Towards Practical Feasibility of Core Computation in Data Exchange

Reinhard Pichler

Vadim Savenkov

Vienna University of Technology

Abstract

Data exchange is concerned with the transfer of data from some source database to some target database. Given a source instance, there may be many solutions, i.e., target instances. The most compact solution is called the core. Gottlob and Nash have recently presented a core computation algorithm which works in polynomial time under very general conditions. In this paper, we present an enhanced version of this algorithm. Moreover, we also report on a proof-of-concept implementation of the enhanced algorithm and on the experience gained from experiments with this implementation.

Key words: data exchange, data integration, core, universal solutions, dependencies, chase, tractability, query evaluation

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, 7], data exchange aims at actually materializing a target database stemming from some source database [5]. 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. Moreover, the target database may also impose additional integrity constraints, called target dependencies. Following [5, 6], we confine ourselves to relational schemas and to dependencies which may either be *tuple generating dependencies* (tgds) or *equality generating dependencies* (egds) [2].

The source schema $\mathbf{S}$ and the target schema $\mathbf{T}$ together with the set Σ_{st} of source-to-target dependencies and the set Σ_t of target dependencies constitute

Email addresses: pichler@dbai.tuwien.ac.at (Reinhard Pichler), savenkov@dbai.tuwien.ac.at (Vadim Savenkov)

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 source-to-target dependencies Σ_{st} and target dependencies Σ_t are satisfied. Such a J is called a *solution* to the data exchange problem. 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 source-to-target dependencies:*

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

and the target dependencies are given by the two tgds:

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

Then the following instances are all valid solutions:
$J = \{$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)$\}$,
$J_c = \{$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 [5]) 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 thus, 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* [6], which is the most compact one. For instance, solution J_c in Example 1.1 is a core.

Fagin et al. [6] gave convincing arguments that, in many cases, the core should be the database to be materialized. Since then, it has been identified as an essential concept for query answering in data exchange, in particular, for aggregate queries [1] and for systematic treatment of incompleteness [11]. Moreover, Gottlob and Nash [8] showed that the core can be computed in polynomial time under very general conditions. But 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. Moreover, this simulation of egds by the FindCore algorithm is done in such a way that, in general, no intermediate results are solutions to the data exchange problem. Only at the very end of the core computation, when FindCore has reached the core, we can be sure that we have a solution to the data exchange problem. In other words, 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 such that we first of all compute a solution to a data exchange problem – no matter whether this solution is later reduced to the core or not.

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 largely uncoupled. In fact, some additional information needed later by the core computation has to be stored when the data-exchange problem is solved by the chase, e.g., the precise chase sequence defined by the sequence of dependencies and the corresponding variable instantiations is needed by the core computation. However, the core computation as such is not required in order to arrive at a universal solution of the data-exchange problem. Hence, the core computation can 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 as witnessed by our experiments. 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. This is in great contrast to the FindCore algorithm from [8], where the intermediate results of the core computation are, in general, not solutions (due to the simulation of egds). Hence, if we stop the FindCore before its completion, we will not get an approximation of the core since, in general, we will thus not even get a solution to the data exchange problem.

(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. 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 $(\mathbf{S}, \mathbf{T})$ 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 (I, J) is an instance of the schema $(\mathbf{S}, \mathbf{T})$.

Embedded dependencies [4] 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* and *target dependencies*. In source-to-target dependencies, the premise may only use relation symbols from the source schema while the conclusion may only use relation symbols from the target schema. In target dependencies, both the premise and the conclusion may only use relation symbols from the target schema. Note that source dependencies may be important for deriving source-to-target dependencies (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 source-to-target dependencies Σ_{st} and the set of target dependencies Σ_t. As source-to-target dependencies, we only consider tuple generating dependencies here (tgds, see definition below), which are referred to as *st-tgds*. The *data exchange problem* associated with this setting is the following: Given a (ground) source instance I, find a target instance J, s.t. $(I, J) \models \Sigma_{st}$ 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 [5, 6], we consider dependencies in Σ_{st} and Σ_t of the following forms: Each source-to-target dependency 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 target dependency 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* [2], which iteratively introduces new facts or equates terms until all desired dependencies are fulfilled. 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 $a_i, a_j \in \text{const}(I)$ 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 termination of the chase is that the tgds be *weakly acyclic* (see [3, 5]). 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 st-tgds 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 shows the dependency graph for the target tgds in Example 1.1, where special edges are dashed. Clearly, this graph has no cycle containing a special edge (actually, it contains no cycle at all). Hence, these tgds are weakly acyclic.

Remark. At this point it is important to note that in [8], the definition of weakly acyclic tgds is slightly more restrictive than the original definition from [3, 5] recalled above. More precisely, in [8], there is a special edge $\left(R^i, S^j\right)$ in the dependency graph if x occurs in R^i and there is a variable $y \in \vec{y}$ in the field S^j – even if x does not occur in some other field T^k in $\psi(\vec{x}, \vec{y})$. In [8], the following simple example was given to illustrate the effect of this more restrictive definition of weak acyclicity: Consider the set Σ of tgds consisting of a single tgd $R(x, y) \rightarrow R(x, z)$. In the definition of [3, 5], there is no self-loop on position $R.2$ since y does not occur on the right-hand side of the tgd. Hence, Σ is weakly acyclic according to [3, 5]. In contrast, by the definition of [8], this singleton Σ is not weakly acyclic since the dependency graph contains a self-loop on position $R.2$. The entire core computation algorithm in [8] is based on this more restrictive notion of weak acyclicity. We shall point of below where in the algorithm this assumption is crucial. However, it is also shown in [8] that the core computation problem for the notion of weak-acyclicity according to [3, 5] can be easily reduced (in fact, in linear time) to the notion of weak-acyclicity according to [8]: Suppose that Σ contains a tgd of the form $\phi(\vec{x}, \vec{y}) \rightarrow \psi(\vec{x}, \vec{z})$, s.t. all variables in $\vec{x}$ do occur in ψ while the variables in $\vec{y}$ occur in ϕ only. Then one can replace this tgd by the following two tgds: $\phi(\vec{x}, \vec{y}) \rightarrow Q(\vec{x})$ and $Q(\vec{x}) \rightarrow \psi(\vec{x}, \vec{z})$, where Q is a fresh predicate symbol. For the resulting set of tgds, the definition of weak-acyclicity in [3, 5] and in [8] coincide. Moreover, core computation for Σ is equivalent to core computation for Σ' followed by the elimination of all atoms with leading symbol Q. At any rate, we have decided to adhere to the "standard notion" of weak-acyclicity since, for the actual implementation, the problem reduction sketched above would incur an unnecessary, additional cost (even though this does of course not matter if one primarily wants to prove the polynomial time upper bound on core computation).

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 $R(\vec{x}) \in I$, 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. $r \circ r = r$. The image $r(I)$ under a retraction r is called a *retract* of I. 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. An instance is called a *core* if it has no proper endomorphisms. A core C of an instance I is an endomorphic image 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 a *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 a *canonical universal solution*. Both T and U can be computed in polynomial time w.r.t. the size of the source instance [5]. Clearly, both the preuniversal instance and the canonical universal solution depend on the chase order, i.e., the order in which the source-to-target tgds respectively the target tgds and egds are applied. Moreover, even if the chase order is fixed, the resulting instances are only unique up to variable renaming. Nevertheless, by slight abuse of notation, we shall sometimes refer to these instances as *the* preuniversal instance and *the* canonical universal solution, respectively.

