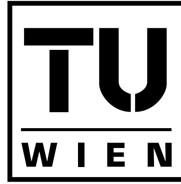
Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

. . . .

Reinhard Pichler



Technische Universität Wien

Vienna University of Technology

M.Sc. Arbeit

Implementing Core Computation for Data Exchange

ausgeführt am

Institut für Informationssysteme
Abteilung für Datenbanken und Artificial Intelligence
der Technischen Universität Wien

unter der Anleitung von

Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler

durch

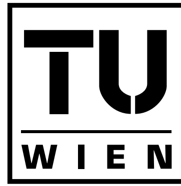
Vadim Savenkov

Wien, 11. Oktober 2007

. . . .

Vadim Savenkov

. . . .
Reinhard Pichler



TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MASTER THESIS

Implementing Core Computation for Data Exchange

carried out at the

Institute of Information Systems
Database and Artificial Intelligence Group
of the Vienna University of Technology

under the instruction of

Univ.Prof. Mag.rer.nat. Dr.techn. Reihard Pichler

by

Vadim Savenkov

Vienna, October 11, 2007 . . . .
Vadim Savenkov

Kurzfassung

Datenaustausch (engl. Data exchange) beschäftigt sich mit dem Datentransfer zwischen Datenbanken mit unterschiedlichen Schemata, wobei die Quelldaten durch die Zieldaten so genau wie möglich dargestellt werden sollten. Zu einer gegebenen Quellinstanz gibt es normalerweise viele Lösungen (d.h. Zielinstanzen) für das Datenaustauschproblem. Fagin et al. [FKP03] gaben überzeugende Argumente, dass in vielen Fällen der "Kern" (engl. Core) als Zieldatenbank gewählt werden sollte. Der allgemeinste Algorithmus, um den Kern eines Datenaustauschproblems in polynomieller Zeit zu berechnen, ist der FindCore Algorithmus von Gottlob und Nash [GN06]. Er lässt sich auf Datenaustauschprobleme anwenden, die folgende Abhängigkeiten verwenden: zwischen Quell- und Zielschema bestehen tupel-erzeugende Abhängigkeiten, und auf dem Zielschema bestehen sowohl schwach azyklische tupel-erzeugende als auch gleichheits-erzeugende Abhängigkeiten.

Ein wesentliches Merkmal des FindCore Algorithmus ist die Simulation der gleichheits-erzeugenden Abhängigkeiten durch tupel-erzeugende Abhängigkeiten. In dieser Diplomarbeit wird eine verbesserte Version dieses Algorithmus vorgestellt, die anstelle der Simulation mittels tupel-erzeugenden Abhängigkeiten die gleichheits-erzeugenden Abhängigkeiten direkt behandelt. Außerdem wurde im Rahmen dieser Arbeit der veresserte Algorithmus implementiert. Die Diplomarbeit enthält auch eine Beschreibung der Implementierung sowie erste experimentelle Resultate.

Abstract

Data exchange is concerned with the transfer of data between databases with different schemas, whereby the source data should be reflected as accurately as possible by the target data. Given a source instance, there may be many solutions (i.e., target instances) to the data exchange problem. The most compact one among the most general (universal) solutions is called a core. Fagin et al. [FKP03] gave convincing arguments that, in many cases, the core should be the database to be materialized. The most general to-date polynomial time algorithm for core computation is the FindCore algorithm developed by Gottlob and Nash [GN06]. It tackles data exchange problems where dependencies between source and target schemas are arbitrary tuple generating dependencies (TGDs), and target constraints consist of equality generating dependencies (EGDs) and weakly-acyclic TGDs.
One important feature of the FindCore algorithm is that the EGDs are simulated by TGDs. In this paper, we present an enhanced version of this algorithm, which includes EGDs directly in the target chase and avoids the simulation with TGDs. We have developed a prototype implementation of the enhanced algorithm. The thesis contains its description, as well as a summary of the first experimental results.

Category and Subject Descriptors:
H.2.5 [Heterogenous Databases]: Data Translation; H.2.4 [Systems]: Relational databases; H.2.4[Systems]: Rule-based databases; H.2.4 [Systems]: Query Processing; D.2.12 [Interoperability]: Data mapping.

General terms:
Algorithms, Theory, Database

Additional Key Words and Phrases:
Chase, core, complexity, conjunctive queries, constraints, data exchange, data integration, dependencies, query evaluation, tractability, universal solutions

Dedication

I dedicate this work to my parents, Anatoly Savenkov and Lidia Levchenkova, to whom I owe any success I've ever had in my life, and to my dear wife Aleksandra whose immense love and patience I constantly feel.

Acknowledgments

This thesis would be impossible without sustained support, advice and encouragement of my supervisor, Prof. Reinhard Pichler. I am extremely grateful to him for all the time and effort he invested in this work. I would also like to thank Prof. Georg Gottlob for sharing his insightful ideas both in personal conversation and through his published research.

My Master's studies were funded by the generous Erasmus Mundus Scholarship, which I was awarded as a member of European Master's Program in Computational Logic. I would like to thank the Head of the Program, Prof. Steffen Hölldobler, for giving me this opportunity, and the Program coordinator in Vienna, Prof. Alexander Leitsch for kind support in all academic and organizational issues.

Finally, I am grateful to all my friends and colleagues who made these last two years abroad a really enjoyable and enriching time for me.

Contents

1 Introduction **1**
1.1 Summary of results . . . 2
1.2 Organization . . . 3

2 Preliminaries **4**
2.1 The data exchange problem . . . 4
2.2 Relations between instances . . . 5
2.3 Universal solutions . . . 6
2.4 Chase and canonical universal solutions . . . 7
2.4.1 Additional definitions . . . 8
2.5 Cores . . . 8

3 Core computation **10**
3.1 Development of the core computation algorithms . . . 10
3.1.1 Computing a core in absence of target dependencies . . . 11
3.1.2 Computing a core in presence of target EGDs . . . 13
3.1.3 Adding restricted target TGDs . . . 14
3.2 The FindCore algorithm . . . 15
3.2.1 Prerequisites . . . 15
3.2.2 Basic ideas . . . 15
3.2.3 Tackling weakly acyclic target TGDs . . . 18
3.3 Discussion of the FindCore algorithm . . . 21

4 The enhanced core computation algorithm **22**
4.1 Preliminary definitions . . . 23
4.1.1 Introduction of an id . . . 23
4.1.2 Source position and origin . . . 23
4.1.3 Normalization of TGDs . . . 25
4.1.4 The parent and sibling relations over facts . . . 26
4.2 Enhancing the algorithm . . . 26
4.3 Discussion . . . 32

5 Algorithm Implementation **34**
5.1 General system architecture . . . 34
5.2 Design decisions . . . 35
5.2.1 Labeled nulls support . . . 35
5.2.2 Blocks and disjoint sets . . . 36
5.2.3 Unlabeled nulls in the source instance . . . 38
5.2.4 Partitions . . . 38
5.2.5 Chase . . . 39

5.2.6 Computing non-rigid blocks . . . 41
5.2.7 Tracking variable families . . . 41
5.2.8 Computing a characteristic homomorphism . . . 43
5.2.9 Homomorphism extension . . . 45
5.2.10 Choosing the optimal mapping . . . 45
5.2.11 Obtaining the retraction . . . 46
5.3 Implementation . . . 46
5.3.1 Database manipulation . . . 46
5.3.2 Configuration . . . 47
5.3.3 Main system classes . . . 50

6 System tests and evaluation **52**
6.1 Redundant tuples . . . 53
6.1.1 3-level target TGDs . . . 53
6.1.2 4-level target TGDs . . . 54
6.1.3 3-level target TGDs and an EGD . . . 57
6.2 Normalization and denormalization . . . 58
6.3 Introducing a new relation with nulls . . . 59
6.4 Summary and evaluation of results . . . 63

7 Conclusions **64**
7.1 Future work . . . 64

References **66**

Chapter 1

Introduction

Due to the rapid progress of business process automatization, companies and organizations are today, more than ever before, faced with the problem of transferring data between and answering queries over heterogeneous systems. In particular, there is a need of transferring data between databases with different schemas. Consequently, two closely-related areas, both dealing with schema mappings, have emerged in database research: *data integration* and *data exchange*. In *data integration* (see e.g. [HRO06; Kol05; Len02]), the target (global) instance serves mainly for encapsulation of the source schemas. The queries against the target database are being rewritten in terms of the encompassed source schemas and passed to the respective databases. Then the data integration system combines the results of the dispatched queries in a single result set, which is returned to the client. Since data are distributed among the source databases, the global system is only a virtual database: an interface, translating the queries over the virtual global schema into queries over local schemas.

Conversely, *data exchange* is concerned with physical materialization of the source data in the target storage, according to the *source-to-target dependencies* (STDs). More specifically, STDs are *tuple generating dependencies* (TGD), which are implications with a conjunction of atoms both in the premise and in the conclusion. Notably, source-to-target implicational dependencies generalize global-as-view (GAV) mappings and local-as view (LAV) mappings and are sometimes referred to as GLAV mappings [FKMP03; Len02].

In data exchange, the target database is also allowed to impose additional data constraints called *target dependencies* (TDs), and expressed by the TGDs, and additionally *equality generating dependencies* (EGDs), which are implications with a conjunction of atoms in the premise, and equations between premise terms in the conclusion. TGDs and EGDs can express the most important dependencies used in database design like functional dependencies, inclusion dependencies, multi-valued dependencies, and join dependencies [Fag82].

The transferring of data between two databases is a routine task solved by a host of tools available on the market. However, the problem of *semantics* of data exchange only recently received the due attention. Since a materialized target database reflects the source data, what is the meaning of a target query in terms of a source schema? Of an answer to such query? Usually, there are many possible materializations of the source data in the target storage that satisfy source-to-target dependencies. Given an answer to a target query, does it reflect the source data and the dependencies, or just some particular materialization? These and many further questions were explored in [FKMP03; FKP03; Lib06].

Particularly, in [FKMP03] the *data exchange problem* was defined, and important properties of solutions to such problems were identified. Informally, the source and target

schemas, as well as STDs and TDs, are considered fixed, and are called the *data exchange setting*. The corresponding *data exchange problem* consists of finding such a target instance (a materialization of the source data in the target schema), that, together with a given source instance, satisfies all the dependencies. It was shown [FKMP03; FKP03] that the good solution must be as general as possible: a notion of *universal solution* was introduced to capture this. As shown by Fagin et al., a universal solution for a data exchange problem can be obtained with the *chase* procedure [BV84], which is essentially a systematic modification of the target instance via the STDs and TDs until all dependencies are satisfied, or impossibility of the satisfaction is witnessed. The result of the chase is called the *canonical universal solution*.

Since (canonical) universal solutions are generally not unique, the most compact of them are preferred. It turned out [FKP03], that the concept of *core*, well known in graph theory, perfectly captures such a minimality property. Moreover, Fagin et al. argued that cores of the universal solutions have advantageous properties w.r.t. semantics of data exchange, and therefore must be preferred for materialization.

Research in the area of data integration is already resulting in practical applications. The best example is perhaps IBM's research project "Clio" [HHH$^+$05]. For data exchange, the vast majority of *Extract-Transform-Load* (ETL for short) systems available on the market follow a procedural approach to specification of data transfers. Consequently, it is by no means easy to relate the transformation procedures featuring most ETL systems, to the data exchange semantics. What kind of solution do they compute? Furthermore, target constraints are often not supported by such tools, and not even research prototypes are available for core computation. A major goal of our work is to initiate the development of such a tool.

In a series of papers ([FKP03; Got05; GN06]) the complexity of core computation was investigated. The most general to-date polynomial time algorithm for core computation is the FindCore algorithm developed by Gottlob and Nash [GN06]. It tackles data exchange problems where target dependencies can include EGDs and *weakly-acyclic* TGDs (a precise definition of weak acyclicity is given in Chapter 2).

One of the specifics of FindCore is that EGDs are simulated by TGDs, by introducing additional target relation for storing equal terms, and adding the number of additional TGDs. As a consequence, the core computation becomes an integral part of finding any solution to the data exchange problem. In fact, there are data exchange semantics [Lib06] that favor the materialization of canonical universal solutions rather than cores. Hence, the core computation should be treated as an optional service of a data exchange framework and, therefore, integrating it into the process of finding any solution is *conceptually problematical*. Moreover, as we shall point out in the Section 4.3, the simulation of EGDs by TGDs is also *computationally problematical*.

1.1 Summary of 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 EGDs are treated directly in the target chase. This allows us to obtain the canonical universal solution first and to minimize it to the core only if this is requested. (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.

1.2 Organization

The thesis is organized as follows. In Chapter 2, we formalize some basic notions of data exchange, then Chapter 3 presents the existing core computation algorithms and in particular, FindCore [GN06]. In Chapter 4 we define and justify FindCore^E, our version of FindCore which does not need to simulate EGDs with TGDs. The comparison of the two procedures is given in Section 4.3. Chapter 5 describes our implementation of FindCore^E, in particular, some further optimizations of the algorithm. First experimental results are presented in Chapter 6. We conclude with Chapter 7.

Chapter 2

Preliminaries

In this chapter, we formally present the basic definitions of data exchange, generally following the works of Fagin et al. [FKMP03] and [FKP03].

2.1 The data exchange problem

A *schema* is a finite set of relation symbols, having a name and a positive integer *arity*. *Relation instance* is a function mapping relation symbols into relations of the same arity. Relation symbols are often abused to denote relations themselves, especially when relation instance is clear from the context or is not important.

Tuples of the relations may contain two types of *terms*: *constants* and *variables*. The latter are also called *labeled nulls*. Two nulls are equal iff they have the same label. For every instance J, $\text{dom}(J)$ denotes the set of terms of J, $\text{var}(J)$ denotes the set of variables, and $\text{const}(J)$ denotes the set of constants. Clearly, we have $\text{dom}(J) = \text{var}(J) \cup \text{const}(J)$ and $\text{var}(J) \cap \text{const}(J) = \emptyset$. 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 $1 \leq i \leq n$, 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 x_i.

In data exchange, there are two participating schemas: $\mathbf{S} = \{S_1, S_2, ...S_n\}$ and $\mathbf{T} = \{T_1, T_2, ...T_n\}$, which are respectively called *source* and *target* schemas. It is required that $\mathbf{S}$ and $\mathbf{T}$ were disjoint. Relations, associated with symbols S_i are called *source relations* and constitute *source schema instance*. Similarly, *target schema instance* is represented by the set of *target relations* T_i.

Source-to-target dependencies (STDs for short) are dependencies of the form

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

where $\phi_S(\vec{x})$ with free variables $\vec{x}$ is a logical formula over the source schema, and $\psi_{\mathbf{T}}(\vec{x})$ is a logical formula over the target schema with free variables $\vec{x}$. *Target* dependencies are closed formulas over the target schema $\mathbf{T}$. Since STDs span over two different schemas, it is sometimes useful to view the two as a unit (this also applies to instances). We use angle brackets to express this, writing $\langle I, J\rangle \models \tau$ when a formula τ over two schemas is satisfied by the combination of instances I and J.

The terms *dependency* and *constraint* are synonyms in the context of data exchange. Intuitively, they allow to distinguish between two modes of usage: in the first, we want to stress that one relation depends on another. For instance, source-to-target dependency prescribes that whenever a certain tuple combination occurs in the source instance, some dependent tuples should be also present to the target instance. The term *constraint* conveys a different meaning: they restrict a schema instance preventing certain data patterns

from appearing in it. To find a solution of a data exchange problem, we *build* a target schema instance, thus the term "dependency" better fits here. On the other hand, it might be not possible to enforce a dependency on a target schema, which results in failure of a data exchange process. In this situation, the term "constraint" is appropriate. In the following, we use both terms and do not draw any formal distinction between them.

