of T are in W and $W \subseteq T^{\Sigma_t}$. Hence, for every fact $P(u_1, \dots, u_k) \in T_0$, we have $P([u_1], \dots, [u_k]) \in W$ and, therefore, $P(h(u_1), \dots, h(u_k)) = P(h([u_1]), \dots, h([u_k])) \in B$. Hence h_0 is the desired homomorphism.

[induction step.] Let $h_{s-1}\colon T_{s-1} \to B$ be a homomorphism, s.t. h_{s-1} is consistent with h. At step s of the chase, there are four types of dependencies that can be enforced:

1. an EGD,
2. a full TGD,
3. a non-full TGD, introducing facts not present in W.
4. a non-full TGD, introducing facts present in W.

Note that cases 3 and 4 do not intersect, by Lemma 3.2 and by the fact that W is closed under siblings.

Below we show that in each of these 4 cases, it is indeed possible to transform $h_{s-1}\colon T_{s-1} \to B$ into a homomorphism $h_s\colon T_s \to B$ consistent with h. The following simple fact is used throughout the proof: if there is an assignment $\vec{a} \in \mathrm{dom}(T_i)$ for some conjunction $\phi(\vec{x})$ s.t. $T_i \models \phi(\vec{a})$, and $h_i\colon T_i \to B$ is a homomorphism, then $B \models \phi(h_i(\vec{a}))$.

Case 1. T_s is obtained from T_{s-1} via the EGD $\varphi(\vec{x}) \to x_i = x_j$, where $i, j \le |\vec{x}|$ s.t. $T_{s-1} \models \phi(\vec{a})$. W.l.o.g., $a_i \in \mathrm{var}(T_{s-1})$ is a variable and T_s is obtained from T_{s-1} by replacing every occurrence of a_i by a_j. Clearly, $\mathrm{dom}(T_s) = \mathrm{dom}(T_{s-1}) \setminus \{a_i\}$. We claim that $h_s = h_{s-1}|_{\mathrm{dom}(T_s)}$ is the desired homomorphism, i.e. h_s is obtained from h_{s-1} simply by restricting its domain.

Let $P(\vec{b})$ be a fact in T_s. Then either $P(\vec{b})$ is also a fact in T_{s-1} (not containing the variable a_i) or T_{s-1} contains some fact $P(\vec{c})$, s.t. $\vec{b} = \vec{c}[a_i \leftarrow a_j]$, i.e., $\vec{b}$ is obtained from $\vec{c}$ by replacing all occurrences of a_i with a_j. In the former case, we clearly have $P(h_s(\vec{b})) = P(h_{s-1}(\vec{b})) \in B$. It remains to consider the latter case: We again have $P(h_{s-1}(\vec{c})) \in B$. In order to show that also $P(h_s(\vec{b})) = P(h_{s-1}(\vec{c})) \in B$, it suffices to show that $h_{s-1}(a_i) = h_{s-1}(a_j)$. Indeed, we have $T_{s-1} \models \phi(\vec{a})$, since the EGD $\varphi(\vec{x}) \to x_i = x_j$ fires with this assignment in step s of the chase. Then $B \models \phi(h_{s-1}(\vec{a}))$, since h_{s-1} is a homomorphism. By condition 1 of the Theorem, $B \models \Sigma$. In particular, the EGD $\varphi(\vec{x}) \rightarrow x_i = x_j$ holds in B. But then $h_{s-1}(a_i) = h_{s-1}(a_j)$.

Case 2. A full TGD $\phi(\vec{x}) \rightarrow \psi(\vec{x})$ leaves the domain unchanged. Thus, we simply set $h_s = h_{s-1}$. Suppose that $\phi(\vec{x})$ was satisfied by T_{s-1} with some assignment $\vec{a}$. Hence, the only facts introduced by this chase step are atoms $\psi(\vec{a})$. We have to show that $\psi(h_s(\vec{a}))$ (which is identical to $\psi(h_{s-1}(\vec{a}))$) holds in B. We use the same argument as above: $T_{s-1} \models \phi(\vec{a})$ holds, since the TGD τ fires with this assignment on $\vec{x}$. Hence, $B \models \phi(h_{s-1}(\vec{a}))$, since h_{s-1} is a homomorphism Finally, since $B \models \Sigma$, also $B \models \psi(h_{s-1}(\vec{a}))$ holds.