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 $\tau\colon \phi(\vec{x}) \rightarrow \psi(\vec{x}, \vec{y})$ 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 [6], 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 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. In [6], the following important results concerning the block size were proved:

Theorem 2.1. [6] *Let A and B be instances, and suppose that* $\text{blocksize}(A) \leq c$ *holds. Then the check if a homomorphism $h\colon A \rightarrow B$ 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 $h\colon A \rightarrow B$, we may search for homomorphisms from every block of A

Procedure FindCore

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

(1) Chase (S,$\emptyset$) with Σ_{st}
to obtain $(S,T) := (S,\emptyset)^{\Sigma_{st}}$;
(2) Compute $\bar{\Sigma}_t$ from Σ_t;
(3) Chase T with $\bar{\Sigma}_t$ (using a nice order)
to get $U := T^{\bar{\Sigma}_t}$;
(4) **for** each $x \in \mathrm{var}(U)$, $y \in \mathrm{dom}(U)$, $x \neq y$ **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.

onto B separately. Note that A has $\leq |A|$ blocks, each containing $\leq c$ variables. Hence, from each block of A, we have to consider $\leq |B|^c$ possible mappings. □

Theorem 2.2. [6] *If Σ_{st} is a set of st-tgds of height e, S is ground, and $(S,T) = (S,\emptyset)^{\Sigma_{st}}$, then* blocksize$(T) \leq e$.

Sibling, parent, ancestor. Consider the chase of the preuniversal instance T with target dependencies 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 target schema $\mathbf{T}$ is transformed into the set $\bar{\Sigma}_t$ of tgds over the schema $\mathbf{T} \cup \{E\}$, where E (encoding equality) is a binary relation not present in $\mathbf{T}$. 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 $R \in \mathbf{T}$ 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 $R \in \mathbf{T}$ and $i \in \{1, 2, \ldots, k\}$.

Even if Σ_t was weakly acyclic, $\bar{\Sigma}_t$ may possibly be not. 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 efficiently transformed into a retraction [8]:

Theorem 2.3. [8] *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 2.4. [8] *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 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 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 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 Σ. So we first search for a homomorphism $h\colon T_{xy} \rightarrow U$ with $h(x) = h(y)$; and then h is extended to a homomorphism $h\colon U \rightarrow U$, 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* $O(|\mathrm{dom}(T)|^c)$.

Recall our discussion on different notions of weakly acyclic tgds in Section 2.1. In fact, the above lemma crucially depends on the more restrictive definition of weak acyclicity given in [8], while it would not hold under the more general definition of weak acyclicity according to [3, 5]. We shall come back to this point in Example 3.5 below, when we present our modified notion of siblings and parents (which guarantees that our analogue of Lemma 2.1 also holds for the more general notion of weak acyclicity). But of course, as we have already mentioned in Section 2.1, core computation under the more general notion of weak acyclicity can be easily reduced to core computation under the more restrictive one.

Theorem 2.5. (LIFTING) [8] *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 parents 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 $E(a_i, a_j)$ for distinct constants a_i and 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 st-tgds* Σ_{st} *and target dependencies* Σ_t. *Moreover, let* S *be a ground instance of the source 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 $h'\colon U \to U$, and
- 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 preuniversal 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 st-tgds and in all atoms occurring in target dependencies, we have to add a unique existentially-quantified variable at position 0. For example, the source-to-target tgd $\tau\colon S(x) \rightarrow R(x, y)$ is transformed into $\tau^{id}\colon S(x) \rightarrow R^{id}(t, x, y)$ 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 $R(x_1, x_2, \ldots, x_n)$ 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. These id's allow us to refer to facts in an unambiguous way no matter how the attribute values in a fact are altered by the application of egds during the chase. The id's are thus helpful for the discussion in this paper and they are also important for the implementation of our core computation algorithm.

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. When we say that a "fact A is present in some set W", then we mean that a fact with the same id as A is present in W.
3. However, when we explicitly write that two facts are equal then we actually mean that they have the same attribute values.

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* $\{S(id_S, x, y) \rightarrow P(id_P, y, z); P(id_P, y, z) \rightarrow Q(id_Q, y, v)\}$ *as the set of target dependencies 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:* $(id_{S1}, 1)$ *for* x_1*,* $(id_{S1}, 2)$ *for* y_1*,* $(id_{P1}, 2)$ *for* z_1*, and* $(id_{Q1}, 2)$ *for* v_1*.*

Lemma 3.1. *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.*

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 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 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. The following example illustrates that the normal form may possibly lead to a significantly smaller canonical universal instance than non-normalized tgds.

Example 3.2. *Let* $\Sigma = \{P(x_1, x_2), Q(x_2, x_3) \rightarrow R(x_1, y_1), S(x_3, y_2)\}$ *and let instance* $T = \{P(1,1), P(1,2), \ldots, P(1,n), Q(1,1), Q(2,2), \ldots, Q(n,n)\}$ *for an arbitrary integer* $n \geq 1$. *Chasing* T *with* Σ *produces (among others) the new facts* $R(1,u_1), S(1,v_1)$, $R(1,u_2), S(2,v_2)$, $\ldots, R(1,u_n), S(n,v_n)$, *where* $R(1,u_2), \ldots, R(1,u_n)$ *are clearly redundant.*

Now consider the normal form Σ' *of* Σ *with* $\Sigma' = \{P(x_1, x_2), Q(x_2, x_3) \rightarrow R(x_1, y_1); P(x_1, x_2), Q(x_2, x_3) \rightarrow S(x_3, y_2)\}$. *In this case, the chase no longer produces the facts* $R(1,u_2), \ldots, R(1,u_n)$.

In the proof of Theorem 3.1, we will have to make sure that all facts on the right-hand side of a non-full tgd are indeed created when the non-full tgd fires. For this purpose, it would suffice that the right-hand side of non-full tgds contains only atoms with at least one existentially quantified variable occurring in them. Below, we use the normal form to establish a slightly stronger property of dependencies in normal form.[2]

Lemma 3.2. *Let the preuniversal instance* J *be chased with the set* Σ_t *of target dependencies 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

[2]Strictly speaking, in the presence of id's, it is excluded anyway that some atom on the right-hand side of some tgd contains no existentially quantified variable.

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 3.3. *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 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 $\tau\colon \phi(\vec{x}) \rightarrow \psi(\vec{x}, \vec{y})$ 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 fact.

Example 3.4. *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 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* $S(x_1, y_1)$.

Example 3.5. *Consider the set* Σ *of tgds (we omit the id's since, in the absence of egds, the attribute values suffice to uniquely identify the atoms) with* $\Sigma = \{A(x, y) \rightarrow (\exists u)R(u, x, y);\ R(u, x, y), R(v, y, z) \rightarrow (\exists w)R(w, x, z)\}$. *Moreover, for arbitrary integer* $n \geq 1$, *let the instance* I_n *be defined as* $I_n = \{A(1, 2), A(2, 3), \ldots, A(n-1, n)\}$.

The chase of I_n with Σ introduces all facts of the form $R(z, i, j)$, where z is a variable not occurring anywhere else and i, j are integers with $i < j$. In particular, the following facts are introduced via the chase: $R(u_1, 1, 2), R(u_2, 2, 3), \ldots$, $R(u_{n-1}, n-1, n)$ as well as $R(v_1, 1, 3), R(v_2, 1, 4), \ldots, R(v_{n-2}, 1, n)$.

Now consider the fact $R(v_{n-2}, 1, n)$ in I_n^Σ. Our definition of parents (via facts) traces back the origin of labelled nulls. However, the only labelled null in the above fact (i.e., v_{n-2}) is native *to this fact. Hence, $R(v_{n-2}, 1, n)$ has no parents and, therefore, no ancestors according to our fact-based definition.*

In contrast, by the definition of parents and ancestors (via variables) in [8], which we recalled in Section 2.1, the fresh variable v_{n-2} resulting from an application of the second tgd in Σ has as parents v_{n-3}, 1, $n-1$, u_{n-1}, $n-1$ and n. The variable u_{n-2} only has the constants $n-2$ and $n-1$ as parents while v_{n-3} again has six parents, namely v_{n-4}, 1, $n-2$, u_{n-2}, $n-2$ and $n-1$. In total, all of the variables $v_1, \ldots, v_{n-3}$ are ancestors of v_{n-2} according to the (variable-based) definition in [8].