Definition 2.1.1 *A* data exchange setting *is a quadruple* ($\mathbf{S}$, $\mathbf{T}$,Σ_{st},Σ_t) *consisting of a source schema* $\boldsymbol{S}$*, a set* Σ_{st} *of source-to-target dependencies and a set* Σ_t *of target dependencies. The* data exchange problem *for this setting is the following: given a finite source instance* I*, find a finite target instance* J *such that* $\langle I, J\rangle$ *satisfies* Σ_{st} *and* J *satisfies* Σ_t*. Such* J *is called a* solution *for* I*.*

In this thesis we deal only with two classes of dependencies: *tuple generating dependencies* (TGDs) and *equality generating dependencies* (EGDs). A *tuple generating dependency* is a logical sentence of the form:

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

where both ϕ and ψ are conjunctions of atoms. An *equality generating dependency* is a logical sentence of the form:

$$\forall \vec{x}\ (\phi(\vec{x}) \rightarrow x_i = x_j), \text{such that } i, j \leq |\vec{x}|.$$

We call the left-hand side of an implication a *premise*, and the right-hand side — a *conclusion*. For both EGDs and TGDs, it is required that all the variables of $\vec{x}$ occur in the premise of the dependency, though some elements of $\vec{x}$ can be missing in the conclusion, as justified in [FKMP03]. In the following, we will omit the (outermost) universal quantifier when specifying dependencies, and consider all the variables universally quantified by default, unless existential quantification is explicitly specified for a variable.

Source-to-target constraints are restricted to TGDs (with premise query over the source schema, and conclusion over the target schema), while target constraints can be either TGDs or EGDs (over the target schema only). EGDs and TGDs are called *embedded implicational dependencies* [Fag82], and can express the most important types of database constraints, like functional dependencies, inclusion dependencies, multi-valued dependencies, and join dependencies.

2.2 Relations between instances

Given two instances of the same schema, it is often necessary to compare the two, in particular, with respect to generality. To do so, relations based on a notion of homomorphism are commonly used. All relations we deal with are *functions*, i.e., deterministic mappings, defined on the domains of two instances. Besides, they are all constant-preserving in a sense that, while variables can be mapped on arbitrary terms, an image of a constant can be only the constant itself. Let r be a relation between the domains of two instances I and J, mapping $\text{dom}(I)$ onto $\text{dom}(J)$. Then $\text{dom}(I)$ is also the *domain of* r, and $r(\text{dom}(I)) \subseteq J$ — a *range of* r.

For two instances I and J, a relation h between $\text{dom}(I)$ and $\text{dom}(J)$ is a homomorphism (written $h\colon I \rightarrow J$) if, whenever a fact $S(x_1, \ldots x_n)$ of the relation S/n is present in I, a fact $S(h(x_1), \ldots h(x_n))$ is in J. A homomorphism mapping an instance on itself is called *endomorphism*. Furthermore, one class of endomorphisms will be very useful for our further discussion, namely *idempotent* endomorphisms. That is, for an endomorphism r,

$\forall x \in \text{dom}(r)\ r(r(x)) = r(x)$. Such endomorphisms are called *retractions*. If there is a retraction mapping an instance K onto its subinstance $K' \subseteq K$, we also say that K' is a *retract* of K, denoting it as $K \hookrightarrow K'$.

2.3 Universal solutions

Data exchange problems can have infinite number of solutions. For example, consider the case where the source instance consists of the single fact $S(a)$, the set of STDs $\Sigma_{st} = \{S(x) \to \exists y R(x, y)\}$, and the set Σ_t of TDs is empty. Then all of the sets $\{R(a, a)\}$, $\{R(a, x_1)\}$, $\{R(a, x_1), R(a, x_2)\}$, etc. are solutions to the corresponding data exchange problem. Quite naturally, the question of optimality of the solution arises. A criterion proposed in [FKMP03] is *universality*, that is, there should be a homomorphism from the universal solution to any other solution possible. That is, a universal solution is general enough to "encompass" all possible materializations.

Example 2.3.1 *Suppose that the source instance consists of two relations* Tutorial(course, tutor)*: {('java', 'Yves')} and* BasicUnit(course)*: {'java'}. Moreover, let the target schema have four relation symbols* NeedsLab(id_tutor,lab), Tutor(idt,tutor), Teaches(id_tutor, id_course) *and* Course(idc, course). *Now suppose that we have the following STDs:*

1. BasicUnit$(C) \to$ Course(Idc, C).
2. Tutorial$(C, T) \to$ Course(Idc, C), Tutor(Idt, T), Teaches(Idt, Itc).

and the TDs are given by the two TGDs:

3. Course$(Idc, C) \to$ Tutor(Idt, T), Teaches(Idt, Idc).
4. Teaches$(Idt, Idc) \to$ NeedsLab(Idt, L).
5. Course$(Id_1, Course)$, Course$(Id_2, Course) \to Id_1 = Id_2$.

Then the following instances are all valid solutions:

J = {Course(C_1,'java'), Tutor(T_1,'Yves'), Tutor(T_2,N_1), Teaches(T_1,C_1), Teaches(T_2,C_1), NeedsLab(T_1,L_1), NeedsLab(T_2,L_2)},

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)}

Note that J' is not universal, since there exists no homomorphism $h\colon J' \to J$. Indeed, a homomorphism maps any constant onto itself and, therefore, the fact Course('java','java') cannot be mapped onto a fact in J.

Intuitively, one can say that non-universal solutions are "not induced" by the source database and the set of dependencies: there is no formal reason to prefer one non-universal solution over another (e.g., there is no reason to instantiate the variable of Course.idc necessarily with the constant "java"). Conversely, there is a universal solution which is logically forced [BV84] by a source instance and a set of dependencies and is called *canonical*. A procedure capable of finding such a logically justified solution is called *the chase*. It is presented in the following section.

2.4 Chase and canonical universal solutions

The task of computing a solution for a data exchange problem can be accomplished by the *chase* [BV84], a sequence of steps, each enforcing a single constraint within some limited set of tuples.

In case of enforcing source-to-target dependencies, we start with a pair of instances $\langle S, \emptyset \rangle$, where S is a source instance, and $\emptyset$ is an empty target instance T. Then the chase procedure updates T, ultimately yielding a *preuniversal instance*.

The target chase starts with a single target instance, which is then being updated, until the canonical universal solution is reached.

More precisely, suppose that Σ contains a TGD $\tau\colon \phi(\vec{x}) \to \exists \vec{y}\ \psi(\vec{x}, \vec{y})$, 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 $\xi\colon \phi(\vec{x}\,) \to x_i = x_j$, s.t. $I \models \phi(\vec{a}\,)$ for some assignment $\vec{a}$ on $\vec{x}$. Then 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 the dependencies Σ is denoted as I^{Σ}.

Since TGDs introduce new tuples in the instance, which can in turn lead to further chase steps, the termination of the procedure is an important question. As proposed in [FKMP03], the *weak-acyclicity* property of target TGDs is necessary for preventing infinite chase sequences. To define it formally, a notion of *dependency graph* of a set of dependencies Σ was introduced by Fagin et al.

The vertices of the graph are *fields* ("attributes") of relations referenced by the dependencies of Σ. We specify the fields as R^i, where R is a relation symbol, and i is a field (attribute) index.

For every TGD $\phi(\vec{x}) \to \exists \vec{y}\ \psi(\vec{x}, \vec{y})$ of Σ, the edge $\left(R^i, S^j\right)$ is present in G^D whenever a variable $r \in \vec{x}$ occurs both in ψ *and* in the field R^i in $\phi(\vec{x}\,)$ and (1) S^j is a field of $\psi(\vec{x}, \vec{y}\,)$ occupied by r, or (2) S^j is a field of $\psi(\vec{x}, \vec{y}\,)$ occupied by a variable $v \in \vec{y}$. Edges resulting from rule (2) are called *special*.

A set of TGDs is *weakly-acyclic* if there is no cycle containing a special edge. A *full TGD* is a TGD without existentially quantified variables, i.e. of the form $\tau\colon \phi(\vec{x}\,) \to \psi(\vec{x}\,)$. Figure 2.1 below shows the dependency graph for the target TGDs in Example 2.3.1, where special edges are marked with *. Clearly, this graph has no cycle containing a special edge (actually, it contains no cycle at all). Hence, these TGDs are weakly-acyclic.

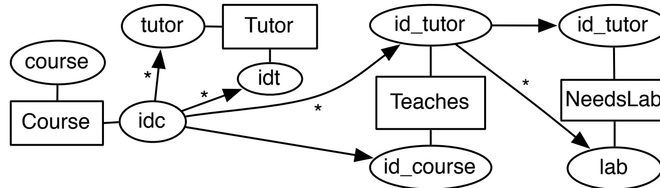


Figure 2.1: Dependency graph.

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 a set of dependencies Σ is the maximal depth of any field occurring in Σ.

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

Fitness of chase for solving data exchange problems was shown in [FKMP03] by proving the following

Theorem 2.4.1 *[FKMP03] If $\Sigma = \Sigma_{st} \cup \Sigma_t$, where*

- *Σ_{st} is the set of source-to-target embedded dependencies and*
- *Σ_t is the set of weakly-acyclic set of TGDs, and*

$(S, \emptyset)^\Sigma$ is defined and equals (S, U) for some U, then U is a universal solution for S under Σ.

For the computational cost of the chase, we have

Theorem 2.4.2 *[FKMP03] For every weakly-acyclic set Σ of TGDs and EGDs, there are b and c such that, for any A, regardless of the order of the chase and except for the case where the chase fails due to EGDs,*

- *A^Σ is defined, and*
- *A^Σ can be computed in $O(|A|^b)$ steps and in time $O(|A|^c)$.*

The chase is thus a procedure having a polynomial running time, guarantied to produce a universal solution for a data exchange problem, if a problem can be solved in principle.

2.4.1 Additional definitions

To reason about the effects of EGDs it is convenient to introduce some additional notation, following [FKP03]. 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. Let $t = S(u_1, u_2...u_s)$ be a fixed fact in the relation S of J. By $[u_i]$ we denote terms at corresponding positions of the same fact t in J', which is now $S([u_1], [u_2]...[u_s])$. If Σ_t contains no EGDs, then $\forall u \in \mathrm{dom}(J)\, u = [u]$. For arbitrary sets of dependencies, constants are mapped onto themselves: $\forall c \in \mathrm{const}(J) c = [c]$. For $u, v \in \mathrm{dom}(J)$, we write $u \sim v$ if $[u] = [v]$, i.e. two terms have the same image in J'.

Property 2.4.1 *The $[\cdot]$ relation on terms is a homomorphism.*

Proof Let instance J_s^Σ result from the sth chase step of J with the set of dependencies Σ. By definition of $[\cdot]$, if a fact $S(u_1, u_2, ...u_n)$ is in J_s^Σ, then $S([u_1], [u_2], ...[u_n])$ is in J^Σ. By definition of the chase, $[\cdot]$ is a function (since it represents a variable substitution at all positions of J_s^Σ), which preserves the constants. Then correctness of the property follows. □

2.5 Cores

In [HN92], studying cores of graphs, the notion of structure is considered, which is very close to the relational instance. A *structure* is determined as a tuple $\mathbf{A} = \{A, R_1, \ldots R_n\}$, where A is a explicit *universe* (which in case of relational instances is the *active domain*), and R_i are relation symbols representing relations over A. In case of structures, there are no special elements of the universe as constants, which should be preserved by homomorphisms.

Definition 2.5.1 *A substructure* $\boldsymbol{C}$ *of structure* $\boldsymbol{A}$ *is called a core of* $\boldsymbol{A}$ *if there is a homomorphism from* $\boldsymbol{A}$ *to* $\boldsymbol{C}$*, but there is no homomorphism from* $\boldsymbol{A}$ *to a proper substructure of* $\boldsymbol{C}$*. A structure* $\boldsymbol{C}$ *is called a core if it is a core of itself, that is, if there is no homomorphism from* $\boldsymbol{C}$ *to a proper substructure of* $\boldsymbol{C}$*.*

As shown in [HN92], the following propositions concerning the cores of structures hold:

- Every finite structure has a core, and the two cores of the same structure are isomorphic.
- If $\mathbf{C}$ is a core of a finite structure $\mathbf{A}$, then there is a homomorphism $h\colon \mathbf{A} \rightarrow \mathbf{C}$ such that $h(v) = v$ for every member v of the universe C of $\mathbf{C}$.

Following [FKP03], one then can identify instances with structures, active domain of an instance being a universe, and additionally distinguish part of the elements of the universe as constants. Then, core of an instance can be defined analogously to the core of a structure, and the properties of the latter carry over to relational instances.

Furthermore, the following theorem is proved in [FKP03], which justifies the use of cores in data exchange:

Theorem 2.5.1 *[FKP03] Let* $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ *be a data exchange setting, in which* Σ_{st} *is a set of TGDs and* Σ_t *consists of TGDs and EGDs. If* J *is a solution for a source instance* I*, then* $core(J)$ *is a solution for* I *as well. Consequently, if* J *is a universal solution for* I*, then also* $core(J)$ *is a universal solution.*

Chapter 3

Core computation

For an arbitrary relational instance with variables, core computation is NP-complete, being equivalent to some well-known NP-complete problems, like computing the core of an arbitrary graph [HN92], or computing the smallest equivalent subquery contained in a conjunctive query [CM77]. However, in the case of data exchange we are dealing with instances generated by the well-defined procedure, namely the chase. Therefore, the target instance is fully determined by the source instance (which is free of variables) together with the dependencies, and for each labeled null in the target instance, it is always possible to say which dependency has forced its creation. One may ask if there is a way to employ this information for effective core computation. Gottlob and Nash showed, that the answer is positive [GN06]. It is not surprising, however, that the complexity of core identification depends the classes of allowed target dependencies. Thus, in absence of target dependencies, the core computation was proven to be tractable by Fagin et al. [FKP03], shortly after the first formulation of data exchange problem as such. It was first shown in [GN06] that the target dependencies can comprise both weakly-acyclic TGDs and EGDs, without ruining the tractability of core computation. In this chapter we present and discuss the core computation algorithm FindCore [GN06]. It will then become the basis of our further development, including algorithmic improvements, and the prototype implementation.

3.1 Development of the core computation algorithms

This section presents the core computation algorithms dealing with different kinds of target dependencies. Although some of the procedures presented here can only tackle restricted classes of target dependencies, their underlying ideas (e.g. the notions of blocks, or rigidity of variables) will be very helpful for us later.

In all of the mentioned algorithms, the core computation problem is being approached in a similar way. Since the core of an instance I is its minimal homomorphic subinstance $I' \subseteq I$, all algorithms perform a descent from the universal solution produced by the chase, via the series of its nested endomorphic images (each being more and more tight core approximation), to a final subinstance, which cannot be further shrunk by an endomorphism. Fig. 3.1 illustrates this approach.

It was shown in [FKP03] that cores are unique up to isomorphism. That is, no matter what sequence of nested subinstances led to it, the minimal subinstance will have the same number of facts as any other minimal subinstance, and one can be transformed into another just with a renaming of variables. Therefore, sometimes we speak about *the core* of an instance.