Case 3. T_s is obtained from T_{s-1} via the non-full TGD $\phi(\vec{x}) \rightarrow \psi(\vec{x}, \vec{y})$ with assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$. Moreover, all atoms in $\psi(\vec{a}, \vec{z})$ are outside W. As above, we have $T_{s-1} \models \phi(\vec{a})$ and $B \models \phi(h_{s-1}(\vec{a}))$. Moreover, by $B \models \Sigma$, there exist a vector $\vec{c}$ of terms in dom(B), s.t. $\psi(h(\vec{a}), \vec{c}) \subseteq B$. By Lemma 3.2, all atoms in $\psi(h(\vec{a}), \vec{c})$ are newly created in T_s and, hence, all terms in $\vec{z}$ are fresh nulls. We extend h_{s-1} to h_s by setting $h_s(\vec{z}) := \vec{c}$. Then h_s is a homomorphism, since the image $\psi(h(\vec{a}), \vec{c})$ of the new atoms $\psi(h(\vec{a}), \vec{z})$ in T_s is in B by definition. Moreover, h_s is consistent with h, since h_{s-1} is consistent with h and h_{s-1} differs from h_s only on variables $\vec{z}$ outside dom(T).

Case 4. T_s is obtained from T_{s-1} via the non-full TGD $\phi(\vec{x}) \rightarrow \psi(\vec{x}, \vec{y})$ with assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$. Moreover, W already contains a fact for every atom in $\psi(\vec{a}, \vec{z})$. Analogously to case 3, the vector $\vec{z}$ consists of fresh nulls. Moreover, since all atoms of $\psi([\vec{a}], [\vec{z}])$ are contained in W, the homomorphism $h\colon W \rightarrow B$ is defined on all variables occurring in $\psi([\vec{a}], [\vec{z}])$. Since h is a homomorphism, we have $B \models \psi(h([\vec{a}]), h([\vec{z}]))$. We extend h_{s-1} to h_s by setting $h_s(\vec{z}) := h([\vec{z}])$ and $h_s(x) := h_{s-1}(x)$ for all variables $x \in$ dom(h_{s-1}). In other words, native positions in $\psi(\vec{a}, \vec{z})$ are mapped consistently with h. In order to show that h_s is a homomorphism, it remains to prove that all atoms in $\psi(h_s(\vec{a}), h_s(\vec{z}))$ are contained in B. By definition, we have $h_s(\vec{z}) = h([\vec{z}])$. Hence, it suffices to show that $h_{s-1}(\vec{a}) = h([\vec{a}])$ holds.

W is closed under parents and siblings and, therefore, the origin of every position of $\psi(\vec{a}, \vec{z})$ is contained in W, by the definition of the parent relation over facts. According to Lemma 3.1, a position p and its origin position o_p (which is either contained in some fact in T or which was introduced previously at some chase step $k < s$) are always occupied by the same term. If the position o_p is contained in some fact in T, then the term u at o_p was mapped to $h([u])$ by h_0 according to the induction begin. If o_p is contained in some fact introduced at some chase step $k < s$, then the term u at o_p was mapped to $h([u])$ by h_k according to Case 4. Note that none of the four cases considered by the induction step modifies a previously chosen image. Hence, o_p is mapped to $h([u])$ also by h_{s-1}. Hence, we indeed have $h_{s-1}(\vec{a}) = h([\vec{a}])$, as required.