Clearly, in the above example, Lemma 2.1 does not allow us to derive a constant upper bound on the number of ancestors (via variables). By the restrictive notion of "weak acyclicity" applied in [8], such a case can never occur, since Σ is clearly not weakly acyclic in the sense of [8]. On the other hand, Σ is weakly acyclic according to the more general definition of weak acyclicity from [3, 5]. Recall that, in this paper, we adhere to the latter notion of weak acyclicity. Indeed, with our definition of parents and sibling (via facts rather than variables), we end up with constantly many ancestors (independently of n and, hence, of the size of the instance I_n). In Lemma 3.3 below (which is the analogue of Lemma 2.1) we shall show that such a constant upper bound on the number of ancestors (defined via facts) can always be guaranteed.

Some useful notation. To reason about the effects of egds, it is convenient to introduce some additional notation, following [6]. 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 target dependencies, 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 $\forall u \in \text{dom}(J)\, u = [u]$ holds. The following property of $[\cdot]$ is immediate:

Proposition 3.1. *The mapping $[\cdot]\colon J \to J'$ 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

$h\colon T_{xy} \to U$ (Lemma 3.5 and Theorem 3.3), and lifting a homomorphism $h\colon T_{xy} \to U$ to a non-injective homomorphism $T^{\Sigma_{st}} \to U$ (Lemma 3.4, Theorem 3.1, and Theorem 3.2).

Lemma 3.3. *For every weakly acyclic set Σ of tgds and egds, 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. $\{Origin_x, Origin_y\} \subseteq T_{xy}$,
2. *all facts of T are in T_{xy}, and $T_{xy} \subseteq T^{\Sigma}$,*
3. *every fact in W was either already present in T or it was introduced by the application of a non-full tgd,*
4. *T_{xy} is closed under parents and siblings over facts, and*
5. $|\mathrm{dom}(T_{xy})| \leq |\mathrm{dom}(T)| + b$.

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

Proof. Let d denote the depth of Σ. Given variable $z \in \{x, y\}$, let the set F_z (= the "family" of z) denote the set of *facts* as follows:

1. Initially, we set $F_z := \{Origin_z\}$.
2. For every fact A in F_z, we add all siblings of A to F_z.
3. For every foreign position (id, j) in F_z, we add the parent of (id, j) to F_z.

Steps 2 and 3 are iterated, until no further fact is added to F_z. We first show that the number of iterations is bounded by the depth d of Σ: Suppose that we apply Step 3 for the i-th time (with $i \geq 1$). Let m denote the maximum depth of those positions in the current set F_z for which the parent is not present in F_z yet. Then the positions introduced by the i-th application of Step 3 and by the $i+1$-st application of Step 2 have depth at most $m-1$. This follows from the definition of parents which are obtained by tracing backward a chase sequence including at least one special edge in the dependency graph. Hence, the maximum depth of the positions added to F_z in each iteration of Step 3 followed by Step 2 strictly decreases. Thus, the number of iterations is indeed bounded by the depth d of Σ.

Now let $T_{xy} := T \cup F_x \cup F_y$. By the construction of F_x and F_y, the set T_{xy} contains $Origin_x$ and $Origin_y$ and T_{xy} is closed under the parent and sibling relations. Moreover, T_{xy} contains only facts which are derived by the chase of T with Σ (recall that we identify a fact with its id; hence, the id's of the facts in T_{xy} are all present in T^{Σ}, even though the values of the attributes in these facts may have been changed by the application of egds.) Thus, T_{xy} satisfies the conditions 1–3.

Before we prove the desired upper bound on the domain size of T_{xy} we show that the number of facts in F_x is bounded by some constant depending on Σ: Every fact has at most constantly many siblings with at most constantly many positions each. Hence, each execution of Step 2 only adds constantly many new positions to F_x. Likewise, every fact added to F_x has at most constantly

many foreign positions. Hence, only constantly many parents are added to F_x whenever Step 3 is executed. Finally, the total number of iterations of Steps 2 and 3 is bounded by the depth d of Σ (which is considered as constant). Thus, in total, the exhaustive application of Steps 2 and 3 of the definition of F_x and F_y introduces only constantly many new facts and, therefore, only constantly many new positions.

Clearly, also the number of new variables introduced whenever a new fact is added to F_x or F_y is bounded by a constant of Σ. Moreover, 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. Hence, we get the desired inequality $|\mathrm{dom}(T_{xy})| \leq |\mathrm{dom}(T)| + b$. 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}. □

Having a homomorphism $h\colon T_{xy} \to U$, we want to extend it to a homomorphism $h'\colon T^{\Sigma_{st}} \to U$, 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, in Theorem 3.1 below we show that also in our case, the lifting can be performed efficiently. First, we define an important property of variables in a subset W of the canonical universal instance and prove sufficient conditions for guaranteeing this property.

Definition 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. Suppose that T^{Σ_t} is obtained from T by a chase sequence of length n. Finally, let W be an instance, s.t. all facts of T are in W (i.e. W contains facts with the same id's) and $W \subseteq T^{\Sigma_t}$.*

We call a variable $x \in \mathrm{dom}(W)$ native to W *if either $x \in \mathrm{dom}(T)$ or x is introduced by a non-full tgd that generated only facts in W, i.e., there exists an $s \in \{1, \ldots, n\}$, s.t. in the s-th chase step, a non-full tgd $\phi(\vec{x}) \to \psi(\vec{x}, \vec{y})$ fires and the following conditions are fulfilled: (1) the tgd fires with assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$, where $\vec{z}$ consists of fresh variables; (2) all facts in $\psi(\vec{a}, \vec{z})$ are in W, and (3) x is among the variables $\vec{z}$.*

Lemma 3.4. *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. Suppose that T^{Σ_t} is obtained from T by a chase sequence of length n. Finally, let W be an instance, s.t. all facts of T are in W (i.e. W contains facts with the same id's) and $W \subseteq T^{\Sigma_t}$. Moreover, W fulfills the following properties:*

1. *Every fact in W was either already present in T or it was introduced by the application of a non-full tgd.*
2. *W is closed under parents and siblings (over facts).*

Then there exists a variable renaming ρ on T^{Σ_t}, s.t. all variables in the instance $\rho(W) \subseteq \rho(T^{\Sigma_t})$ are native to $\rho(W)$. Moreover, $\rho(T^{\Sigma_t})$ can be computed by a

chase sequence in such a way that, throughout the chase procedure, the facts in $\rho(W)$ (i.e., their id is in $\rho(W)$ and hence in W; the values may later change due to the application of egds) contain only variables which are native to $\rho(W)$ (i.e., the condition that all variables in $\rho(W)$ are native to $\rho(W)$ holds for all intermediate steps during the chase and not just at the end of the chase).

Proof. For $0 \leq s \leq n$, let T_s denote the result after step s of the chase and let $\Delta = (\delta_1, \ldots, \delta_n)$ denote the sequence of dependencies that is applied in order to derive T^{Σ_t} from T. Below, we modify the chase with the same dependencies $\delta_1, \ldots, \delta_n$ in such a way that the antecedent of each δ_s is mapped to the same facts as in the original chase sequence Δ, but variable replacements enforced by egds are possibly applied in the opposite direction, i.e., if, for any $s \in \{1, \ldots, n\}$, δ_s is an egd that leads to the replacement of all occurrences of some variable x_i in T_{s-1} by another variable x_j, then the only modification allowed will be to replace all occurrences of x_j by the variable x_i instead. Let T'_s denote the result after step s of this modified chase. Then the canonical universal instance T'_n can obviously be obtained via a variable renaming from T^{Σ_t}.

For every $s \in \{0, \ldots, n\}$, let $W'_s \subseteq T'_s$ denote the set of those facts in W which are contained in T'_s (note that the concrete attribute values of the facts in W'_s may differ from the values in W due to the variable renaming which distinguishes T'_s from T_s; but the id's of these facts are not affected). We show that there exists a sequence of instances $T'_0 = T, T'_1, \ldots, T'_n$ obtained by a chase which modifies the "original" chase leading to T^{Σ_t} in the above described way, s.t. for every $s \in \{0, \ldots, n\}$, every variable in W'_s is native to W'_s. This claim is proved by induction on s.