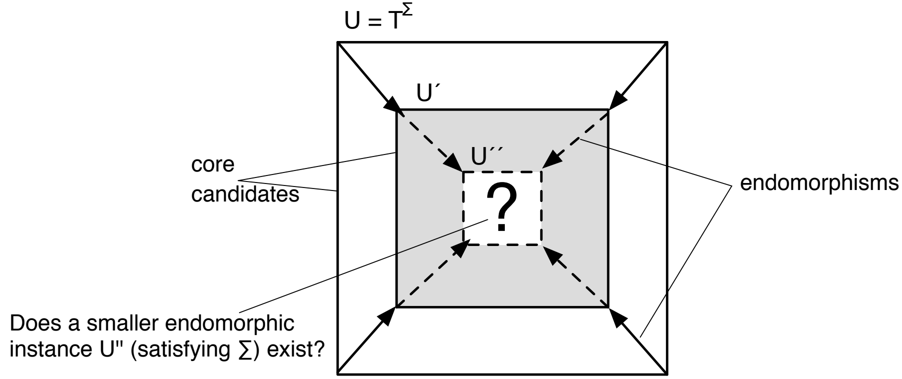


Figure 3.1: A descent to the core

3.1.1 Computing a core in absence of target dependencies

In absence of target dependencies, every endomorphic subinstance of a universal solution trivially satisfies the dependencies, so there is no need to check this at every iteration of the core computation algorithm. The first paper on cores in Data Exchange, [FKP03] provided two algorithms for such a case: a naive one (the GREEDY ALGORITHM) and the BLOCKS ALGORITHM, utilizing the fact, that a homomorphism for an instance can be found as a union of homomorphisms of its subinstances, if those subinstances share no variables with each other. The importance of this idea is easy to understand, since the search space for constructing a homomorhpism $h : I \to J$ is $|\mathrm{dom}(J)|^{|\mathrm{var}(I)|}$. By partitioning the variables of I, one reduces the search space exponentially. Because of the importance of this idea for our further developments, we cite the BLOCKS ALGORITHM here, as it was described in [FKP03].

Definition 3.1.1 *The* Gaifman graph $\mathcal{G}(I)$ *of an instance* I *is an undirected graph whose vertices are variables of an instance* 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* $\mathrm{block}(v, I)$*. The* block size *of instance* I *is the maximal number of variables in any of its blocks.*

It is thus possible to represent any homomorphism $A \to C$ as a union of homomorphisms $h_i : B_i^A \to C$, where B_i^A is the ith block of A. Since blocks share no variables, one can construct h_i independently of other homomorphisms $h_j, j \neq i$. The following theorem of [FKP03] relies on blocks to give a rough estimation of the homomorphism computation complexity for arbitrary instances.

Theorem 3.1.1 *[FKP03] Let* A *and* B *be instances, and suppose that* $\mathrm{blocksize}(A) \leq c$ *holds. Then the check if a homomorphism* $h : A \to B$ *exists and, if so, the computation of* h *can both be done in time* $O(|A| \cdot |B|^c)$*.*

PROOF (SKETCH) The crucial observation is that, in order to search for a homomorphism $h : A \to B$, we may search for homomorphisms from every block of A 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 onto B. □

In absence of target dependencies, it is easy to see, that the blocks of a canonical universal solution are bounded, since a source-to-target dependency of height e can only generate tuples sharing at most e variables (there are no labeled nulls in the source schema).

Theorem 3.1.2 *[FKP03] If Σ is a set of source-to-target constraints of height e, S is ground, and $(S,T) = (S,\emptyset)^{\Sigma}$, then* blocksize$(T) \leq e$.

Now the BLOCKS ALGORITHM itself can be presented. Its general idea agrees with that depicted on Fig. 3.1. The algorithm starts with a canonical universal solution $U = T^{\Sigma}$ as the first core candidate. At every step, the algorithm systematically tries to eliminate a variable from the domain of the last core candidate U' with a proper endomorphism. That is, an endomorphism must eliminate at least one variable from the domain of U'. Let y be such a variable, then a desired endomorphism can be computed as a union of homomorphisms $h_i : B_i^{U'} \rightarrow U'$, where h_j is an identity mapping if $y \notin B_j^{U'}$. Thus, to get rid of a variable y with the help of an endomorphism, it suffices to search only a mapping for the block of y. As the LEMMA 3.1.1 shows, the blocks can be calculated only once.

Definition 3.1.2 *[FKP03] Let K and K' be two instances, such that* $\text{var}(K') \subseteq \text{var}(K)$*. Let h be some endomorphism of K', and let B be a block of nulls of K. We say that h* is K-local for B*, if $h(x) = x$ whenever $x \notin B$. We say that h is $K-local$ for K if h is $K-local$ for some block B of K.*

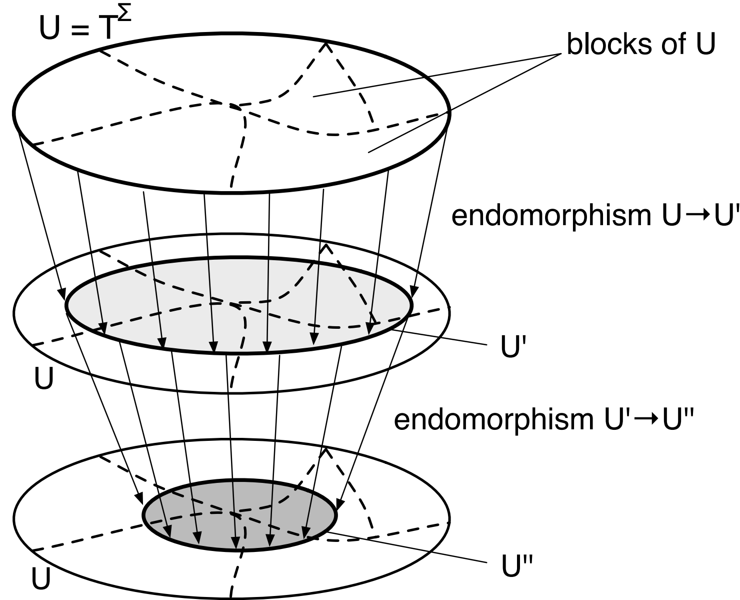


Figure 3.2: Blocks algorithm. Reusing the blocks of the universal solution

For every block $B^{U'}$ of $U' \subseteq U$, there exists block B^U of U, such that $B^{U'} \subseteq B^U$. Thus, the endomorphism eliminating a variable y can be build by only considering variables of U' restricted to a block B_y^U of U:

Lemma 3.1.1 *[FKP03] Assume a data exchange setting where Σ_{st} is a set of TGDs and $\Sigma_t = \emptyset$. Let U' be a subinstance of a canonical universal solution U. If there exists a proper endomorphism of U', then there exists a proper* U-local *endomorphism of U'.*

Fig. 3.2 illustrates the idea of reuse of blocks, computed after the chase, at later iterations of the core computation algorithm.

3.1.2 Computing a core in presence of target EGDs

It was shown in the previous subsection that the notion of blocks is very important for homomorphism computation. It was thus very comforting to find out, that source-to-target dependencies can only generate instances with the fixed block size. Unfortunately, when target dependencies come into play, this is no longer the case. Figure 3.3 shows how blocks are merged by different kinds of dependencies.

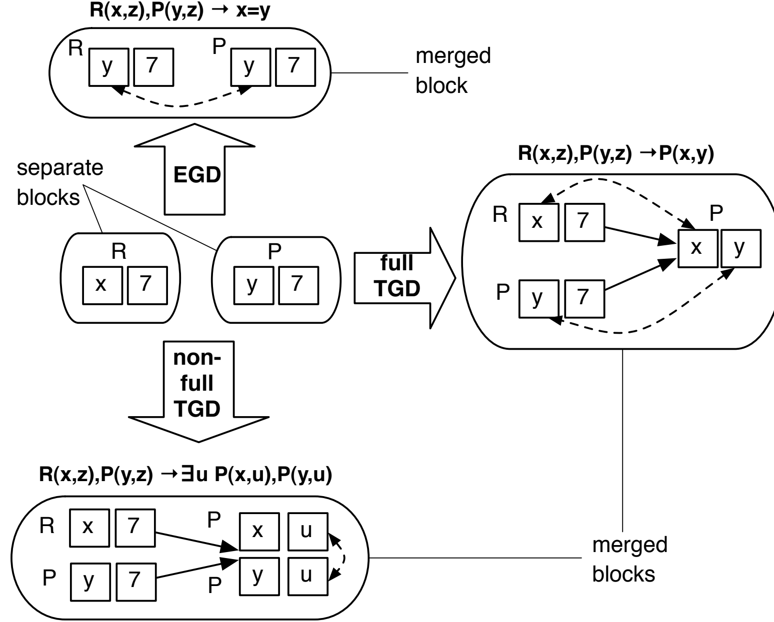


Figure 3.3: Different types of block merges

Of all possible merge types, the effects of EGDs seem to be the most "evil", since they substitute variables in-place, and thus leave little hope to find even a part of an instance with a block size independent of the number of facts in it.

However, it was a bright idea of Fagin et al. [FKP03] to show that one can disregard the effects that target EGDs have on the blocks of the preuniversal instance (i.e. the one created by the source-to-target chase). Thus, when the target dependencies contain the EGDs only, one may rely on the fixed block size property of the canonical solution. This is captured by the notion of *rigid* variables, explained below.

Definition 3.1.3 *Let K be an instance whose elements are constants and nulls. Let y be some element of K. We say that y is* rigid *if $h(y) = y$ for every endomorphism h of K. (In particular, all constants of K are rigid.)*

The rigidity of a variable implies, that with regard to homomorphisms it behaves like a constant: it can be mapped only on itself, and hence, can be safely overlooked when computing blocks of the instance. The block is called *non-rigid*, if it was computed considering only non-rigid variables. The following Rigidity Lemma [FKP03] relates the concept to the solutions of data exchange problems. The original lemma applies only to the case of target EGDs, but the close inspection of its proof shows, that it remains valid when the target constraints include TGDs.

Lemma 3.1.2 *(*Rigidity Lemma*) Assume a data exchange setting where Σ_{st} is a set of tgds and Σ_t is a set of egds* and tgds. *Let J be the canonical preuniversal instance, and let J' be the result of chasing J with the set Σ_t of egds. Let x and y be nulls of J such that $x \backsim y$, and such that $[x]$ is a nonrigid null of J'. Then x and y are in the same block of J.*

Proof sketch (adopted from [FKP03]). Unifications performed while chasing EGDs are logically forced [BV84] formula $\tau : \phi \rightarrow x = y$ where ϕ is a *diagram* of the instance J, that is, the conjunction of all its facts, $\Sigma_t \models \phi \rightarrow x = y$. Given the formula τ and an endomorphism $h : J' \rightarrow B$ where $B \subseteq J'$, one can show that a valuation V for the terms of ϕ can be constructed, s.t. $V(z) = [z]$ if $z \in block(x)$ and $V(z) = h([z])$ otherwise. This is possible because whenever a fact $R(x_1, ...x_n)$ is present in J, the fact $R([x_1], ...[x_n])$ is in J' by definition of $[\cdot]$. The fact $R(h([x_1]), ...h([x_n]))$ is also present in $B \subseteq J'$, because h is an endomorphism. Therefore, V makes ϕ applicable to J', that is, $J' \models \phi$. Since $J' \models \Sigma_t$ and $\Sigma_t \models \tau$ it must be the case that $J' \models \tau$, that is $V(x) = V(y)$. If x and y belong to different blocks of J, by the construction of V we have $[x] = h([y]) = h([x])$, i.e. $[x]$ is rigid. This contradicts the assumption of the lemma, and thus x and y should originate from the same block of J.□

As the Rigidity Lemma shows, when EGD unifies variables belonging to different blocks of a preuniversal instance J, the resulting variable is rigid. That allows to adapt a blocks method to a case with target EGDs, as the following lemma shows.

Lemma 3.1.3 *[FKP03] Assume a data exchange setting where Σ_{st} is a set of tgds and Σ_t is a set of egds. Let J be the canonical preuniversal instance, and let J'' be an endomorphic image of the canonical universal solution J'. If there exists a useful endomorphism of J'', then there exists a useful J-local endomorphism of J''.*

3.1.3 Adding restricted target TGDs

In [Got05] two further algorithms were introduced, capable of dealing with some more expressive target constraints, namely

- EGDs and weakly-acyclic *simple* TGDs and
- EGDs and full TGDs.

A TGD is called *simple* if its premise consists of a single atom and having no variable occuring more than once, i.e. simple TGDs are of the form:

$$\forall x_1 \ldots \forall x_k R(x_1, \ldots x_k) \rightarrow \exists y_1 \ldots \exists y_n (R_1(x_1, \ldots x_k, y_1 \ldots y_n) \wedge \ldots R_m(x_1, \ldots x_k, y_1 \ldots y_n)).$$

Clearly, simple TGDs cannot merge blocks, therefore blocksize of a canonical universal solution is determined by the dependencies (since the depth of each variable is no bigger than the depth of target dependencies, and hence, target chase can only add a fixed number of variables to each block of a preuniversal instance).

The algorithm tackling full TGDs [Got05] is quite involved, and requires some additional definitions and theorems. Since it is not directly relevant for the main matter of our discussion, we omit its description here, postponing the treatment of full TGDs until the next chapter.

Instead, we describe an approach to covering target EGDs (in addition to full TGDs) which carries over to the FindCore algorithm – a basis for our implementation. The trick is to mimic the effect of EGDs with full TGDs: the new "equality" relation E is

introduced, and all terms that would have been equated by EGDs are stored there. Then, for each fact that holds a variable x, such that $(x, y) \in E$ or $(y, x) \in E$, a new variant holding y instead of x is introduced by full TGDs. Thus, among different instantiations, there is a fact that the original EGD would have generated.

As explained in [GN06], every EGD $\varphi(\vec{x}) \rightarrow x_i = x_j$, where $i, j \leq |\vec{x}|$, is replaced by a TGD: $\varphi(\vec{x}) \rightarrow E(x_i, x_j)$. Besides, there are additional TGDs that reflect the equality axioms:

- $E(x, y) \rightarrow E(y, x)$ (symmetry),
- $E(x, y), E(y, z) \rightarrow E(x, z)$ (transitivity), and
- $R(x_1, \dots x_x) \rightarrow E(x_i, x_i)$ (reflexivity) for each target relation R of arity k and for every its field $i \leq k$.

Finally the following *consistency* constraints ensure that all possible instantiations of tuples including terms affected by EGD are present in the target instance:

- $R(x_1, \dots, x_j, \dots, x_n), E(x_j, y) \rightarrow R(x_1, \dots, y, \dots, x_n)$

For the case, when only full TGDs are allowed along with EGDs, such a simulation requires no additional precautions, and the usual chase procedure can be used to enforce them. As shown in [GN06], this is not the case with weakly-acyclic TGDs and EGDs: after the simulation, the weak acyclicity property can be lost.

3.2 The FindCore algorithm

The previous subsections of this chapter presented the core computation algorithms capable of dealing with some limited classes of embedded target dependencies. We quoted them to demonstrate core computational aspects, which play an important role also the procedure we present next, namely the FindCore algorithm by G.Gottlob and A.Nash [GN06]. It contains a number of elegant ideas allowing to compute the cores of canonical universal solutions produced by chasing both weakly-acyclic TGDs and EGDs.

3.2.1 Prerequisites

Consider the chase of the preuniversal instance T with the target dependencies Σ_t. If $\vec{y}$ is a tuple of variables created by enforcing a TGD $\phi(\vec{x}) \rightarrow \exists \vec{y}\ \psi(\vec{x}, \vec{y})$ in Σ_t, such that 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 $\vec{y}$;
- the *ancestor* relation is the transitive closure of the parent relation.

3.2.2 Basic ideas

Use of retractions

One of the insightful results of [GN06] was demonstrating the role of retractions for enabling the iterative core computation algorithm (Fig. 3.1) that computes a sequence of subinstances. The problem with such an approach when target dependencies including TGDs, is that if an instance U satisfies the set of dependencies Σ including TGDs, an endomorphic image $h(U)$ of U may not satisfy Σ [GN06]. To remedy this, a very elegant solution was proposed: it suffices to use retractions instead of endomorphisms, to ensure that resulting subinstances satisfy the same embedded dependencies as the instance itself.