This concludes the induction. □

B. PROOF OF THEOREM 3.2

THEOREM 3.2 *Let T, T^{Σ_t}, B, W, and $h\colon W \rightarrow B$ be as in Theorem 3.1. Then the procedure* EXTEND *extends h to a homomorphism $h'\colon T^{\Sigma} \rightarrow B$.*

PROOF. The only part missing is the proof of the following properties: $W_j \subseteq T^{\Sigma_t}$, W_j contains all facts from T, W_j is closed under parents and siblings, and $h_j\colon W_j \rightarrow B$ is a homomorphism s.t. $\forall x \in$ dom$(W)\colon h'(x) = h(x)$ holds. We prove this claim by induction on j:

[induction begin.] When the while-loop is entered for the first time, we have $W_1 = W$ and the above properties are trivially fulfilled.

[induction step.] Suppose that the while-loop is entered for the $(j+1)$-st time. By the induction hypothesis, W_j together with the homomorphism $h_j\colon W_j \rightarrow B$ fulfills the assumptions on W in Theorem 3.1. Hence, h_j can be extended to a homomorphism $h\colon T^{\Sigma_t} \rightarrow B$, s.t. $\forall x \in$ dom$(h_j)\colon h'(x) = h_j(x)$. Then it is of course also possible to extend h_j to the homomorphism $h_{j+1}\colon W_{j+1} \rightarrow B$ where $W_{j+1} = W \cup S \subseteq T^{\Sigma_t}$, s.t. S is a set of siblings whose parents are in W_j.

This concludes the induction. □

C. TEST CASE EXAMPLE

For illustrative purposes, we describe in this appendix one of the test scenarios used for evaluation of the system. The source schema comprises four denormalized tables, providing information on university departments and their staff members, their publications and received grants:

- `Articles(FirstAuthor, LastAuthor, Title, Journal, IssueDate, CiteSystem)`. *CiteSystem takes either the value 'ABC' or 'Medical' – the former means that authors are listed in alphabetical order while the latter attaches a special role of main contributor to the first author and that of principal investigator (team lead) to the last author.*
- `GrantResults(GrantName,University, ProjectLead, FinalArticle)`.
- `FacultyLeaders(University, Department, Prof1, Prof2)`.
- `TopFundRaisers(University, Researcher, Rank)`.

The target schema represents the same information in a normalized form:

- `Department(DeptId, University, DeptName)`.
- `Researcher(ResearcherId, Name)`.
- `Affiliation(ResearcherId, DeptId)`.
- `Grant(GrantCode, DeptId, PcInvestId, FinalArticleId)`. *Besides the grant code and the department ID, grants are associated with a principle investigator and with a final project report.*
- `Article(ArtId, Title)`.
- `Journal(JournalId, JournalName)`.
- `Publication(ArtId, JournalId, PubDate)`.
- `Author(ResearcherId, ArtId, Role)`. *Role can be either Principal Investigator ('PI') or Main Contributor ('MC').*

There are eleven source-to-target dependencies in the data exchange setting. Recall from Section 2.1 that we normally write dependencies without quantifiers: variables occurring in the conclusion of the formula but not in its premise are assumed to be existentially quantified; all other variables are universally quantified. Below, we shall further simplify the notation in the following ways: We replace variables occurring only once in the formula by underscores (names of those variables are irrelevant). Moreover, we replace successive underscores with dots. Finally, we shall separate the conjuncts in either the premise or conclusion of a dependency by a comma rather than the $\wedge$-symbol.