[induction begin.] By definition, $W'_0 = W_0 = T_0 = T$. Clearly, all variables in W'_0 are in T and are, therefore, native to W'_0 by definition.

[induction step.] 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, or
4. a non-full tgd, introducing facts present in W.

Note that these are indeed all cases that can occur. In particular, since W is closed under siblings, it cannot happen that a part of the facts introduced by a non-full tgd is in W while another part is not. The proof proceeds by a case distinction over these four cases:

Case 1. T_s is obtained from T_{s-1} via the egd $\phi(\vec{x}) \rightarrow x_i = x_j$, where $i, j \leq |\vec{x}|$ 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.

T'_{s-1} contains the same facts (i.e., with the same id's) as T_{s-1} but possibly with attribute values changed by the variable renaming from T_{s-1} to T'_{s-1}. Let $\vec{a}\,'$ denote the instantiation of $\vec{x}$ which maps each position in $\phi(\vec{x})$ to the same position in T'_{s-1} as the instantiation $\vec{a}$ does when $\phi(\vec{x})$ is mapped to T_{s-1}. Then

this egd application enforces in T'_{s-1} the equality of a'_i and a'_j, where a'_i is a variable.

First consider the case that both a'_i and a'_j are variables and that a'_i is native to W'_{s-1} while a'_j is not. In this case, we reverse the sense of variable replacement from the "original" chase, i.e., we produce T'_s by replacing every occurrence of a'_j in T'_{s-1} by a'_i. By the induction hypothesis, all variables in W'_{s-1} are native to W'_{s-1} and, therefore, after the application of the egd in this reversed way, all variables in W'_s are native to W'_s.

It remains to consider the case that one of the following conditions is fulfilled: (1) a'_j is a constant or (2) a'_j is a variable that is native to W'_{s-1}, or (3) a'_i is not native to W'_{s-1}. In all these cases, we apply the egd analogously to the "original" chase, i.e., all occurrences of a'_i in T'_{s-1} are replaced by a'_j. By the induction hypothesis, all variables in W'_{s-1} are native to W'_{s-1} and, therefore, also after the application of the egd in this way, all variables in W'_s are native to W'_s.

Case 2. Suppose that T_s is obtained from T_{s-1} via a full tgd $\phi(\vec{x}) \rightarrow \psi(\vec{x})$, s.t. $T_{s-1} \models \phi(\vec{a})$. As in case 1, there exists an instantiation $\vec{a}\,'$ of $\vec{x}$ which maps each position in $\phi(\vec{x})$ to the same position in T'_{s-1} as the instantiation $\vec{a}$ does when $\phi(\vec{x})$ is mapped to T_{s-1}. Then we produce T'_s from T'_{s-1} by firing the tgd with this instantiation $\vec{a}\,'$. By assumption, W contains no facts that are generated by non-full tgds. Hence, $W'_s = W'_{s-1}$ and, therefore, all variables in W'_s are native to W'_s by the induction hypothesis.

Case 3. Suppose that T_s is obtained from T_{s-1} via a 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. Then we produce T'_s from T'_{s-1} by firing this tgd with the modified instantiation $\vec{a}\,'$ as in case 2. Since all facts in $\psi(\vec{a}\,', \vec{z})$ are outside W, we have $W'_s = W'_{s-1}$ and, therefore, all variables in W'_s are native to W'_s by the induction hypothesis.

Case 4. Suppose that T_s is obtained from T_{s-1} via a 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 in W. Then we produce T'_s from T'_{s-1} by firing this tgd with the modified instantiation $\vec{a}\,'$ as in the cases above. Now all facts in $\psi(\vec{a}\,', \vec{z})$ are new in W'_s compared with W'_{s-1}. By definition, the new variables in $\vec{z}$ are native to W'_s. Thus, it only remains to show that all variables in $\vec{a}\,'$ are native to W'_s.

Let v be an arbitrary variable in $\vec{a}\,'$, i.e., v occurs in some position in $\psi(\vec{a}\,', \vec{z})$. Since W is closed under parents and siblings, 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. Clearly, position o_p was already contained in W'_{s-1}. Hence, by the induction hypothesis, the variable v at this position must be native to W'_{s-1} and, thus, also to to W'_s. □

Theorem 3.1. *(*Lifting*) Let T^{Σ} be a universal solution of a data exchange*

problem obtained by chasing a preuniversal instance T with the set Σ of weakly acyclic tgds and egds. Suppose that B and W are instances with the following properties:

1. $B \models \Sigma$,
2. *all facts of T are in W (i.e. W contains facts with the same id's) and* $W \subseteq T^{\Sigma}$,
3. *every fact in W was either already present in T or it was introduced by the application of a non-full tgd, and*
4. *W is closed under parents 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 $h'\colon T^{\Sigma} \to B$, 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 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 $T_n = T^{\Sigma}$. W.l.o.g., we may assume that all variables in W are native to W and that, throughout the chase procedure, the facts in W (i.e., their id is in W; the values may later change due to the application of egds) contain only variables which are native to W. Indeed, suppose that this were not the case. Then, by Lemma 3.4, there exists a variable renaming ρ on T^{Σ}, s.t. all variables in the instance $\rho(W) \subseteq \rho(T^{\Sigma})$ are native to $\rho(W)$. Moreover, $\rho(T^{\Sigma})$ can be computed by a chase sequence in such a way that, throughout the chase procedure, the facts in $\rho(W)$ contain only variables which are native to $\rho(W)$. We would then construct a homomorphism $h'\colon \rho(T^{\Sigma}) \to B$, s.t. $h' \circ \rho$ is the desired homomorphism from T^{Σ} to B.

We now show that $h\colon W \to B$ can indeed be extended to a homomorphism $h'\colon T^{\Sigma} \to B$, s.t. $\forall x \in dom(h)\colon h(x) = h'(x)$. For every s, we say that a homomorphism $h_s\colon T_s \to B$ is *consistent with h* if for all $x \in \mathrm{dom}(h_s)$, s.t. x is native to W, the equality $h_s(x) = h([x])$ holds. Recall that we write $[x]$ to denote the term to which x is mapped by the chase. Moreover, a variable x is called native to W either if x occurs in some fact in T or x occurs in $\vec{z}$ for some fact $P(\vec{a}, \vec{z})$ that is introduced into W by a non-full tgd. In both cases, $[x]$ is in $\mathrm{dom}(W)$ and, therefore, $h([x])$ is clearly defined.

We claim that for every $s \in \{0, \ldots, n\}$, such a homomorphism h_s consistent with h exists. This claim is proved 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 $x \in \mathrm{dom}(T)$. We first show that h_0 is well-defined. Let $x \in \mathrm{dom}(T)$. Then x occurs in some fact with id i in T. By condition 2 of the theorem, all facts of T are in $W \subseteq T^{\Sigma}$. Thus, $[x]$ occurs in the fact with id i in W and, therefore, $h([x])$ is indeed defined.

Moreover, h_0 is consistent with h. This follows easily from the fact that $x = [x]$ holds for every variable in $\mathrm{dom}(T) \cap \mathrm{dom}(W)$. It remains to show that h_0 is a homomorphism. By condition 2 of the theorem, all facts of T are in W. Hence, for every fact $P(u_1, \ldots, u_k) \in T_0$, we have $P([u_1], \ldots, [u_k]) \in W$ and,

therefore, $P(h(u_1), \ldots, h(u_k)) = P(h([u_1]), \ldots, h([u_k])) \in B$. 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, or
4. a non-full tgd, introducing facts present in W.

Note that these are indeed all cases that can occur. In particular, since W is closed under siblings, it cannot happen that a part of the facts introduced by a non-full tgd is in W while another part is not.

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 $B \models \phi(h_i(\vec{a}))$. This is the well-known fact that conjunctive queries are closed under homomorphisms.