Theorem 3.2.1 *If Σ is the set of embedded dependencies, then Σ is closed under retractions. That is: if $A \models \Sigma$ and $A \hookrightarrow B$, then $B \models \Sigma$.*

But how to compute the retraction? The answer is given by the following theorem:

Theorem 3.2.2 *[GN06] Given an endomorphism $h : A \to A$ such that $h(x) = h(y)$ for some $x, y \in \mathrm{dom}(A)$, there is a proper retraction r on A s.t. $r(x) = r(y)$. Moreover, such a retraction can be found in time $O(|\mathrm{dom}(A)|^2)$.*

Proof [GN06] Assume h is as in the hypothesis. We write h^q for the composition of h with itself q times. Since h is an endomorphism, $h(A) \subseteq A$. Since also $B \subseteq A$ implies $h(B) \subseteq h(A)$, it follows that $h^1(A) \supseteq h^2(A) \supseteq h^3(A) \supseteq \ldots$. This sequence can not decrease forever, so there must be some $q \leq |\mathrm{dom}(A)|$ such that $h \circ h^q(A) = h^q(A)$.

Set $g := h^q$ and $B := g(A)$. Then $h(g(A)) = g(A)$, and thus $g(B) = h^q(g(A)) = g(A) = B$. That is, g is an automorphism on B. Denote by G the graph of g on B, i.e., the digraph whose set of vertices is B and whose set of arcs is $\{(u, v) \colon u, v \in B \ \wedge \ g(u) = v\}$. Since g is a permutation, G consists of a collection $C_1, C_2, \ldots, C_k$ of disjoint cycles or loops C_i of respective lengths $c_1, \ldots, c_k$. These cycles correspond to the strongly connected components of G and can thus be identified in linear time. Obviously, we have

$$\sum_{i=1}^{k} c_i = |\mathrm{dom}(B)| \leq |\mathrm{dom}(A)|$$

Let $g_0 = g$ and for $0 \leq i \leq k-1$, let $g_{i+1} := g_i^{c_i}$. Set $r = g^k$. We thus have:

$$r = (((g^{c_1})^{c_2}) \ldots)^{c_k}$$

Note that for $0 \leq i \leq k$, g_i is the identity on vertices of all cycles $C_1, C_2, \ldots C_k$. Thus r restricted to B is the identity on B. Moreover, $r(A) = g(A) = B$, so r is a retraction of A. Note that r, arising from compositions of endomorphisms, is itself an endomorphism. Furthermore, $r = (h^q)^{c_1 \cdot c_2 \cdot \cdots \cdot c_k}$ and therefore, since $h(x) = h(y)$, it also holds that $r(x) = r(y)$. Thus r is the desired retraction. Computing r as described requires no more than $|\mathrm{dom}(A)| + \sum_{i=1}^{k} c_i \leq 2|\mathrm{dom}(A)|$ compositions. Each single composition is feasible in linear time in $|\mathrm{dom}(A)|$, and therefore r can be done in time quadratic in $|\mathrm{dom}(A)|$. $\square$

Nice chase

We already mentioned in the previous chapter, that the FindCore algorithm employees the approach of simulating EGDs with TGDs. However, when there are non-full TGD permitted in the target constraints, the resulting set of dependencies might become not weakly-acyclic [GN06]. To tackle this, Gottlob and Nash proposed the following: modify the classic chase procedure (terminating independently on the order of application of dependencies) by the chase, preserving the so-called *nice chase order*. Prior to defining it, yet another notion of *good* tuples w.r.t. a dependency must be introduced.

Definition 3.2.1 *[GN06] Fix some weakly-acyclic set Σ of TGDs and EGDs. Consider the dependency graph associated with the TGDs in Σ. If R is a relation, we say that a tuple $\vec{a}$ is good for R if the depth of every value in it is smaller than or equal to the depth of the corresponding field in R. That is, depth$(a_i) \leq$ depth(R_i). If $\varphi(\vec{x})$ is a conjunction of atoms with variables $\vec{x}$, we say that a tuple $\vec{a}$ of the same arity as $\vec{x}$ is good for φ if the depth of every value a_i in it is smaller than or equal to the the depth of every position in φ where x_i appears.*

Now, when several dependencies apply, the nice chase preserves the following order:

1. an equality constraint (see section 3.1.3)
2. a consistency constraint
3. a constraint firing on a tuple which is good for its premise
4. any other constraint

It is shown in [GN06], that any nice chase terminates on a set of weakly-acyclic dependencies and full TGDs simulating EGDs. Moreover, it is easy to see that, the nice chase makes only a fixed number of tests before each chase step, and hence it is only by some multiplicative constant less efficient than a usual, non-deterministic chase.

Gottlob and Nash show, that there are no other measures except the transformation of the target dependencies and preserving the chase order that are necessary to embrace target EGDs. Moreover, the core computation algorithm they propose allows to ignore the changes in the original target schema (i.e., the E-relation) as well as the host of simulating TGDs. Therefore, in the rest of this chapter, we only discuss the target dependencies consisting of weakly-acyclic TGDs.

Search for local endomorphism

According to the approach, illustrated by Fig. 3.1, the core can be found via series of proper endomorphisms. For any instance K, there can be no more than $|\text{var}(K)|$ elements in any such endomorhpism sequence. Consequently, what is left to find out is how to compute a local proper endomorphism effectively. Every proper endomorphism necessarily maps some two terms on one target. The algorithm idea is to systematically test all the pairs of terms (one of which must be a variable) and check if there exist an endomorphism "lumping" them together. The range of such an endomorphism is always the last found retract (a current *core candidate*), so the sequence of subinstances is shrinking and is guaranteed to eventually reach the core.

The effective way of computing a local proper endomorphism constitutes yet another bright finding of [GN06]. As it became clear from the previous chapters, blocks are the key factor for this. However, in presence of the target TGDs, blocks of the universal solutions are unbounded. Rigid variables are of limited help here, as they must occur in the facts of the preuniversal instance, while other variables (from the facts that were introduced by the target chase) can still form huge non-rigid blocks. To cope with this, one needs to further limit the number of variables participating in the generate-and-test process of endomorphism search.

The idea of [GN06] was to use the fact, that every step of the target chase is determined by the target dependencies and the instance that has been developed by the preceding chase sequence. If we are interested in a homomorphism treating a variable x in some specific way, we may trace back the chase steps that led to creation of x, select the facts participating in these steps, and hope that these facts will allow us to test if a desired homomorphism exists.

If such an instance segment, relevant w.r.t. some variable x could be built from facts over *ancestor* and *sibling* variables of x, it would be enough to make checking for existence of a local proper endomorphism tractable, as the number of ancestors and siblings for each variable is fixed for each set of target dependencies Σ.

This is exactly the idea implemented in the FindCore algorithm, which we present next.

3.2.3 Tackling weakly acyclic target TGDs

The FindCore algorithm is defined as follows:

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}; *(Lemma 3.2.1)*
(6) | Look for $h : T_{xy} \to U$ s.t. $h(x) = h(y)$; *(Lemma 3.2.3)*
(7) | **if** there is such h **then**
(8) | | Extend h to an endomorphism h' on U; *(Theorem 3.2.3)*
(9) | | Transform h' into a retraction r; *(Theorem 3.2.2)*
(10) | | Set $U := r(U)$;
(11) **return** U.

The first three steps are responsible of enforcing all the necessary constraints. As explained earlier, the target EGDs are simulated by full TGDs (step 1) which together with the weakly-acyclic TGDs of Σ constitutes the new set of target dependencies $\bar{\Sigma}$. After the target chase, the instance produced by the step 3 is guaranteed to satisfy $\bar{\Sigma}$, and the descent to a core begins. We illustrate an individual iteration of such descent, corresponding to steps (4)–(10) of the algorithm, with the Fig. 3.4.

The following theorem is essential for the described compuation method. It proves that given a subinstance W of a canonical universal solution T^{Σ} that includes a preuniversal instance T and is closed under parents and siblings, it is possible to extend any homomorphism $W \to B$ to a homomorphism $T^{\Sigma} \to B$, provided that the instance B satisfies Σ.

Recalling the core computation ideas outlined before, at each iteration of the algorithm we check if a further minimization of a locally minimal solution U' is possible, i.e., if a proper retraction r of U' exists, s.t. $r(x) = r(y)$ for some $x, y \in \mathrm{dom}(U')$. We check this by constructing a characteristic segment T_{xy} of a canonical solution T^{Σ}, and searching for a homomorphism $h : T_{xy} \to U'$ (see Lemma 3.2.2 for the justification). Such a check is effective, since T_{xy} has the domain only by a constant larger than $\mathrm{dom}(T)$ (Lemma 3.2.1), and thus enjoying a bounded block size property (see also Lemma 3.2.3).

Identifying T_{xy} with the instance W, and U' with the instance B of the Theorem 3.2.3 below, we obtain an important proof contribution for the chosen approach (step 8 of the algorithm): the theorem shows that a homomorphism $T^{\Sigma} \to U'$, which trivially converts into a desired endomorphism of $U' \subseteq T^{\Sigma}$, can be built effectively.

Theorem 3.2.3 *Let T^{Σ} be a universal solution of a data exchange problem obtained by chasing a preuniversal instance T with the weakly-acylclic set Σ_t of target TGDs. If B and W are instances such that:*

1. $B \models \Sigma$,
2. $T \subseteq W \subseteq T^{\Sigma}$, *and*
3. $\mathrm{dom}(W)$ *is closed under ancestors and siblings,*

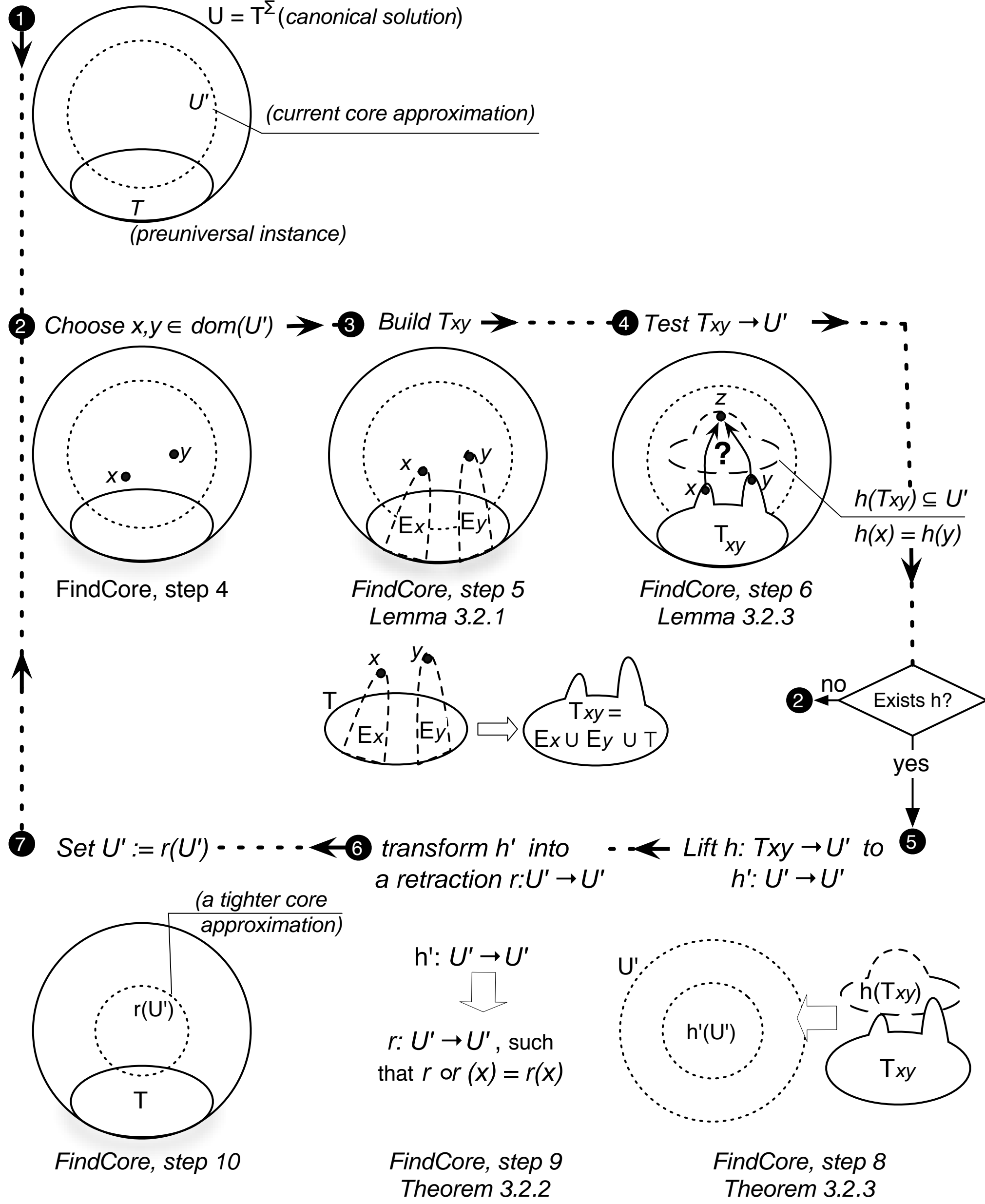


Figure 3.4: An iteration of the FindCore algorithm

then any homomorphism $h : W \rightarrow B$ *can be extended in time* $O(|\mathrm{dom}(T)|^b)$ *to a homomorphism* $h' : T^\Sigma \rightarrow B$ *where* B *depends only on* Σ.

The proof is postponed until the next chapter, where we prove a more general theorem, supporting EGDs in the set Σ, along with weakly-acyclic TGDs. For now, we just mention that the extension procedure replays all the non-full TGDs (the only type of dependencies that introduce new variables), and after every such TGD extending the homomorphism to the nulls it has generated. The full TGDs are ignored by this process.

Next, the Lemma 3.2.1 shows how that given a terms x and y, chosen on the step 4, it is possible to find a subinstance T_{xy} including all the facts that induced their creation by the chase. Such a subinstance is computed on the step 5.

More precisely, first the sets E_x and E_y of nulls closed under ancestors and siblings of

x and y respectively are built, and then a subinstance $T_{xy} = T \cup T^{\Sigma}|_{E_x \cup E_y}$ is extracted from T^{Σ}, as illustrated by Fig. 3.4.

Lemma 3.2.1 *For every weakly-acyclic set Σ of TGDs of depth d, width w, and height e, instance T and $x, y \in dom(T^{\Sigma})$, there is 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. $|\mathrm{dom}(T_{xy})| \leq |\mathrm{dom}(T)| + 2edw^d$,

then T_{xy} can be computed in time $O(|\mathrm{dom}(T)|^c)$ for some c which depends only on Σ.

A proof of the lemma can be found in [GN06]. We will prove an analogous statement in the next chapter, adopted for our version of the core computation algorithm.

It is important, that the domain of T_{xy} is only by a constant larger than a domain of a preuniversal instance. Thus, the block size of T_{xy} is determined by the block size of the preuniversal instance T (that is, by Σ_{st}) and by the target dependencies. That is, the block size of T_{xy} *is fixed for a given data exchange setting.* The next lemma demonstrates the role of such a characteristic subinstance for testing of existence of a local retraction.

Lemma 3.2.2 *[GN06,* THEOREM 8, ***Claim 2****] Let T be an instance, Σ – a set of weakly-acyclic TGDs and EGDs, U' – a retract of $U = T^{\Sigma}$, and x and y – two terms from* $\mathrm{dom}(U')$. *Then a proper retraction r of U', s.t. $r(x) = r(y)$, exists* iff *there is a homomorphism $h : T_{xy} \to U$, s.t. $h(x) = h(y)$.*

PROOF If there is such a homomorphism h, a proper endomorphism of U' can be built as a restriction of an endomorphism $h' : T^{\Sigma} \to U'$ (which exists by THEOREM 3.2.3) to the domain of U'. The desired retraction then can be obtained by THEOREM 3.2.2.