- `Articles(..., Title, Journal, IssueDate,_)` → `Article(ArtId, Title), Journal(JnId, Journal), Publication(ArtId, JnId, IssueDate)`.
- `Articles(Aut1, _, Title,..., 'ABC' )` → `Researcher(ResId, Aut1), Article(ArtId, Title), Author(ResId, ArtId, _)`. *If authors are listed in alphabetical order, we cannot infer special roles ('MC' or 'PI') of the authors.*

- `Articles( Aut1, _, Title,..., 'Medical')` → `Researcher(ResId, Aut1), Article(ArtId, Title), Author(ResId, ArtId, 'MC')`. *For medical articles, the first author is usually the main contributor.*
- `Articles(_, AutLast, Title,..., 'ABC')` → `Researcher(ResId, AutLast), Article(ArtId, Title), Author(ResId, ArtId, _)`.
- `Articles( AutLast, Title, 'Medical' )` → `Researcher(ResId, AutLast), Article(ArtId, Title), Author(ResId, ArtId, 'PI')`. *For medical articles, the last author is usually the principal investigator of the team.*
- `GrantResults(..., ProjLead, FinalArticle)` → `Researcher(ResId, ProjLead), Article(ArtId, FinalArticle), Author(ResId, ArtId, 'PI')`. *Assume that the project lead is a co-author of the final report.*
- `GrantResults(GrantName, Uni, ProjLead, _)` → `Grant(GrantName, DeptId,...), Department(DeptId, Uni, _), Researcher(ResId, ProjLead), Affiliation(ResId, DeptId)`.
- `GrantResults(GrantName,..., FinalArticle)` → `Grant(GrantName,..., ArtId), Article(ArtId, FinalArticle)`.
- `FacultyLeaders(Uni, Dept, Prof1, _)` → `Department(DeptId, Uni,Dept), Researcher(ResId, Prof1), Affiliation(ResId, DeptId)`.
- `FacultyLeaders(Uni, Dept, _, Prof2 )` → `Department(DeptId, Uni,Dept), Researcher(ResId, Prof2), Affiliation(ResId, DeptId)`.
- `TopFundRaisers(Uni, Researcher, _)` → `Department(DeptId, Uni, _), Researcher(ResId, Researcher), Affiliation(ResId, DeptId), Grant(_, DeptId, ResId, _)`.

Target dependencies further restrain the situation with staff, publications and grants in the addressed universities. One can either include or omit any of those dependencies, or introduce arbitrary variations thereof to obtain more or less nulls in the target instance.

- `Article(Id1, Title), Article(Id2, Title)` → `Id1=Id2`. *Assume that article title is unique.*
- `Journal(Id1, Jrn), Journal(Id2, Jrn)` → `Id1 = Id2`. *Assume that journal name is unique.*
- `Researcher(Id2, Name), Researcher(Id1, Name)` → `Id1 = Id2`. *Assume that researcher name is unique.*
- `Grant(GrantCode, Dept1, Researcher1, Article1), Grant(GrantCode, Dept2, Researcher2, Article2)` → `Dept1 = Dept2, Researcher1=Researcher2, Article1 = Article2`. *Inclusion of this rule allows to deprecate joint grants.*
- `Department(Id1, Uni, Dept), Department(Id2, Uni, Dept)` → `Id1 = Id2`. *Assume that department id is unique.*
- `Author(AuthorId,...)`→ `Researcher(AuthorId,_)`. *List all authors as researchers*
- `Author(AuthorId, ArtId, _)` → `Article(ArtId, _), Publication(ArtId, JnId, _), Journal(JnId, _)`. *Authors must be published.*
- `Publication(ArtId,...)` → `Author(_, ArtId, _)`. *Publications must be associated with at least one author.*
- `Affiliation(ResId,DeptId1), Department(Dept1, DeptName1, Uni), Affiliation(ResId, DeptId2), Department(DeptId2, DeptName2, Uni)` → `DeptId1 = DeptId2, DeptName1 = DeptName2`. *We assume that one researcher cannot work in more than one department of the same university.*
- `Researcher(ResId, Name)` → `Department(DeptId, Uni, DeptName), Affiliation(ResId, DeptId)`. *Assume that all researchers are affiliated.*
- `Grant(_, DeptId, ResearcherId, ArtId), Department(DeptId, _, _)` → `Author(ResearcherId, ArtId, 'PI'), Affiliation(ResearcherId, DeptId)`. *Issue of a grant to a department implies that the principal investigator is employed in that department.*