Case 1. T_s is obtained from T_{s-1} via the egd $\phi(\vec{x}) \to x_i = x_j$, where $i, j \leq |\vec{x}|$ 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 $\phi(\vec{x}) \to x_i = x_j$ 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 $\phi(\vec{x}) \to x_i = x_j$ holds in B. But then $h_{s-1}(a_i) = h_{s-1}(a_j)$.

Case 2. A full tgd $\phi(\vec{x}) \to \psi(\vec{x})$ 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 analogous 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 $\phi(\vec{x}) \to \psi(\vec{x}, \vec{y})$ 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 exists a vector $\vec{c}$ of terms in $\mathrm{dom}(B)$, s.t. $B \models \psi(h_{s-1}(\vec{a}), \vec{c})$.

By definition, all terms in $\vec{z}$ are fresh variables (not yet occurring in T_{s-1}). 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 $\psi(h_s(\vec{a}), h_s(\vec{z})) = \psi(h_{s-1}(\vec{a}), \vec{c})$ of the new atoms $\psi(\vec{a}, \vec{z})$ in T_s is indeed in B.

It remains to show that h_s is consistent with h. By the induction hypothesis, h_{s-1} is consistent with h. Note that $\mathrm{dom}(T_s) \setminus \mathrm{dom}(T_{s-1})$ consists precisely of the fresh variables $\vec{z}$. Recall that we are assuming that all variables in W are native to W, i.e., $\mathrm{dom}(W)$ contains no variable in $\vec{z}$. By the induction hypothesis, h_{s-1} is consistent with h. Moreover, since h_{s-1} differs from h_s only on variables $\vec{z}$ outside $\mathrm{dom}(W)$, also h_s is consistent with h.

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 variables. 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 $x \in \mathrm{dom}(h_{s-1})$. 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. This is then also sufficient in order to prove that h_s is consistent with h.

Recall that we are assuming that, throughout the chase procedure, the facts in W (i.e., their id is in W; the values may later change due to the application of egds) contain only variables which are native to W. Hence, all variables in $\vec{a}$ are native to W. Hence, by the induction hypothesis, $h_{s-1}(\vec{a}) = h([\vec{a}])$ holds.

This concludes the induction. But then $h' = h_n$ is the desired homomorphism. 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'. □

Remark. Let us briefly point out the main differences between the proof of Theorem 3.1 and the proof of the lifting theorem (recalled in Theorem 2.5) according to [8]. Obviously, the treatment of egds introduces additional complications, which have to be handled in the above proof (see Case 1 in the induction). However, technically, the most important difference between the two proofs is due to the fact that we define parents and siblings w.r.t. positions and facts rather than w.r.t. variables. In particular, the *closure under parents* is thus used in completely different ways in the two proofs: In [8], the closure under parents (of variables) is used to conclude that if the *variables* $\vec{z}$ introduced by a non-full tgd are contained in W, then the corresponding facts $\psi(\vec{a}, \vec{z})$ are contained in W as well. In contrast, we get the property that all facts in $\psi(\vec{a}, \vec{z})$ are contained in W "for free" (see Case 4 in the proof above) since we also define the notion of *siblings* via facts. On the other hand, we need the *closure under parents* (w.r.t. facts!) in two places in the proof of our lifting theorem, where

this property is not needed in the corresponding place of the proof in [8]: In Case 3 above, where the atoms in $\psi(\vec{a}, \vec{z})$ are outside W, it is by no means trivial that the variables in $\vec{z}$ are never propagated into W later on in the chase. This is guaranteed by Lemma 3.4, which (in Case 4 of its proof) needs the closure under parents w.r.t. facts. The second place in our proof of the lifting theorem, where we need Lemma 3.4 and, therefore, the closure under parents w.r.t. facts, is the very last step in Case 4 above: Only by Lemma 3.4, we are allowed to assume that all variables in $\vec{a}$ are native to W, which enables us to conclude by the induction hypothesis, that $h_{s-1}(\vec{a}) = h([\vec{a}])$ holds. Actually, no analogue of Lemma 3.4 is needed in [8].

Example 3.6. *Consider a schema mapping with the following dependencies: Source-to-target tgd:*

$$\sigma\colon \quad S(x,y) \to P(x,w,y)$$

Target constraints:

$$\tau_1\colon \quad P(x,w,y) \to \exists v \exists r\ R(v,x,y,r)$$

$$\tau_2\colon \quad P(x,w,y) \to \exists z \exists q\ R(x,z,w,q)$$

$$\tau_3\colon \quad R(x,y',y'',q) \to \exists p\ Q(x,q,p)$$

$$\tau_4\colon \quad Q(x,y',y'') \to \exists o\ T(x,o,x)$$

$$\varepsilon_1\colon \quad P(x_1,w_1,y_1) \wedge P(x_2,w_2,y_2) \to w_1 = w_2$$

$$\varepsilon_2\colon \quad R(x,y_1,w_1,z_1) \wedge R(y_2,w_2,x,z_2) \to y_1 = y_2$$

For the source instance $I = \{S(1,1), S(1,2)\}$, a preuniversal instance $J = I^{\Sigma_{st}}$, a canonical universal solution J^{Σ_t} and its core are depicted in Figure 3.

We illustrate how a proper endomorphism h' on J^{Σ_t} can be built, s.t. h' sends q_2 to q_1. Lemma 3.3 says that one can construct an instance W (referred to as $T_{q_1 q_2}$ in Lemma 3.3) satisfying the following conditions: $W \subseteq J^{\Sigma_t}$, W contains the facts in $J = I^{\Sigma_{st}}$ as well as $Origin_{q_1} = R(1, z_1, w_1, q_1)$ and $Origin_{q_2} = R(1, z_1, w_1, q_2)$. Moreover, W is closed under the ancestor and sibling relations over facts. Clearly, $W = \{P(1, w_1, 1), P(1, w_1, 2), R(1, z_1, w_1, q_1), R(1, z_1, w_1, q_2)\}$ is such an instance (no atom in our example has siblings, and the parent relation is shown in Figure 3(b) as thin arrows).

Consider the homomorphism $h\colon W \to J^{\Sigma_t}$, such that $h(q_2) = q_1$ and h is the identity on all elements in $var(W) \setminus \{q_2\}$. We now use Theorem 3.1 to turn h into the desired proper endomorphism h' on J^{Σ_t}. Consider a chase sequence which first enforces all tgds and then the egds. It is not difficult to see that the result of this procedure indeed satisfies all the dependencies. Figure 3(b) shows the target database before the first egd ε_1 has fired, and Figure 3(c) gives the final state of the target database. Note that every variable in W is native in the sense of Definition 3.1. We now show how h' is constructed from h by the steps described in the induction proof of Theorem 3.1.