In the other direction, let r be retraction of U', $r(x) = r(y)$.By assumption, $U \hookrightarrow U'$ i.e., there is a retraction $r' : U \to U'$. Then $r'' = r' \circ r$ is a retraction $U \to U'$ as well and besides, $r''(x) = r''(y)$. We then get a homomorphism $h = r''|_{\mathrm{dom}T_{xy}}$. □

To check for a homomorphism $h : T_{xy} \to U$ on step 6 of FINDCORE, it suffices to examine all the blocks of T_{xy} independently of each other. In [GN06] the following lemma is proved.

Lemma 3.2.3 *Let T and U be instances, Σ – a set of weakly-acyclic TGDs. Given two terms $x, y \in \mathrm{dom}(T^{\Sigma})$, let T_{xy} be an instance closed under parents and siblings, s.t. $T \subseteq T_{xy} \subseteq T^{\Sigma}$ and $x, y \in \mathrm{dom}(T_{xy})$. It is then possible to check if there exists a homomorphism $h : T_{xy} \to U$ s.t. $h(x) = h(y)$ in the time $O(|dom(U)|^c)$ for some c which depends only on Σ and* $\mathrm{blocksize}(T)$.

In the next chapter, we prove the similar THEOREM 4.2.2 tailored for our version of the algorithm. Based on the above theorems and lemmata, the following theorem is proved in [GN06].

Theorem 3.2.4 *[FKMP03] For every $\Sigma = \Sigma_{st} \cup \Sigma_t$, where*

- *Σ_{st} is the set of source-to-target embedded dependencies and*
- *Σ_t is the set of weakly-acyclic set of TGDs and EGDs, and*

every ground instance S, a core for a canonical universal solution U under Σ can be computed in time $O(|\mathrm{dom}(S)|^b)$ for some b that depends only on Σ.

We give our proof for this statement, based on the algorithm $\textsc{FindCore}^E$ which is presented in the next chapter.

3.3 Discussion of the FindCore algorithm

The essence of this chapter was presentation of the FindCore algorithm, capable of computing the minimal universal solution for a data exchange problem. It follows the iterative approach to the minimization of a target instance, at each iteration shrinking the range of an endomorphism, which projects the instance onto its part. It is the first polynomial algorithm capable of tackling data exchange problems with target dependencies consisting of weakly-acyclic TGDs as well as of EGDs. Though it runs in polynomial time, there is a room for improvement. In particular, we do not find simulating EGDs with TGDs the optimal choice for implementation. There are two main reasons for this:

- Chase does not produce a solution for a data exchange problem. It is the core computation which first outputs a universal solution in presence of target EGDs. This is rather a conceptual issue, since computing a core is quite an expensive procedure, in contrast to the chase. Additionally, there are arguments in favor of materialization of canonical universal solutions *rather* then cores, based on alternative query answering semantics [Lib06].
- A lots of redundant tuples are added to the target instance, by full TGDs simulating EGDs.

Hence, we are confident that, unless the core computation is as easy as producing the universal solution by the chase, it should be regarded as an optional service of a data exchange engine. In many cases, canonical universal solutions are already usable options for materialization, and we argue that it is desirable to first produce a comparatively cheap yet useful result, and then refine it, when necessary.

Therefore, in the next chapter we present our version of the FindCore algorithm, capable of applying EGDs in the target chase, and thus better fitting our vision of the place of core computation in data exchange. There are other arguments in favor of our approach, which we present in the Discussion section of the next chapter.

Chapter 4

The enhanced core computation algorithm

As it was shown in the previous chapter, simulating EGDs with TGDs is rather a disadvantage for the FindCore algorithm. In this chapter, we propose a solution fixing this. The crucial point of our enhanced core computation procedure, 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.

We observe that EGDs may have both a positive and a negative effect on this core computation: Positively, the EGDs eliminate variables, thus leaving less work. Negatively, the EGDs may merge different blocks of the preuniversal instance T. Hence, without further measures, this would destroy the tractability of the homomorphism search, which is a basis of the FindCore algorithm. The required adaptations comprise both the actual computation and the proofs to establish the correctness and the polynomial time upper bound.

In this chapter, we present a FindCore^E algorithm, which seemingly proceeds exactly as the FindCore algorithm from Section 3.2, i.e.:

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

However, many of these steps cannot accommodate canonical solutions computed by the chase with EGDs. Actually, only the construction of a retraction r via Theorem 3.2.2 and the closure of embedded dependencies w.r.t. retractions according to Theorem 3.2.1 are general enough to carry over to the new environment without any changes. In contrast, the first 3 steps above require significant adaptations, e.g.:

- T_{xy} in Section 3.2 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 $h : T_{xy} \to U$ 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 leaving less work. 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 $h : T_{xy} \to U$.

- Since we have to define T_{xy} differently from Section 3.2, also the lifting of $h : T_{xy} \to U$ 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 compared with Section 3.2 are worked out below.

4.1 Preliminary definitions

4.1.1 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 the first position. Hence, in the atoms of the conclusions of STDs and in all atoms occurring in TDs, we have to add a unique existentially-quantified variable at the first position. For example, the source-to-target TGD $\tau \colon S(x) \to \exists y R(x, y)$ is transformed into $\tau^{id} \colon S(x) \to \exists t, y R^{id}(t, x, y)$.

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

During the chase, every fact of the target instance is assigned a unique *id* variable, which is never substituted by an EGD. We can therefore identify a fact with this variable:

1. If $R^{id}(t_1, x_1, \ldots, x_n)$ is a fact of a target instance $\mathbf{T}$, then we refer to it as *fact* t_1.
2. We define equality on facts as equality between their *id* terms: $R^{id}(t_1, x_1, \ldots, x_n) = R^{id}(t_2, y_1, \ldots, y_n)$ iff $t_1 = t_2$

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

4.1.2 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 the *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 4.1.1 *Let* $J = \{S(id_{S1}, x_1, y_1)\}$ *be a preuniversal instance and consider the following target dependencies:*

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

yielding the canonical universal solution J^Σ *shown in Figure 4.1:*

$$J^\Sigma = \{S(id_{S1}, x_1, y_1), P(id_{P1}, y_1, z_1), Q(id_{Q1}, y_1, v_1)\}$$

.

Every position of J *is native, being created by the source-to-target chase, which never copies labeled nulls. Thus the origin positions of* $(id_{S1}, 1)$ *and* $(id_{S1}, 2)$ *are these positions themselves. The latter is also the origin position for the two foreign positions* $(id_{P1}, 1)$ *and* $(id_{Q1}, 1)$*, introduced by the target chase. The remaining two positions of the facts* id_{P1} *and* id_{Q1} *are native.*

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

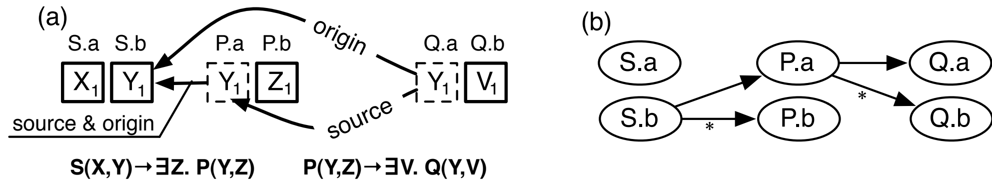


Figure 4.1: Positions of the instance J^Σ (foreign positions are dashed) (a) and the dependency graph of Σ (b).

Lemma 4.1.1 *Let* I *be an instance. Moreover, let* p *a position in* I *and* o_p *its origin position. Then* p *and* o_p *always contain the same term.*

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

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

4.1.3 Normalization of TGDs

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

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

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

Now let the full TGD τ_0 be defined as $\tau_0 \colon \phi(\vec{x}) \to \psi(\vec{x})$ and let the non-full TGDs τ_i with $i \in \{1, \ldots, n\}$ be defined as $\tau_i \colon \phi(\vec{x}) \to \psi(\vec{x}, \vec{y}_i)$. 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 the dependencies $\tau_0 \wedge \tau_1 \wedge \cdots \wedge \tau_n$.

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

Example 4.1.2 *Consider the non-full TGD*

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

Then τ is logically equivalent to the conjunction of the three TGDs: $\tau_0 \colon S(x, y) \to R(x, y)$, $\tau_1 \colon S(x, y) \to \exists z\ P(x, z)$, and $\tau_2 \colon S(x, y) \to \exists v\ Q(y, v)$. Clearly, these dependencies τ_0, τ_1, and τ_2 are normalized in the sense above.

The following property will play an important role when it comes to lifting a homomorphism h defined on T_{xy} to a homomorphism h' on T^{Σ_t} in Theorem 4.2.1.

Lemma 4.1.2 *Let the preuniversal instance J be chased with the set Σ_t of TDs in* normal form. *Suppose that at some step in the chase, the non-full TGD $\tau : \phi(\vec{x}) \to \exists \vec{y}\ \psi(\vec{x}, \vec{y})$ fires. Then τ introduces a new fact for every atom in the conclusion $\psi(\vec{x}, \vec{y})$. More precisely, suppose that τ fires with the assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$. Then all atoms in $\psi(\vec{a}, \vec{z})$ are newly created by this chase step.*

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

subformula $\rho'(\vec{a},\vec{z})$ of $\psi(\vec{a},\vec{z})$ already exist in J'. We have to show that $\rho(\vec{a},\vec{z})$ comprises all atoms of $\psi(\vec{a},\vec{z})$.

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

4.1.4 The parent and sibling relations over facts

Let I be an instance after the jth chase step and suppose that in the next chase step, the *non-full* TGD $\tau\colon \phi(\vec{x}) \rightarrow \exists \vec{y}\ \psi(\vec{x},\vec{y})$ 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 4.1.2, all other atoms of $\psi(\vec{a},\vec{z})$) are the *siblings* of t. Given a fact t, its *parent* set consists of the origin facts for any foreign position in t or in any of its siblings. The *ancestor* relation on facts is the transitive closure of the parent relation.

This definition of siblings and parents implies that facts introducing no fresh nulls (since we are assuming the above normal form, these are the facts created by a full TGD) can be neither parents nor siblings.

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

Example 4.1.3 *Let us revisit the two TGDs $S(id_S, x, y) \rightarrow \exists z\ P(id_P, y, z)$ and $P(id_P, y, z) \rightarrow \exists v\ Q(id_Q, y, v)$ from Example 4.1.1, see also Figure 4.1. 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 $S(x_1, y_1)$.*

4.2 Enhancing the algorithm

We are now ready to prove the main results underlying the FindCore^E algorithm, i.e.: Definition of T_{xy} (Lemma 4.2.1), search for a homomorphism $h : T_{xy} \rightarrow U$ (Lemma Lemma 3.1.2 and Theorem Theorem 4.2.2), and lifting a homomorphism $h : T_{xy} \rightarrow U$ to a non-injective homomorphism $T^{\Sigma_{st}} \rightarrow U$ (Theorem Theorem 4.2.1).

Lemma 4.2.1 *For every weakly-acyclic set Σ_t of TGDs and EGDs, instance T, and $x, y \in dom(T^{\Sigma_t})$, there exist constants b, c which depend only on $\Sigma = \Sigma_{st} \cup \Sigma_t$ and an instance T_{xy} satisfying*

1. *$Origin_x, Origin_y \subseteq T_{xy}$,*
2. *All facts of T are in T_{xy}, and $T_{xy} \subseteq T^{\Sigma_t}$,*
3. *T_{xy} is closed under parents and siblings over facts,*
4. *$|\mathrm{dom}(T_{xy})| \leq |\mathrm{dom}(T)| + b$.*

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

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

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

Having a homomorphism $h : T_{xy} \to U$, we want to extend it to a homomorphism $h'\colon T^{\Sigma_{st}} \to U$, analogously to Theorem 3.2.3. However, compared with Lemma 3.2.1, 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 [GN06, Theorem 7]. At any rate, the following theorem shows that also in our case, the lifting can be performed efficiently by essentially the same procedure as described in [GN06].

Theorem 4.2.1 *(LIFTING) Let T^{Σ_t} be a universal solution of a data exchange problem obtained by chasing a preuniversal instance T with the weakly-acyclic set Σ_t of TGDs and EGDs. If B and W are instances such that:*

1. *$B \models \Sigma$ with $\Sigma = \Sigma_{st} \cup \Sigma_t$;*
2. *All facts of T are in W (i.e. W contains facts with the same* ids*) and $W \subseteq T^{\Sigma_t}$, and*
3. *W is closed under ancestors and siblings (over facts),*

then any homomorphism $h\colon W \to B$ can be transformed in time $O(|\text{dom}(T)|^b)$ into a homomorphism $h'\colon T^{\Sigma_t} \to B$, s.t. $\forall x \in dom(h)\colon h(x) = h'(x)$, where b depends only on Σ.

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

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

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

$P([u_1], \ldots, [u_k]) \in W$ 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.
4. a non-full TGD, introducing facts present in W.

Note that cases 3 and 4 do not intersect, by Lemma 4.1.2 and by the fact that W is closed under siblings.

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

Case 1. T_s is obtained from T_{s-1} via the EGD $\varphi(\vec{x}) \to x_i = x_j$, where $i, j \leq |\vec{x}|$ 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 $\varphi(\vec{x}\,) \to x_i = x_j$ fires with this assignment in step s of the chase. Then $B \models \phi(h_{s-1}(\vec{a}\,))$, since h_{s-1} is a homomorphism. By condition 1 of the Theorem, $B \models \Sigma$. In particular, the EGD $\varphi(\vec{x}\,) \to x_i = x_j$ 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 same argument as above: $T_{s-1} \models \phi(\vec{a}\,)$ holds, since the TGD τ fires with this assignment on $\vec{x}$. Hence, $B \models \phi(h_{s-1}(\vec{a}\,))$, since h_{s-1} is a homomorphism Finally, since $B \models \Sigma$, also $B \models \psi(h_{s-1}(\vec{a}\,))$ holds.

Case 3. T_s is obtained from T_{s-1} via the non-full TGD $\phi(\vec{x}) \to \exists \vec{y}\ \psi(\vec{x}, \vec{y})$ with assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$. Moreover, all atoms in $\psi(\vec{a}, \vec{z}\,)$ are outside W. As above, we have $T_{s-1} \models \phi(\vec{a}\,)$ and $B \models \phi(h_{s-1}(\vec{a}\,))$. Moreover, by $B \models \Sigma$, there exist a vector $\vec{c}$ of terms in $\mathrm{dom}(B)$, s.t. $\psi(h(\vec{a}\,), \vec{c}\,) \subseteq B$. By Lemma 4.1.2, all atoms in $\psi(h(\vec{a}\,), \vec{c}\,)$ are newly created in T_s and, hence, all terms in $\vec{z}$ are fresh nulls. We extend h_{s-1} to h_s by setting $h_s(\vec{z}\,) := \vec{c}$. Then h_s is a homomorphism, since the image $\psi(h(\vec{a}\,), \vec{c}\,)$ of the new atoms $\psi(h(\vec{a}\,), \vec{z}\,)$ in T_s is in B by definition. Perhaps, for some $z \in \vec{z}$ also $[z] \in \mathrm{dom}(h) = \mathrm{dom}(W)$ holds. Since all facts of W are handled by the next case, it has to be an effect of some EGDs triggering later in the chase. By Case 1, a term y replaces a variable x only if their homomorphic images coincide. Hence, h_s is consistent with h.