1. *Start with the preuniversal instance* $J = \{P(1, w_1, 1), P(1, w_2, 2)\}$*, and a homomorphism* h_0*, s.t.* $h_0(w_1) = h(w_1) = w_1$ *and* $h_0(w_2) = h([w_2]) = w_1$*.*
2. *"Replay" the chase step with* τ_1 *firing on* $P(1, w_1, 1)$ *and add the variables* v_1 *and* r_1 *to the domain of* h_0*.* $R(v_1, 1, 1, r_1) \notin W$*, hence* h_0 *is extended to a valid homomorphism* h_1 *according to the Case 3 of the proof of Theorem 3.1. The fact* $R(v_1, 1, 1, r_1)$ *is introduced via the tgd* τ_1 *by matching the antecedent of* τ_1 *with* $P(1, w_1, 1)$*. Note that* $P(1, w_1, 1)$ *is left unchanged by* h_0*. We know that* τ_1 *is satisfied in* J^{Σ_t}*. In particular, if the antecedent of* τ_1 *is matched with* $P(1, w_1, 1)$*, then we can match the conclusion with* $R(z_1, 1, 1, r_1)$*. Hence,* h_1 *extends* h_0 *to* $\{v_1, r_1\}$ *in such a way that* $R(v_1, 1, 1, r_1)$ *is sent to* $R(z_1, 1, 1, r_1)$*, i.e., we set* $h_1(v_1) = z_1$ *and* $h_1(r_1) = r_1$*.*
3. *Likewise, we replay the next three chase steps introducing the remaining tuples in the R-relation, extending* h_1 *to* h_4 *on the additional variables* $\{z_1, q_1, v_2, r_2, z_2, q_2\}$*. We thus apply either Case 3 or Case 4 of the proof of Theorem 3.1: By Case 4, we have* $h_2(z_1) = z_1$ *and* $h_2(q_1) = q_1$*. By Case 3, we have* $h_3(v_2) = v_2$ *and* $h_1(r_2) = r_2$*. Finally, by Case 4, we have* $h_4(z_2) = h([z_2]) = h(z_1) = z_1$ *and* $h_4(q_2) = h([q_2]) = h(q_2) = q_1$*.*
4. *We now extend* h_4 *to* h_8 *via Case 3 of Theorem 3.1 to cover the new variables* $\{p_1, p_2, p_3, p_4\}$ *when the tuples in the Q-table are introduced by the chase: We clearly have* $h_5(p_1) = p_1$*,* $h_6(p_2) = p_2$*, and* $h_7(p_3) = p_3$*. Now* h_8 *is obtained from* h_7 *as follows: The fact* $Q(1, q_2, p_4)$ *is introduced via the tgd* τ_3 *by matching the antecedent of* τ_3 *with* $R(1, z_2, w_2, q_2)$*. Note that* $R(1, z_2, w_2, q_2)$ *is sent to* $R(1, z_1, w_1, q_1) \in J^{\Sigma_t}$ *by* h_7*. Moreover,* τ_3 *is satisfied in* J^{Σ_t}*. In particular, if the antecedent of* τ_3 *is matched with* $R(1, z_1, w_1, q_1)$ *then the conclusion of* τ_3 *is matched with* $Q(1, q_1, p_2)$*. Hence,* $Q(1, q_2, p_4)$ *must be sent to* $Q(1, q_1, p_2)$ *by* h_8*, i.e., we have* $h(p_4) = p_2$*.*
5. *There are three more chase steps with tgds remaining. By Case 3 of Theorem 3.1, we thus extend* h_8 *to* h_{11} *by defining that* h_{11} *is the identity on* o_1, o_2 *and* o_3*.*
6. *Finally, the egds have to be replayed. By Case 1 of Theorem 3.1, this simply means to restrict the domain of* h_{11} *to the domain of* J^{Σ_t}*, i.e., we eliminate the variables* $\{v_1, z_2\}$ *from the domain of* h_{11} *to get the desired endomorphism* h' *on* J^{Σ_t}*. In summary, we have* $h'(q_2) = q_1$ *and* $h'(p_4) = p_2$*, while* h' *is the identity on all other elements in* $\mathrm{dom}(J^{\Sigma_t})$*.*

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}.

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. $Parents(A) \neq \emptyset$ and $Parents(A) \subseteq W$
(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'.

In the procedure EXTEND, we use the following terminology: For a fact A, we write $Parents(A)$ and $Siblings(A)$ to denote the set of all parents respectively all siblings of A. Of course, only for facts generated by the application of a non-full tgd, these sets are non-empty. For a homomorphism f, we write $\mathrm{dom}(f)$ to denote those domain elements for which f is defined. Moreover, if two homomorphisms g_1 and g_2 coincide on $\mathrm{dom}(g_1) \cup \mathrm{dom}(g_2)$, we write $g_1 \cup g_2$ to denote the combination of these two homomorphisms, i.e., $(g_1 \cup g_2)(x)$ is defined as $g_1(x)$ if $x \in \mathrm{dom}(g_1)$ and $g_2(x)$ otherwise.

The idea of procedure EXTEND is as follows: Our goal is to construct a homomorphism h' which is is defined on the entire domain $\mathrm{dom}(T^{\Sigma_t})$ and which coincides with h on $\mathrm{dom}(h) \cap \mathrm{dom}(h')$. Initially, $h' = h$. In the loop at lines (2) – (8), we try to extend h' to further *facts* in T^{Σ_t} and, hence, to further domain elements in T^{Σ_t}. The facts on which h' is already defined are accumulated in the set W. In this extension of h to h', we fully concentrate on facts which have been introduced by non-full tgds (since only these facts contain new variables). Now consider the facts that have been introduced by a non-full tgd $\phi(\vec{x}) \to \psi(\vec{x}, \vec{y})$ with assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$. Moreover, suppose that some fact $A \in \psi(\vec{a}, \vec{z})$ has not yet been assigned a function value by the homomorphism h'. By step (4) in the algorithm, we always extend h' to all siblings of a new fact. Hence, if h' is not yet defined on A then it is not defined on its siblings either. Following Case 3 in the proof of Theorem 3.1, we have to know $h'([\vec{a}])$ in order to extend h' to $\vec{z}$. Hence, in line (2) of the algorithm, we choose A in such a way that its parents are already contained in the current set W. The easiest way to achieve this is to follow the chase sequence by which T^{Σ_t} was produced. Then the facts $S = \{A\} \cup Siblings(A)$ processed by each iteration of

the while-loop are simply the facts introduced by the next non-full tgd in this chase sequence. At the end of each iteration of the while-loop, h' and W are extended according to the homomorphism g, which is determined as in Case 3 in the proof of Theorem 3.1.

The correctness of the procedure EXTEND is the subject of the following theorem.

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 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. $\forall x \in \mathrm{dom}(W_j)\colon h'(x) = h_j(x)$ holds:

[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 $h_j\colon W_j \to B$ fulfills the assumptions on W in Theorem 3.1. Hence, h_j can be extended to a homomorphism $h\colon T^{\Sigma_t} \to B$, s.t. $\forall x \in \mathrm{dom}(h_j)\colon h'(x) = h_j(x)$. Then it is of course also possible to extend h_j to the homomorphism $h_{j+1}\colon W_{j+1} \to B$ 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.

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. □

Example 3.7. *In Example 3.6, the construction of a proper endomorphism on J^{Σ_t} (Figure 3) via Theorem 3.1 was described: to this end, the whole chase sequence was replayed starting with the preuniversal instance $J = I^{\Sigma_{st}}$. In particular, it was necessary to choose the images for the variables w_2, v_1 and z_2, although later on they were eliminated by egds. The procedure* EXTEND *allows us to avoid this unnecessary activity, and consider only the elements in the domain of J^{Σ_t}. Moreover, all facts in W (and not just the ones in J) are covered by the homomorphism h' in the procedure* EXTEND *right from the beginning and do not have to be reconsidered when the chase is replayed.*

We start with h' defined on all facts in W. We thus have h' with $h'(q_2) = q_1$ and h' is the identity on $\{w_1, z_1, q_2\}$. The first chase step introduces the fact $R(z_1, 1, 1, r_1)$ (recall that we immediately take the values that these facts have at the end of the chase) and we set $h'(r_1) = r_1$. The introduction of the fact $R(v_2, 1, 2, r_2)$ leads to the extension of h' with $h'(v_2) = v_2$ and $h'(r_2) = r_2$. Clearly, the second and fourth fact in the R-table are ignored by the procedure EXTEND*, since these facts are in W.*

By considering all tuples in the Q-table and T-table of J^{Σ_t}, we define h' as the identity on all variables in $\{p_1, p_2, p_3, o_1, o_2, o_3\}$. As far as the variable p_4 is

concerned, we proceed analogously to the extension of h_7 to h_8 in Example 3.6 and set $h'(p_4) = p_2$.

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 [6].

Definition 3.2. *Let K be an instance whose elements are constants and variables. 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 [6] reveals that it remains valid when tgds are added.

Lemma 3.5. *(*RIGIDITY*) Assume a data exchange setting where Σ_{st} is a set of tgds and Σ_t is a set of egds* and weakly-acyclic tgds*. Let J be the canonical preuniversal instance and let $J' = J^{\Sigma_t}$ be the canonical universal instance. Let x and y be variables 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. [6]) 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 variables 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 $R(u_1, \ldots, u_n)$ be a fact in J (and, therefore, a conjunct in ϕ). Then the fact $R([u_1], \ldots, [u_n])$ is in J' by the definition of $[\cdot]$. Moreover, it can be shown (by exactly the same arguments as in [6]), that $V(u_i) = h([u_i])$ holds for every element $u_i \in \mathrm{dom}(J)$. Hence, $R(V(u_1), \ldots, V(u_n)) = R(h([u_1]), \ldots, h([u_n]))$. 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 $h([x]) = [y] = [x]$. □

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

Definition 3.3. *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 st-tgds 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 depending 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 $B_1, \ldots, B_n$ denote the non-rigid blocks of T_{xy}. Moreover, for every $i \in \{1, \ldots, n\}$, 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. □

Example 3.8. *Let us revisit Example 3.6. We start building a proper endomorphism h' on J^{Σ_t} by constructing a homomorphism $h\colon W \to J^{\Sigma_t}$ with $W = \{P(1, w_1, 1), P(1, w_1, 2), R(1, z_1, q_1), R(1, z_1, q_2)\}$. The variables in W fall into two blocks, namely $\{w_1\}$ and $\{z_1, q_1, q_2\}$.*

Now consider the preuniversal instance J, which has the following blocks: $\{w_1\}$, $\{w_2\}$, $\{v_1, r_1, p_1, o_1\}$, $\{z_1, q_1, p_2\}$, $\{v_2, r_2, p_3, o_3\}$, $\{z_2, q_2, p_4\}$, $\{o_2\}$. The egd ε_1 enforces the equality $w_1 = w_2$; the egd ε_2 enforces the equalities $z_1 = v_1 = z_2$. In J, w_1 and w_2 are in different blocks. Likewise, z_1, v_1, and z_2 are all in different blocks. Hence, the variables w_1 and z_1 in W are rigid. Thus, we only search for homomorphisms $h\colon W \to J^{\Sigma_t}$ with $h(w_1) = w_1$ and $h(z_1) = z_1$.

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

Input: Source ground instance S

Output: Core of a universal solution for S

(1) Chase (S,$\emptyset$) 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 by calling the procedure Extend;
(8) | | Transform h' into a retraction r;
(9) | | Set $U := r(U)$;
(10) **return** U.

The non-rigid blocks of W are $\{q_1\}$ and $\{q_2\}$. The search for a homomorphism is thus reduced to finding the image of q_1 and of q_2.

Actually, if we consider all of J^{Σ_t} (rather than just W), then the blocks $\{v_1, r_1, p_1, o_1\}$, $\{z_1, q_1, p_2\}$, and $\{z_2, q_2, p_4\}$ of J collapse to a single block $\{r_1, p_1, o_1, z_1, q_1, p_2, q_2, p_4\}$ (note that v_1 and z_2 have disappeared due to the egd-applications). This block is considerably bigger than the original ones in the preuniversal instance. However, since z_1 is a rigid variable, this block can be split into the non-rigid blocks $\{r_1, p_1, o_1\}$, $\{q_1, p_2\}$, and $\{q_2, p_4\}$, which even have smaller size than the original blocks.

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 3.5 and Theorem 3.3, and step (7) is based on Lemma 3.4 as well as 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 st-tgds Σ_{st} and target dependencies 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

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 4(a). 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 st-tgds and target dependencies 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(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 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`,'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. Step 4 of FindCore^E *might, for instance, yield 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 $\{T_2 \leftarrow T_1,\ C_1 \leftarrow C_2\}$*. In order to extend* $h\colon T_{N,'\mathsf{Yves}'} \to J$ *with* $\mathrm{var}(T_{N,'\mathsf{Yves}'}) = \{N, C_1, T_2\}$ *to an endomorphism* h' *on* J*, we have to find images of one variable after the other in* $J \setminus T_{N,'\mathsf{Yves}'}$*. 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 (i) a variable shared between f_1 in the premise and f_2 in the conclusion of the same tgd, (ii) a variable shared by f_1 and f_2 in the conclusion of a tgd, or (ii) 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`}.

Partitions not only reduce the search space for endomorphism computation, but also allow to optimize the storage schema for evaluation of joins. Since, under arbitrary schema mapping, both nulls and constants can occur in every column including the key one, neither column can be defined as unique, and each join condition `col1 = col2` must be rewritten as `col1 = col2 OR col1_var = col2_var`, which considerably hinders query performance. To overcome this, during the chase we compute the *domain* of each partition, and store the domain identifier of each database value in the auxiliary columns in the target tables. Now, if `col1` and `col2` belong to the same partition, the join condition can be rewritten as `col1_domid = col2_domid`, where prefix "`_domid`" marks such auxiliary columns.

5. Experiments and Discussion

So far, neither core computation nor labeled nulls are featured in any DBMS resp., data integration tool, 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. By adding redundant target tgds and, failing to specify necessary egds, we were able to vary the amount of minimization effort for the core computation algorithm from mere checking the optimality of the instance to removing approximately a half of the tuples generated by the chase.

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. The runtimes reported in this section 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.

Test scenario. The tests were carried out with the following scenario, based on the TPC-H database schema [13] (see Figure 5).[3] The schema depicted in Figure 5 is chosen as the *target schema*. Hence, the foreign key constraints (depicted as arrows in Figure 5) and key constraints (e.g., on columns `n_nationkey`, `p_partkey` and others) give rise to the following target tgds and egds:

[3]Note that no database size requirements of the TPC-H test are met here. The motivation for our choice is solely to use a well-known database schema for illustration. Furthermore, we omitted the "Comment" fields present in each table in the original schema [13].

- $LineItem(OrdKey, \dots) \rightarrow Order(OrdKey, \dots)$,
- $LineItem(\dots, SuppKey, \dots) \rightarrow Supplier(SuppKey, \dots)$, etc.: similar tgds are used for each foreign key in the schema.
- $Nation(Key, N_1, R_1) \wedge Nation(Key, N_2, R_2) \rightarrow N_1 = N_2 \wedge R_1 = R_2$ — primary key constraint on the `Nation` relation; other PKs are defined in the same way.

Here, by three dots inside an atom, we abbreviate a list of variables that occur only once in a formula thus being irrelevant for evaluating the precondition of a dependency, or – in case of conclusion variables – serving as placeholders for distinct fresh nulls introduced in the course of the chase.

Now, for the source schema and the source-to-target dependencies, we consider the following scenario: Suppose that the database was accidentally dropped and needs to be recovered using a number of sources, each containing some part of the original data, namely:

— Sales database containing the extracts $\texttt{Order}_{\texttt{SALE}}$ and $\texttt{Customer}_{\texttt{SALE}}$ from the original tables `Order` and `Customer`, and the $\texttt{LineItem}_{\texttt{SALE}}$ table, extracted from `LineItem`, with the difference that the fields `l_partkey` and `l_suppkey` are substituted by the pair `l_partname`, `l_suppname` containing the names of a part resp. a supplier. The following st-tgd brings the data from Sales back into the original schema:

$$
\begin{aligned}
&Order_{SALE}(OK, CK, S, TPr, Dt, OPri, Cl, SPri) \\
&\quad \wedge Customer_{SALE}(CK, CN, Addr, Nat, Ph, ActB, Seg) \\
&\quad \wedge LineItem_{SALE}(OK, PN, SN, Num, Q, EP, Dc, Tax, RFg, LS, SDt, CDt, RDt, ShI, ShM) \\
&\quad \rightarrow Order(OK, CK, S, TPr, Dt, OPri, Cl, SPri) \\
&\quad \wedge Customer(CK, CN, Addr, Nat, Ph, ActB, Seg) \\
&\quad \wedge Lineitem(OK, \underline{PK}, \underline{SK}, Num, Q, EP, Dc, Tax, RFg, LS, SDt, CDt, RDt, ShI, ShM) \\
&\quad \wedge PartSupp(\underline{PK}, \underline{SK}, \dots) \wedge\ Part(\underline{PK}, PN, \dots) \wedge\ Supplier(\underline{SK}, SN, \dots)
\end{aligned}
$$

Here, the existentially-quantified variables are either shown underlined (e.g. the keys of the `Supplier` and `Part` tables have to be invented anew) or skipped (in case they occur only once in the formula).