Case 4. T_s is obtained from T_{s-1} via the non-full TGD $\phi(\vec{x}) \to \exists \vec{y}\ \psi(\vec{x}, \vec{y})$ with assignment $\vec{a}$ on $\vec{x}$ and assignment $\vec{z}$ on $\vec{y}$. Moreover, W already contains a fact for every atom in $\psi(\vec{a}, \vec{z}\,)$. Analogously to case 3, the vector $\vec{z}$ consists of fresh nulls. Moreover, since all atoms of $\psi([\vec{a}\,], [\vec{z}\,])$ are contained in W, the homomorphism $h\colon W \to B$

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 $x \in \text{dom}(h_{s-1})$. In other words, native positions in $\psi(\vec{a}, \vec{z})$ are mapped consistently with h. In order to show that h_s is a homomorphism, it remains to prove that all atoms in $\psi(h_s(\vec{a}), h_s(\vec{z}))$ are contained in B. By definition, we have $h_s(\vec{z}) = h([\vec{z}])$. Hence, it suffices to show that $h_{s-1}(\vec{a}) = h([\vec{a}])$ holds.

W is closed under parents and siblings and, therefore, the origin of every position of $\psi(\vec{a}, \vec{z})$ is contained in W, by the definition of the parent relation over facts. According to Lemma 4.1.1, a position p and its origin position o_p (which is either contained in some fact in T or which was introduced previously at some chase step $k < s$) are always occupied by the same term. If the position o_p is contained in some fact in T, then the term u at o_p was mapped to $h([u])$ by h_0 according to the induction begin. If o_p is contained in some fact introduced at some chase step $k < s$, then the term u at o_p was mapped to $h([u])$ by h_k according to Case 4. Note that none of the four cases considered by the induction step modifies a previously chosen image. Hence, o_p is mapped to $h([u])$ also by h_{s-1}. Hence, we indeed have $h_{s-1}(\vec{a}) = h([\vec{a}])$, as required.

This concludes the induction proof. In order to actually construct the homomorphism $h' = h_n$, we may thus simply replay the chase and construct h_s for every $s \in \{0, \dots, n\}$. The length n of the chase is polynomially bounded by Theorem 2.4.2. The action required to construct h_s from h_{s-1} by the above case distinction clearly fits into polynomial time as well. We thus get the desired upper bound on the time needed for the construction of h'. $\Box$

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 4.2.1, 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 3.1.1, 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 [FKP03] to our purposes, which was presented in the previous chapter. As we already pointed out, although the original RIGIDITY LEMMA was formulated for the data exchange scenarios with target dependencies restricted to EGDs only, it remains valid in presence of target TGDs as well. The lemma shows that if two blocks of variables of the preuniversal instance are merged by an EGD, i.e., two distinct variables are replaced by a single one, then the variable resulting from a substitution can *only be mapped on itself* by *any* endomorphism. Our intention to use rigidity can be captured with the following definition:

Definition 4.2.1 *The* non-rigid Gaifman graph $\mathcal{G}'(I)$ *of an instance I is a usual Gaifman graph 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 $\mathcal{G}'(I)$.*

Clearly, the non-rigid blocks can be much more advantageous for testing homomorphisms in presence of EGDs. It is important to note, that they are not just usual blocks modulo rigid variables, but rather blocks *constructed* while completely ignoring those variables. As a simple example, suppose that the variable y in the instance $\{R(x, y), R(z, y), R(w, y)\}$ is rigid. Then the non-rigid blocks are $\{x\}$, $\{z\}$ and $\{w\}$, which is much better than a usual block modulo y, containing 3 elements: $\{x, z, w\}$. The following lemma shows how the concept can be used.

Theorem 4.2.2 *Let T be a preuniversal instance obtained via the STDs Σ_{st}. Let Σ_t be a set of weakly-acyclic TGDs and EGDs, and let U be a retract of T^{Σ_t}. Moreover, let $x, y \in \mathrm{dom}(T^{\Sigma_t})$ and let $T_{xy} \subseteq T^{\Sigma_t}$ be constructed according to Lemma 4.2.1. Then we can check if there exists a homomorphism $h : T_{xy} \to U$, s.t. $h(x) = h(y)$ in time $O(|dom(U)|^c)$ for some c which depends only on $\Sigma = \Sigma_{st} \cup \Sigma_t$.*

Proof First, we prove that the rigid variables of T^{Σ_t} are also rigid in T_{xy}. Assume to the contrary that $x \in \mathrm{var}(T_{xy})$ is rigid in T^{Σ_t} and that there exists a homomorphism $h : T_{xy} \to U$ s.t. $h(x) \neq x$. By Theorem 4.2.1, h can be transformed into an endomorphism $h' : T^{\Sigma} \to U$, s.t. $\forall x \in \mathrm{dom}(h)$: $h(x) = h'(x)$. 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 $h : T_{xy} \to U$ 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 $h_i : B_i \to U$ be a homomorphism. Then the mapping $h : T_{xy} \to U$ defined as follows is well-defined and a homomorphism: For every $x \in B_i$, we set $h(x) := h_i(x)$ and for all x outside all B_i (i.e, x is rigid), we set $h(x) := [x]$.

Recall from Lemma 4.2.1 that T_{xy} has only constantly many variables in addition to T. By Theorem 3.1.2, 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.

Moreover, we did not need to consider all the blocks of T_{xy} to find a desired homomorphism, since we only should care that $h(x) = h(y)$, and therefore only two blocks, the one of x and the one of y are of particular interest, while other blocks of T_{xy} can be mapped no matter how. Recall that by a theorem assumption, U is a retract of T^{Σ}, i.e. there exists a retraction $r : T^{\Sigma} \to U$. (In the descent to a core, the first such retraction is just the identity mapping, and every subsequent optimization step results in a new retraction with a tighter range). Hence, we use already existent retraction r for mapping the blocks containing neither x nor y.

Hence, the problem boils down to finding a homomorphism for at most two blocks of fixed size, which depends on Σ only. The claim of the theorem then follows immediately. □

Prior to finally defining FindCore^E, we would like to revisit the procedure of lifting a homomorphism on the set T_{xy} into an endomorphism on the last found core approximation. Recall that the proof of Theorem 4.2.1 directly yields an algorithm for transforming a homomorphism $h : W \to B$ to an appropriate homomorphism $h' : T^{\Sigma_t} \to B$ in that we simply replay all n steps of the chase from T to T^{Σ_t} and construct intermediate homomorphisms $h_s : T_s \to B$ for $s \in \{0, \ldots, n\}$. Even though this algorithm has a polynomial time upper bound, it is slightly unsatisfactory since, 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 have therefore implemented the following simplified procedure Extend, where only chase steps corresponding to Case 4 in the proof of Theorem 4.2.1 are replayed. This procedure allows us to literally extend h to $h' : T^{\Sigma_t} \to B$ starting with W and considering only the variables present in T^{Σ_t}.

The correctness of this simplified procedure for extending h to h' is shown in the following lemma.

Lemma 4.2.2 *Let T, T^{Σ_t}, B, W, and $h : W \to B$ be as in Theorem 4.2.1. Then the procedure* Extend *extends h to a homomorphism $h' : T^{\Sigma} \to B$.*

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. We claim that W_j fulfills the following properties: $W_j \subseteq T^{\Sigma_t}$,

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. $Parents(A) \subseteq W$
(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'.

W_j contains all facts from T, W_j is closed under parents and siblings, and $h_j\colon W_j \to B$ is a homomorphism s.t. $\forall x \in \mathrm{dom}(W)\colon h'(x) = h(x)$ holds. We prove this claim by induction on j:

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

[induction step.] Suppose that the while-loop is entered for the $(j+1)$-st time. By the induction hypothesis, W_j together with the homomorphism $h_j\colon W_j \to B$ fulfills the assumptions on W in Theorem 4.2.1. 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.

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

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 $h : T_{xy} \to U$ s.t. $h(x) = h(y)$;
(6) | **if** there is such h **then**
(7) | | Call Extend to lift h to an endomorphism h' on U;
(8) | | Transform h' into a retraction r;
(9) | | Set $U := r(U)$;
(10) **return** U.

Putting all these pieces together, we get the FINDCORE^E algorithm. It has basically the same overall structure as the FINDCORE algorithm of [GN06], which we recalled in Section 3.2. 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 4.2.1, step (5) is based on LEMMA 3.1.2 and THEOREM 4.2.2, and step (7) is based on THEOREM 4.2.1. Analogously to THEOREM 3.2.4, we thus get

Theorem 4.2.3 *Let $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ be a data exchange setting with STDs Σ_{st} and TDs Σ_t. Moreover, let S be a ground instance of the target schema* $\mathbf{S}$. *If this data exchange problem has a solution, then* FINDCORE^E *correctly computes the core of a canonical universal solution in time $O(|\text{dom}(S)|^b)$ for some b that depends only on $\Sigma_{st} \cup \Sigma_t$.*

4.3 Discussion

The FINDCORE and FINDCORE^E algorithm look very similar in structure and have essentially the same asymptotic worst-case behavior (see Theorem 3.2.4 and 4.2.3). Nevertheless, there are some fundamental differences between these two algorithms, which we want to outline in this section.

Canonical solution vs. core. In our implementation of the FINDCORE^E algorithm, the chase first produces a solution of the data exchange problem, while the core computation is considered as an optional add-on. In many cases, a canonical universal solution is already a viable option for materialization. Since the minimization step is usually much more complex than the chase, it might be desirable first to obtain an already usable canonical universal solution, and to defer the costly core computation to a later time, e.g., to periods of low database user activity. Moreover, as was already mentioned in Chapter 1, there are data exchange semantics [Lib06] that favor the materialization of canonical universal solutions rather than cores. To support these scenarios, it is important that solving a data exchange problem and computing a core are two cleanly separated steps – which is the case with FINDCORE^E. In contrast, the FINDCORE algorithm never computes a canonical solution; the core computation thus becomes an integral part of solving the data exchange problem.

Chase order. The FINDCORE^E algorithm applies the target chase in a don't care non-deterministic manner, while the FINDCORE algorithm requires a particular chase order referred to as "nice chase". In theory, the additional cost of the nice chase is just a multiplicative constant (before each rule application, we ensure that no rule with higher priority applies, every time checking some fixed number of preconditions). In practice, however, there are other concerns. In our implementation, we take advantage of the fact that the non-deterministic chase can be run in a kind of "batch mode", where several instantiations of a single dependency can be enforced simultaneously. In contrast, if a nice chase order has to be followed, then a separate statement has to be issued for every individual rule application.

Simulation of the EGDs by TGDs. In order to simulate the EGDs in Σ_t, the FINDCORE algorithm has to introduce TGDs which ultimately enforce the desired equalities along the core computation. In theory, the number of additional TGDs needed is bounded by a constant. Nevertheless, the extra effort thus needed can be considerable, as the following simple example illustrates:

Example 4.3.1 *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 [GN06]:*

1. $R(z, v), P(v, z) \rightarrow E(z, v)$
2. $E(x, y) \rightarrow E(y, x)$
3. $E(x, y), E(y, z) \rightarrow E(x, z)$
4. $R(x, y) \rightarrow E(x, x)$
5. $R(x, y) \rightarrow E(y, y)$
6. $P(x, y) \rightarrow E(x, x)$
7. $P(x, y) \rightarrow E(y, y)$
8. $R(x, y), E(x, z) \rightarrow R(z, y)$
9. $R(x, y), E(y, z) \rightarrow R(x, z)$
10. $P(x, y), E(x, z) \rightarrow P(z, y)$
11. $P(x, y), E(y, z) \rightarrow P(x, z)$

where E is the auxiliary predicate representing equality.

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

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

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

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

On the other hand, if EGDs are directly enforced by the target chase, then the chase ends with the canonical universal solution $J^{\Sigma_t} = \{R(x, x), P(x, x)\}$. In this case, the core computation is only needed to check that this solution is already a core.

The most common use of EGDs is, perhaps, to enforce functional dependencies and, in particular, key constraints. That is, given two facts with equal keys, unify the rest of the fields. This is exactly the case when the simulation approach causes the blow-up of a target database, as demonstrated by the above example. Additional TGDs and redundant tuples thus generated have a negative effect on the performance of both the chase and the core computation.

Chapter 5

Algorithm Implementation

We have implemented a prototype system based on the FINDCORE^E algorithm presented in Chapter 4, relying on a DBMS back-end. 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 the freely available HSQLDB and PostgreSQL, but it can be easily adapted to any other RDBMS.

5.1 General system architecture

The overall system architecture is shown on Figure 5.1. The data exchange scenario, namely, the source and target databases, the target schema and the dependencies are specified with an XML configuration file. The source schema is fetched from a database.

The XML configuration data is passed to a Java program, which uses XSLT templates to automatically generate the SQL-statements for managing the target database (creating tables and views, transferring data between tables etc.). Every SQL command can be customized for the specific database platform if it uses an SQL dialect different from the default implementation (the generation of SQL from the configuration files is explained in the Section 5.3 “Implementation”). For the source data, every JDBC-compatible data source is fine.

Our prototype provides no graphical interface, since the use of a database allows to track data transfers with the help of any graphical database management tool, that supports JDBC. The configuration can be performed with a single XML file.

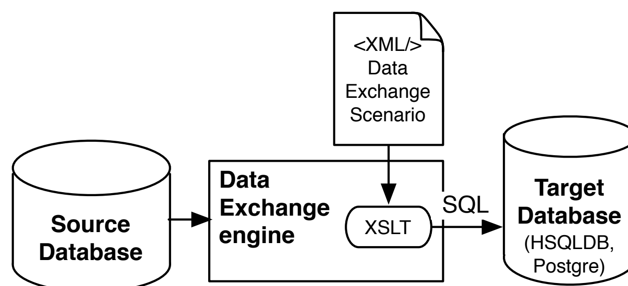


Figure 5.1: Overview of the implementation.

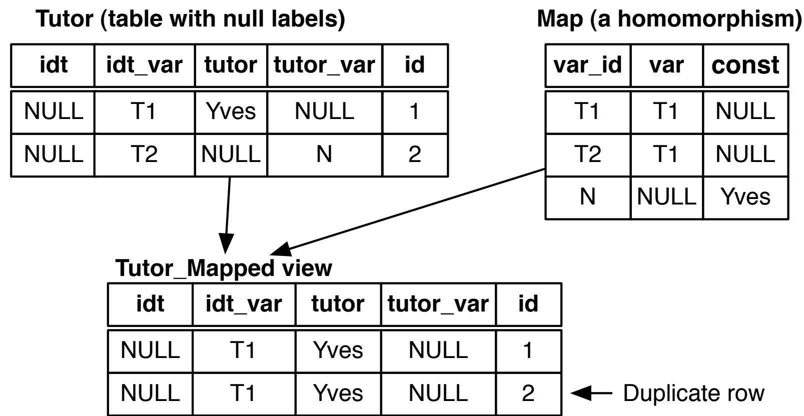


Figure 5.2: Modeling labeled nulls.

5.2 Design decisions

In this section we provide an overview of the system design, clarifying the main decisions, but without going too much into detail, which is the next *Implementation* section responsible for. We explain the system based on our running EXAMPLE 2.3.1. Recall that we have a source schema containing of the two relations Tutorial(course, tutor): {('java', 'Yves')} and BasicUnit(course): {'java'}, and the target schema of four relation symbols NeedsLab(id_tutor,lab), Tutor(idt,tutor), Teaches(id_tutor, id_course) and Course(idc, course). The STDs are:

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

The target dependencies consist of two TGDs and one EGD:

3. $\mathsf{Course}(Idc, C) \rightarrow \mathsf{Tutor}(Idt, T), \mathsf{Teaches}(Idt, Idc)$.
4. $\mathsf{Teaches}(Idt, Idc) \rightarrow \mathsf{NeedsLab}(Idt, L)$.
5. $\mathsf{Course}(Id_1, Course), \mathsf{Course}(Id_2, Course) \rightarrow Id_1 = Id_2$.

5.2.1 Labeled nulls support

None of the common database management systems to-date support labeled nulls. Therefore, to implement this feature on top of the usual RDBMS, we augmented 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. Fig. 5.2 gives a flavor of what this part of the database looks like. Each target relation has a unique `id` attribute, which is not present anywhere in the dependencies, but is used internally to reference the facts (i.e., the table rows). The values for the `id` attribute are generated by the usual database auto-increment mechanism and are not part of the target domain considered for core computation.