— The Supplies database contains the suppliers, parts, and ordered items in the tables $Supplier_{SUP}$, $Part_{SUP}$, $PartSupp_{SUP}$, and $LineItem_{SUP}$. The following st-tgd allows us to reimport the data from this schema:

$$
\begin{aligned}
&Supplier_{SUP}(SK, CN, Addr, Nat, Ph, ActB) \\
&\quad \wedge Part_{SUP}(PK, PN, MfG, B, Typ, Sz, Cnt, RP) \\
&\quad \wedge PartSupp_{SUP}(PK, SK, AQty, SC) \\
&\quad \wedge LineItem_{SUP}(PK, SK, Num, Q, EP, Dc, Tax, RFg, LS, SDt, CDt, RDt, ShI, ShM) \\
&\quad \rightarrow Supplier(CK, CN, Addr, Nat, Ph, ActB, Seg) \\
&\quad \wedge Part(PK, PN, MfG, B, Typ, Sz, Cnt, RP) \\
&\quad \wedge PartSupp(PK, SK, AQty, SC) \\
&\quad \wedge Lineitem(\underline{OK}, PK, SK, Num, Q, EP, Dc, Tax, RFg, LS, SDt, CDt, RDt, ShI, ShM)
\end{aligned}
$$

Note that the above source-to-target tgds are not normalized (cf. the tgd in the Example 3.3) for the sake of brevity of notation: e.g., normalizing the second dependency leads to four tgds with a single atom in the conclusion each.

— Finally, suppose that the source schema contains yet another database, called SAMPLE, such that SAMPLE conforms to the original full TPC-H schema, but contains only an extract of the original data. One immediately makes use of it by copying its contents into the database being recovered, applying the source-to-target tgds of the form $R_{SAMPLE}(\vec{x}) \rightarrow R(\vec{x})$ to each relation R in the schema.

Due to the data in SAMPLE, some tuples from the SALES or SUPPLIES databases may become redundant. We have exploited this observation for our tests in that it allowed us to control the rate of redundancy in the target database by properly populating the source databases. Additionally, we also experimented with slight modifications of the target tgds presented above in order produce further redundancy in the target database. For instance, turning an inclusion dependency on the `Nation` table:

$$Nation(NK, RK, NName) \rightarrow Region(RK, RName)$$

into

$$Nation(NK, RK, NName) \rightarrow Region(RK, RName), Region(RK_1, RName)$$

and so forth. Such modifications of tgds produce logically equivalent tgds, which generate further redundant target facts that can be eliminated by the core computation.

Performance of core computation. In a setting where the canonical solution had about 50% more nulls than the core, our system managed to compute the core for a target DB with about 6,000 labeled nulls in almost 180 min (solid curve with square symbols on Figure 6,a). In contrast, the core of an instance with about 20,000 nulls was computed in similar time (solid curve with triangles) when only 10% of the variables were redundant.

We have also implemented the FINDCORE algorithm of [8] in order to compare its performance with our algorithm. The left-most curve in Figure 6(a) corresponds to a run of FINDCORE on an instance with approximately 10% of variables being redundant. The runtime is comparable to (in fact, worse than) the most problematic case with over 50% redundancy for the FINDCORE^E algorithm. Actually, this is not surprising: One of the principal advantages of FINDCORE^E 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 5.1. *Let $J = \{R(x, y), P(y, x)\}$ be a preuniversal instance, and a single egd $R(z, v), P(v, z) \rightarrow z = v$ 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]:*

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

where E is an auxiliary predicate representing equality. Chasing J with $\bar{\Sigma}_t$ (in a nice order), yields the instance $J^{\bar{\Sigma}_t} = \{R(x,y), R(x,x), R(y,x), R(y,y), P(y,x), P(y,y), P(x,y), P(x,x), E(x,x), E(x,y), E(y,x), E(y,y)\}$[4]. 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 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(b): The solid line shows the number of redundant nulls remaining after i iterations (i.e., i nested endomorphisms) while the dashed 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.

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 dashed curves in Figure 6(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.

Effect on query answering. We also carried out tests to shed light on the negative effect of redundant tuples in the target database on the performance of query answering. Consider, for instance, the following query, retrieving the links between customers and suppliers residing in the same country and processing similar parts.

```
SELECT DISTINCT c_name, s_name, p1.p_partname
FROM Customer
        JOIN LineItem ON l_orderkey = o_orderkey
        JOIN PartSupp ps1 ON l_partkey = ps1.ps_partkey
                          AND l_suppkey = ps1.ps_suppkey
        JOIN Part p1 ON ps1.ps_partkey = p1.p_partkey
        JOIN Part p2 ON p1.p_partname = p2.p_partname
        JOIN PartSupp ps2 ON p2.p_partkey = ps2.ps_partkey
        JOIN Supplier ON ps2.ps_suppkey = s_suppkey
WHERE s_nationkey = c_nationkey
```

[4]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.

Note that, of course, such a query could not be run "as is" on our database simulating labeled nulls: the join conditions must be defined either disjunctively on the pair of columns representing constants and variables, or on the domain identifiers columns (see the end of Section 4). To keep the notation simple, we opted to avoid these technical details here.

The chart in Figure 7 juxtaposes the execution time of one iteration of FINDCORE^E and the performance gain for the above query, when one such iteration of FINDCORE^E has been carried out. In these tests, the core of the target instance was kept fixed, while ever increasing portions of redundant tuples were inserted at every stage of the experiment. First, the query was run against the database with redundant tuples, after which a single iteration of the FINDCORE^E was executed, and the query was evaluated again on the resulting, shrunk database. Under "performance gain" the difference of the query evaluation times before and after core approximation is understood.

This example demonstrates a situation where the effect of core approximation is significant even for a sinlge execution of a conjunctive query. Moreover, one has to keep in mind that the core approximation has to be carried out only once while the performance gain in query answering is achieved every time a (sufficiently complex) query has to be executed by the database system.

Lessons learned. Our experiments have demonstrated that redundancy elimination by core computation (or at least by an approximation to the core) can have a significant effect on query answering. As far as the performance of core computation is concerned, 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 as witnessed by our experiments (see also Example 5.1 for an illustration of the negative effect of simulating the egds). 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.

Acknowledgement. This work was supported by the Vienna Science and Technology Fund (WWTF), project ICT08-032. Additionally, V. Savenkov receives a scholarship from the European program "Erasmus Mundus External Cooperation Window". We are also very grateful to the anonymous referees as well as to Georg Gottlob for their valuable comments on previous versions of this article.

References

[1] F. Afrati and Ph. G. Kolaitis. Answering aggregate queries in data exchange. *Proc. PODS'08*, pages 129–138. ACM, 2008

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

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

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

[5] 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.

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

[7] 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.

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

[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.

[13] TPC Benchmark H, Standard Specification, Revision 2.8.0. Available at `http://tpc.org/tpch/spec/tpch2.8.0.pdf`, as of August 2009.

List of figures

- Figure 1: Dependency graph.
- Figure 2: Positions of the instance J^{Σ} (foreign positions are dashed) (a) and the dependency graph of Σ (b).
- Figure 3: (a) Dependency graph (special edges are dashed), (b) the preuniversal instance and the result of tgds chase, and (c) the universal solution and the core for the Example 3.6. Thin arrows show parent tuples.
- Figure 4: Overview of the implementation (a) and modelling labeled nulls (b).
- Figure 5: The TPC-H based schema used for experiments, adapted from [13].
- Figure 6: Performance (a) and the progress (b) of core computation.
- Figure 7: Performance gain of the conjunctive query evaluation vs. core approximation time