The relation under a homomorphism is simulated by a view. To implement it, it suffices to perform an outer join for each field of the relation with the `Map` table, and then glue the the column mappings together with the help of inner join on the `id` field.

We give an example definition of `Tutor_Mapped` view that simulates a `Tutor` relation under a homomorphism stored in the table `Map`.

```
CREATE Tutor_Mapped AS
SELECT COALESCE(Tutor1.idt, Map1.const), Map1.var,
        COALESCE(Tutor2.tutor, Map2.const), Map2.var,
        Tutor1.id
FROMTutor AS Tutor1 LEFT JOIN Map AS Map1 ON Tutor.idt_var = Map1.var_id
      INNER JOIN Tutor AS Tutor2 ON Tutor1.id = Tutor2.id
      LEFT JOIN Map AS Map2 ON Tutor2.tutor_var = Map2.var_id;
```

Our approach of simulating labeled nulls in a "normal" RDBMS is probably not the most efficient solution and, thus, not suitable for real-world applications (especially, when the number of columns is large). However, the flexibility of such a design proved very convenient for the prototype development.

5.2.2 Blocks and disjoint sets

Block computation problem arises at different steps of the FINDCORE^E algorithm. Moreover, in this chapter we show, that some other tasks which are necessary for algorithm implementation can be reduced to block computation, so it becomes one of the central routines of the system.

One approach to compute the connected components of undirected graphs is to reduce it to the well-known *set union* problem [GF64]. It consists of maintaining a collection of disjoint sets under the operation of union. That is, starting with some initial collection of disjoint sets, perform one of the two operation on them:

- Union(A,B) merges the sets A and B in one,
- Find(x) returns the identifier of a unique set containing x.

Additionally, one can introduce an initialization operation, MakeSet(x) that constructs a singleton set given an element.

Let now start with the elementary sets containing the variables of some instance K. Perform Union (Find(x), Find(y)) whenever variables x and y occur together in the same tuple, for all pairs of variables in $\text{var}(K)$. Having done this for every pair of variables of each tuple in K, the disjoint set of a variable is exactly its block. Alternatively, we could build the sets over tuples and join them whenever two tuples have a variable in common. We illustrate this in the EXAMPLE 5.2.1.

We refer to the algorithm allowing to maintain such a structure as to DISJOINTSETS and give only its outline here. The detailed description and discussion of this technique is contained, for example, in [GI91] and in [RL90].

To implement the find and union operation effectively, the disjoint sets are represented as trees. The root of the tree serves a set identifier. The Find operation is then just following the parent references up to the root element. And to combine two disjoint sets it suffices to update the parent reference of the root of one tree, so that it become a branch of another tree.

```
function MakeSet(x)
    x.parent = x;
```

```
function Find(x)
    if x.parent == x
        return x;
    else
        return Find(x.parent);

function Union(x,y)
    xRoot = Find(x);
    yRoot = Find(y);
    xRoot.parent = yRoot;
```

This is rather a naive approach, since the trees can become highly unbalanced after a series of random Uion operations. Two simple improvements allow to tackle this: firstly, take care to append the smaller trees to larger ones and not vice versa. Secondly, a side effect of Find operation is introduced: namely, it flattens the tree while traversing it. The last improvement is called *path compression* and it is quite easy to implement:

```
function Find(x)
    if x.parent == x
        return x;
    else
        x.parent = Find(x.parent);
        return x.parent;
```

Back to the first improvement, we need to effectively estimate the tree size. To do so, the *rank heuristic* can be used:

```
function MakeSet(x)
    x.parent = x;
    x.rank = 0;

function Union(x,y)
    xRoot = Find(x);
    yRoot = Find(y);
    if xRoot.rank > yRoot.rank
        yRoot.parent = xRoot;
    elseif xRoot.rank < yRoot.rank
        xRoot.parent = yRoot;
    elseif xRoot.rank == yRoot.rank
        yRoot.parent = xRoot;
        xRoot.rank = xRoot.rank + 1;
```

In the worst case, the running time per operation in this approach is $O(A(n,n)^{-1})$ [RL90], where $A(n,n)$ is an extremely quickly-growing Ackermann function. It was shown in [FS89] that *any* disjoint sets algorithm representing collections of sets as forests, accesses $\Omega(A(n,n)^{-1})$ words per operation, so it is hardly possible to obtain a more effective algorithm.

Example 5.2.1 *We illustrate the usage of the* DisjointSets *algorithm by the following example. Let* $K = \{S(x,y), P(y,z), S(v,w)\}$ *be an instance. Suppose that we want to check if there exists an endomorphism* h *of* K *mapping* x *onto* v*. To do so, we need to check if it is possible to construct an endomorphism, mapping all tuples over variables of* $\text{block}(x)$ *onto tuples of* K *over* $h(\text{block}(X))$*. That is, we need to select the facts of* K *which are relevant for checking the existence of a homomorphism.*

We start with three singletons, constructed by MakeSet*:* $\{S(x,y)\}, \{P(y,z)\}$ *and* $\{S(v,z)\}$*. Then we call* $\mathsf{Union}\,(\mathsf{Find}(t_1), \mathsf{Find}(t_2))$ *for each pair* $\langle t_1, t_2\rangle$ *of facts, whenever* t_1 *and* t_2 *share a variable, but belong to different sets, i.e.,* $\mathsf{Find}(t_1) \neq \mathsf{Find}(t_2)$*.*

Let us start with the pair $\langle S(x,y), P(y,z)\rangle$*. The two facts have a common variable* y *but belong to different sets, so the* Union *operation yields the superset* $\{S(x,y), P(y,z)\}$*. Then, after processing the pair* $\langle S(v,z), P(y,z)\rangle$*, all the three facts become elements of the same (trivially disjoint) set. Hence, one needs to consider all the facts of* K *to check if a desired endomorphism exists. It is easy to see that the result is negative, since* y *has to be mapped on* z*, and there is no such* P*-fact in* K*.*

5.2.3 Unlabeled nulls in the source instance

The FindCore^E algorithm, as well as FindCore, is defined on ground source instances only. The reason for that is that the bounded block size of the source instance is critical for the tractability of core computation, as shown in previous chapters. Since our implementation takes the data from arbitrary `JDBC` databases, we opted to make this quite restrictive requirement a little weaker. That is, we allow the usual, not labeled nulls to appear in the source instance. Source-to-target chase then assigns a unique label to every such non-labeled null, should it be copied from the source into the target database. Clearly, this affects the preuniversal instance block size, which nonetheless remains fixed and being determined by the source-to-target dependencies only. More precisely, if h is the height of STDs, and w — their width, then by allowing unlabeled source nulls we increase the maximal block size of the preuniversal instance from h to $h + w$.

Additional argument in favor of such a decision is that it facilitates the performance evaluation. By adding the *incomplete duplicates* of the source tuples to the source instance, one can easily vary the load on the core computation. By *incomplete duplicate* we understand a variant of the tuple with some values replaced by a `NULL`. For example, let Tutorial('Yves','java') be the ground fact. Then Tutorial('Yves',NULL), Tutorial(NULL,'java') are both its incomplete duplicates.

Since the target counterparts of such incomplete source tuples, produced by the source-to-target chase, are guaranteed to be homomorphic to the facts induced by the original source tuples without nulls, the former can never appear in the core of the canonical solution. We used this approach for experimenting with the system, however, it could be a viable option for implementing in any data exchange tool, since `NULL` values are quite common in real-world databases.

5.2.4 Partitions

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 (thus, in our example it makes no sense to try to unify a variable from the `Tutor.tutor` column with any term from `Course.idc`). 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 either

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

All the three rules partition the fields occurring in an individual dependency (let us call them *local* partitions). To compute the global partitions from the local ones, it suffices to feed them to the above described DisjointSets algorithm. Back to our example, the target field partitions are

1. {`Course.course`},
2. {`Tutor.tutor`},
3. {`NeedsLab.lab`},
4. {`Course.idc, Teaches.id_course`} and
5. {`Tutor.idt, Teaches.id_tutor, NeedsLab.id_tutor`}.

5.2.5 Chase

Chase means, essentially, the search of term assignments leading to constraint violation, and the usage of those found terms (together with fresh nulls, in case of non-full TGDs) for repairing the target instance, so that the constraints became satisfied. Since the premise of every constraint is a query, it is natural to convert it in SQL and pass to a DBMS to retrieve such violating assignments.

For example, consider the source instance of our example: { Tutorial('java', 'Yves'), BasicUnit('java') }, and suppose that the target instance is empty. Then the following SQL query, representing the first STD

$$\mathsf{BasicUnit}(C) \rightarrow \mathsf{Course}(Idc, C)$$

retrieves a unary tuple ('java'):

```
SELECT BasicUnit.course FROM BasicUnit
WHERE NOT EXISTS (SELECT 1 FROM Course WHERE Course.course = BasicUnit.course )
```

As a slightly more complex case, we take the second STD:

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

The corresponding SQL query returns ('java', 'Yves'), as expected:

```
SELECT Tutorial.course, Tutorial.tutor FROM Tutorial
WHERE NOT EXISTS (SELECT 1 FROM Course
        JOIN Teaches ON Course.idc = Teaches.id_course
        JOIN Tutor ON Teaches.id_tutor = Tutor.idt
        WHERE Course.course = Tutorial.course AND Tutor.tutor = Tutorial.tutor)
```

The results of the above queries are then used to generate new facts in the target instance. If executed one after another, they yield the following target preuniversal instance:

J = {Course(C_1, 'java'), Course(C_2, 'java'), Tutor(T_1,'Yves'), Teaches(T_1,C_2) }

Since target TGDs are enforced essentially in the same way as source-to-target ones, we omit the first two target dependencies, and illustrate the application of EGDs, based on the same idea of SQL representation. The only thing that is different for target dependencies, is how the equality between terms is expressed: two terms are equal if they represent the same constant, or if they are both nulls with the same label. This differs from the usual SQL behavior, when two NULLs are never equal (in fact, any logical operation over a NULL operand must return "undefined", i.e. is neither true nor false). We first give the query retrieving all pairs of terms that violate the last target dependency

$$\mathsf{Course}(Id_1, Course), \mathsf{Course}(Id_2, Course) \rightarrow Id_1 = Id_2 :$$

```
SELECT Course1.course, Course1.course_var, Course2.course, Course2.course_var
FROM Course AS Course1 JOIN Course AS Course1 ON Course1.course = Course2.course
WHERE ( (Course1.idc IS NULL) != (Course2.idc IS NULL) )
        OR Course1.idc != Course2.idc OR Course1.idc_var != Course2.idc_var
```

We cannot, of course, just write Course1.idc != Course2.idc **OR** Course1.idc_var != Course2.idc_var to test that two values are distinct, because of the special treatment of NULL values by most database systems: when testing a variable (Course1.idc is NULL and Course1.idc_var is not null) and a constant (Course2.idc is not null and Course2.idc_var is NULL) both inequalities retrun "undefined", and the search condition fails, leaving the not equal pair undetected.

Note that the above queries retrieve *all* the assignments violating a given dependency. Thus, so to say, a "batch mode" of dependency enforcement is implemented (we may do so, as the success of the chase does not depend on the order of rules application: this would clearly be impossible with the nice chase proposed in [GN06]).

There is an additional subtle aspect to consider, when using such a batch enforcement of EGDs. Consider the EGD of our example and the following instance:

$K = \{$Course(C_1, 'java'), Course(C_2, 'java'), Course(C_3, 'java')$\}$

The above SQL query will retrieve the all three IDs in all possible combinations: $\{(C_1, C_2), (C_1, C_3), (C_2, C_3), (C_2, C_1), (C_3, C_1), (C_3, C_2)\}$. After pruning the symmetric pairs, we are left with $\{(C_1, C_2), (C_2, C_3), (C_3, C_1)\}$. Clearly, if performed independently, the three substitutions would lead just to different distribution of the same nulls among the facts.

One solution allowing to prevent such circular assignments is first to find the set of the terms E which are equal according to some EGD τ (there may be no more than one constant in E, otherwise chase must fail), then choose one element $t_s \in E$ to serve a substitution (obviously, if there is a constant in E, it should be chosen), and replace every variable in $E \setminus \{t_s\}$ with t_s. There are different ways of identifying such sets of equal terms, and one of them is to view the pairs of terms as edges of the graph, while terms being its vertices (it is thus not a Gaifman graph, as we allow constants to serve as vertices). Its connected components are then the terms that should be set equal. Quite naturally, to compute these connected components we use the Disjoint-Sets algorithm described above, similarly to computing blocks and partitions.

During the chase, it is also necessary to store some additional information. In particular, we need

- track variables and origin of their positions (see Section 5.2.7 for implementation details), which is needed to construct the subinstance T_{xy}, as described in LEMMA 4.2.1;
- chase history required for the homomorphism extension, according to the procedure EXTEND from Chapter 4 (see Section 5.2.9)

5.2.6 Computing non-rigid blocks

We use the `Nrblock(blockid, varid)` table to store the non-rigid blocks, where the `blockid` field stores the block identifier, and the `varid` — variables belonging to the block. To compute them, it suffices

1. during the source-to-target chase, create a new block in the `Nrblock` table for each TGD firing, and add to it all the introduced variables;
2. reflect in `Nrblock` each variable replacement caused by a target EGD (i.e., substitute the values in `varid` field with the same term as in target tables);
3. after the target chase, mark all the variables that occur more than in one block of `Nrblock` as rigid (use an appropriate bit field in the `Var` table). Delete rigid variables from `Nrblock`.

To illustrate this process, let us step through the target chase (the creation of the preuniversal instance was already demonstrated in the previous subsection). We start with a preuniversal instance

$J = \{$Course(C_1, 'java'), Course(C_2, 'java'), Tutor(T_1,'Yves'), Teaches(T_1,C_2) $\}$

First, we compute the blocks of J. They are: $\{\{C_1\}, \{C_2, T_1\}\}$. The first TDG

$$\mathsf{Course}(Idc, C) \rightarrow \mathsf{Tutor}(Idt, T), \mathsf{Teaches}(Idt, Idc)$$

generates another pair of $Tutor$ and $Teaches$ facts for the course C_1: Tutor(T_2,N) and Teaches(T_2,C_1). Then, the second TGD

$$\mathsf{Teaches}(Idt, Idc) \rightarrow \mathsf{NeedsLab}(Idt, L)$$

adds two facts NeedsLab(T_1,L_1) and NeedsLab(T_2,L_2). Finally, the EGD sets the C_1 and C_2 equal, which is reflected in the blocks $\{\{C_1\}, \{C_1, T_1\}\}$. Since C_1 occurs more than in one block (of course, we could have left C_2 instead), we mark it as rigid, and exclude from the non-rigid blocks citizens. Thus, the only non-rigid block left is $\{T_1\}$, which is trivial and can be ignored. A canonical universal solution is:

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

5.2.7 Tracking variable families

As explained in the Chapter 4, the family of a variable is the minimal set of facts, including its origin and closed under parents and siblings. As shown by the Lemma 4.1.1, every position of an instance always has the same contents as its origin. Thus, if we think of facts as of position vectors (or, better yet, of variable placeholder vectors), we can also regard a foreign position p as another instantiation of its origin position o_p.

For the implementation, we found this idea more intuitive, and moreover, more effective, than tracking an origin of every position. To avoid overloading of the word "position" with yet another meaning of a variable placeholder, we call them *cells*. Now, when a fresh null x is introduced by the chase, we create a new cell c_x for it, and say that this cell is used by all facts that were created to store x, i.e.,every origin position of x is identified with c_x; we also call them origin positions of a cell c_x. When later a position p_s, with which a cell c_{p_s} is identified, serves as a source for some position p (i.e. a value from p_s is copied into p by a TGD), we identify with p the same cell c_{p_s}. Hence, foreign positions are always identified with the same cell as their origin positions. To track the variable of the target instance, we use the following tables (Fig. 5.3):

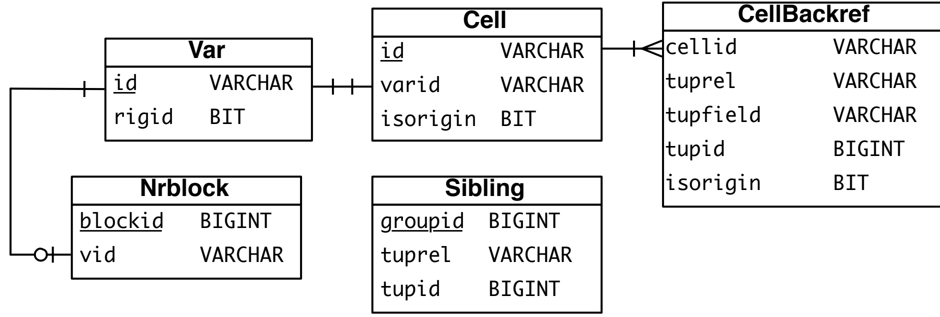


Figure 5.3: Database representation of cells.

- `Var(id VARCHAR, rigid BIT)`, where `id` stores a unique variable id (we use a character data type to facilitate debugging: it is easier to track visually character identifiers in the database than numbers), `rigid` is true for rigid variables only.
- `Cell(id VARCHAR, varid VARCHAR, isorigin BIT)`, where `Cell.isorigin` is set to true initially, when a new cell is created to store a fresh variable; if a variable is replaced with another variable by an EGD, the respective `Cell.isorigin` field is set to false.
- `CellBackref(cellid VARCHAR, tuprel VARCHAR, tupfield VARCHAR, tupid BIGINT, isorigin BIT)` enumerating all occurrences of a cell `cellid`. Occurrences are represented by a triple: a relation name `tuprel`, a relation field `tupfield` and a tuple id `tupid`. `CellBackref.isorigin` is true for origin positions (occurrences) of the cells, and false otherwise.
- `Sibling(groupid BIGINT, tuprel VARCHAR, tupid BIGINT)` storing the sibling relation over facts. Each group of siblings has a unique identifier `cellid`. Facts (tuples) are referenced by the relation name `tuprel`, and a row id `tupid`.
- `Nrblock(blockid BIGINT, vid VARCHAR)` enumerates a non-rigid blocks over the facts belonging to the preuniversal instance.

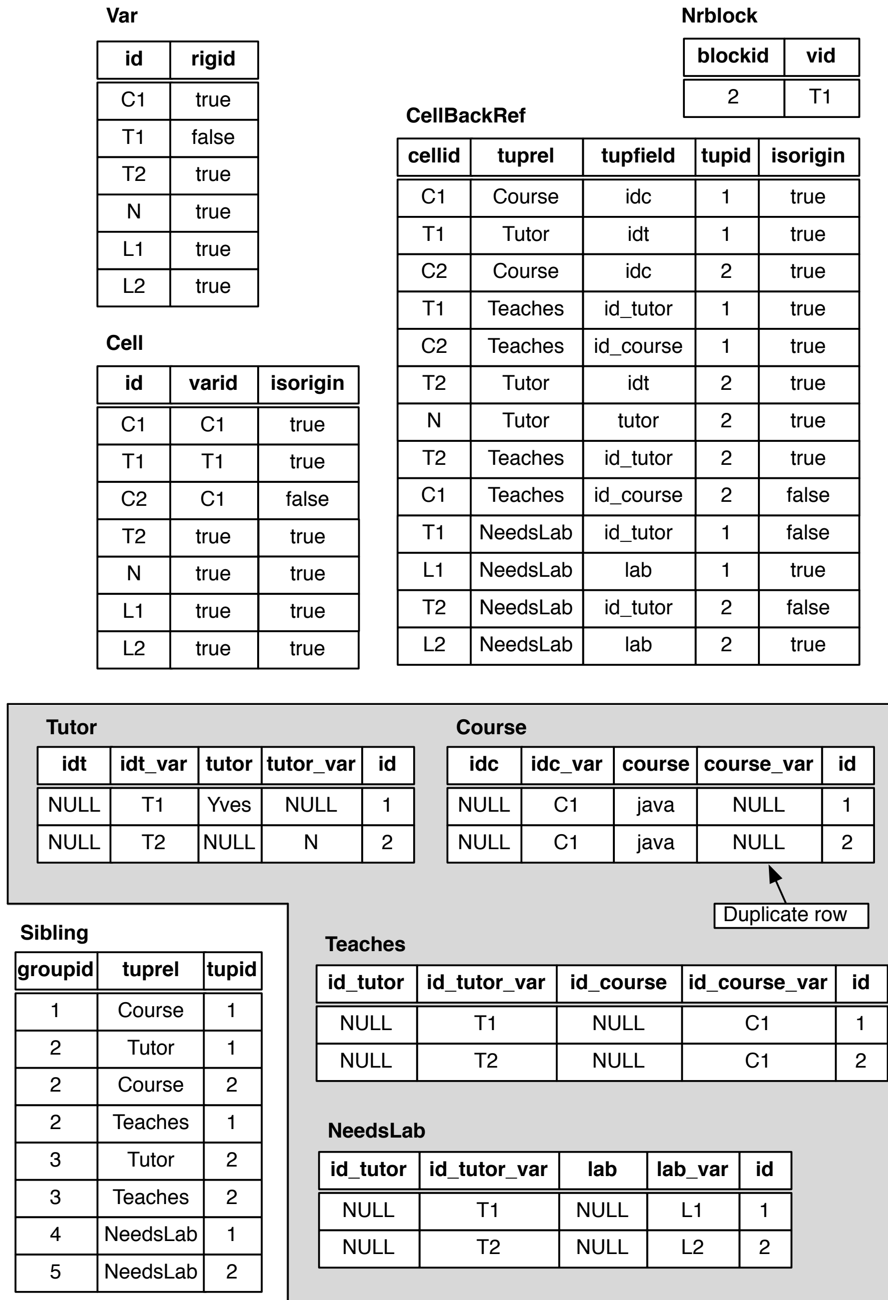
Let us recall our running example. A canonical universal solution is

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

Fig. 5.4 shows the snapshot of tables in the target database, representing this instance. Now, let us get the family of variable N, i.e. the set of facts F_N.

1. Fetch the origin cell c_N of variable N (initially, cell names coincide with the labels of nulls stored in them, until an EGD changes the cell's contents).
2. Fetch from `CellBackRef` and store in F_N the origin facts of the cell c_N and their siblings: $F_N := \{(\mathsf{Tutor},2),\ (\mathsf{Teaches},2)\}$. Let C_N be a set of cells of F_N: $C_N = \{N, T_2, C_1\}$.
3. Fetch from `CellBackRef` and add to F_N the origin facts of the cells in C_N, which are not yet there. Repeat this step until there are such facts. $F_N := F_N \cup \{(\mathsf{Course},1)\}$.

Finally, we obtain the family of N: $F_N = \{(\mathsf{Tutor},2),\ (\mathsf{Teaches},2),\ (\mathsf{Course},1)\}$, that is, the facts `Tutor(`T_2`,`N`)`, `Teaches(`T_2`,`C_1`)`, and `Course(`C_1`,'java')`.



Var

id	rigid
C1	true
T1	false
T2	true
N	true
L1	true
L2	true

Nrblock

blockid	vid
2	T1

CellBackRef

cellid	tuprel	tupfield	tupid	isorigin
C1	Course	idc	1	true
T1	Tutor	idt	1	true
C2	Course	idc	2	true
T1	Teaches	id_tutor	1	true
C2	Teaches	id_course	1	true
T2	Tutor	idt	2	true
N	Tutor	tutor	2	true
T2	Teaches	id_tutor	2	true
C1	Teaches	id_course	2	false
T1	NeedsLab	id_tutor	1	false
L1	NeedsLab	lab	1	true
T2	NeedsLab	id_tutor	2	false
L2	NeedsLab	lab	2	true

Cell

id	varid	isorigin
C1	C1	true
T1	T1	true
C2	C1	false
T2	true	true
N	true	true
L1	true	true
L2	true	true

Tutor

idt	idt_var	tutor	tutor_var	id
NULL	T1	Yves	NULL	1
NULL	T2	NULL	N	2

Course

idc	idc_var	course	course_var	id
NULL	C1	java	NULL	1
NULL	C1	java	NULL	2

Sibling

groupid	tuprel	tupid
1	Course	1
2	Tutor	1
2	Course	2
2	Teaches	1
3	Tutor	2
3	Teaches	2
4	NeedsLab	1
5	NeedsLab	2

Teaches

id_tutor	id_tutor_var	id_course	id_course_var	id
NULL	T1	NULL	C1	1
NULL	T2	NULL	C1	2

NeedsLab

id_tutor	id_tutor_var	lab	lab_var	id
NULL	T1	NULL	L1	1
NULL	T2	NULL	L2	2

Figure 5.4: Database representation of the target instance.

5.2.8 Computing a characteristic homomorphism

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 $h : T_{xy} \to U$, 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 instantiations for the variables in each block.

We demonstrate that on a canonical universal solution J' obtained previously (see the last paragraph of the previous subsection). Suppose that we look for a proper endomor-

phism h' on J'. Step 4 of FINDCORE^E then yields the set $T_{N,'\mathsf{Yves}'} = J \cup F_N$, where F_N is a family of the variable N found in the previous section, and J — a preuniversal instance (see Section 5.2.5).

$$T_{N,'\mathsf{Yves}'} = \{\texttt{Tutor}(T_1,\texttt{'Yves'}),\ \texttt{Teaches}(T_1,C_1),\ \texttt{Course}(C_1,\texttt{'java'}),\ \texttt{Tutor}(T_2,\ N),\ \texttt{Teaches}(T_2,\ C_1)\}.$$

At step 5, a homomorphism $h\colon T_{xy} \to J$ (with $x = N$ and $y =$'Yves'), s.t. $h(N) =$ 'Yves' has to be found. As we found out, non-rigid blocks of a preuniversal instance can be trivially ignored. Moreover, we disregard the rigid variable C_1 when calculating non-rigid blocks of $T_{N,'\mathsf{Yves}'}$.

The usual block of N in $T_{N,'\mathsf{Yves}'}$ is $\{N, T_2, C_1, T_1\}$, since there are edges (T_2, N) and (T_2, C_1) in the Gaifman graph of J'. The non-rigid block is induced by a non-rigid Gaifman graph with no C_1 vertex, thus the non-rigid block of N is $\{N, T_2\}$. The following SQL query returns all possible instantiations of the variable T_2 compatible with the mapping $h(N) = '\mathsf{Yves}'$:

```
SELECT Tutor.idt_var AS T2 FROM Tutor WHERE Tutor.tutor = 'Yves'
```

In our example, the result is $\{T_2 \to T_1\}$. The rest of the variables of $T_{N,'\mathsf{Yves}'}$, namely T_1 and C_1, is mapped using the following considerations:

- C_1 is rigid, hence the only possible mapping is $C_1 \to C_1$;
- T_1 do not belong to the non-rigid block of N, hence can be mapped independently. In particular, we may reuse the endomorphism of J, computed on the previous iteration of the algorithm. We assume, that there were a zero iteration, producing an identity endomorphism of J, and the roughest core approximation, which is J itself. Hence, we use this identity mapping for T_1, setting $T_1 \to T_1$.

Note, that excluding of the rigid variable have led to a notable simplification of the query. Imagine the same example without the target EGD. We would obtain a canonical universal solution

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

Similarly, trying to get rid of N, we should have used its non-rigid block in $T_{N,'\mathsf{Yves}'}$, which in the absence of EGD coincides with the usual block of N: $\{N, T_2, C_1, T_1\}$. The corresponding SQL query would then contain several joins:

```
SELECT Tutor1.idt_var AS T2, Course.idc_var AS C1, Tutor2.idt_var AS T1
FROM Tutor AS Tutor1
        JOIN Teaches AS Teaches1 ON Tutor1.idt_var = Teaches1.id_tutor_var
        JOIN Course ON Teaches1.id_course_var = Course.idc_var
        JOIN Teaches AS Teaches2 ON Teaches2.id_course_var = Course.idc_var
        JOIN Tutor AS Tutor2 ON Teaches2.id_tutor_var = Tutor2.idt_var
WHERE Tutor1.tutor = 'Yves' AND Course.course = 'java' AND Tutor2.tutor = 'Yves'
```

The above query leads to the same mappings of variables, but in a far less efficient way (especially if a database is large). This example shows, that due to rigidity property, EGDs can make the homomorphism computation remarkably faster. This would be impossible with the approach of EGD simulation, employed by the original FINDCORE.

5.2.9 Homomorphism extension

Extension of the homomorphism at step 7 of the FindCore^E, proceeds according to the iterative procedure Extend (Section 4.2). At each step, we choose a set of sibling facts S which contain variables not yet covered by a homomorphism, such that all parents of facts in S are fully covered. Then we extend the homomorphism to cover tuples in S as well.

It is not difficult to show, that if we follow the order in which the non-full TGDs were applied by the target chase, we can use not the parent facts of S, but the facts, that satisfied the premise of TGD which in turn led to the creation of siblings S.

Therefore, for each chase step enforcing a non-full TGD, we track the ids of facts that have satisfied the dependency, and the ids of facts that have been created. Fig. 5.5 shows such chase logs for the two target TGDs of our running example.

Log of the first target TGD (dependency #3):

Course(Idc,C) → Tutor(Idt,T),Teaches(T,C)

ChaseLog_3

id_parent_course	id_tutor	id_teaches
1	2	2

Log of the dependency #4:

Teaches(Idt,Idc) → NeedsLab(Idt,L)

ChaseLog_4

id_parent_teaches	id_needslab
1	1
2	2

Figure 5.5: Logs of non-full TGD application.

Having such a log, it is not difficult to choose a pair of fact sets P and S, such that the creation of S was triggered by the facts in P, and all variables of P are covered by a homomorpism, but not all variables of S are. Then, we use the same approach as in the previous section, to find the mappings for the variables of S, consistent with already found mappings for P.

For our example, we know that $\{C_1 \rightarrow C_1, N \rightarrow \text{'Yves'}, T_2 \rightarrow T_1, T_1 \rightarrow T_1\}$ is a valid homomorphism $T_{N,'Yves'} \rightarrow J'$. To obtain an endomorphism on J', it suffices to map the variables L_1 and L_2, generated by the dependency #4. The following SQL query finds an image for variable L_1 consistent with the previously found mappings $T_1 \rightarrow T_1$ and $C_1 \rightarrow C_1$:

```
SELECT NeedsLab.lab_var AS L1
FROM NeedsLab JOIN Teaches ON NeedsLab.id_tutor_var = Teaches.id_tutor_var
WHERE Teaches.id_tutor_var = 'T1' AND Teaches.id_course_var = 'C1'
```

The query returns L_1, as expected. The mapping $L_2 \rightarrow L_1$, retrieved with the same query (since T_2 is also mapped on T_1), finalizes the creation of a proper endomorphism of J.

5.2.10 Choosing the optimal mapping

As shown previously, to construct or to extend a homomorphism, SQL queries are used, retrieving all possible mappings for a set of variables. If several such mappings are possible, how can we choose the optimal one? Let h be a homomorphism to be built (or extended). W.l.o.g. assume that we already know some its part $h_i \subseteq h$ (it is evident for homomorphism extension; if h is constructed from scratch, it is done blockwise, and h_i could be a mapping

for some already processed block). To choose between several possible images, we use the following simple heuristic: if two mappings $\vec{a}$ and $\vec{b}$ are possible for some tuple of variables $\vec{x} \in \text{dom}(h)$, we choose $\vec{a}$, whenever

- number of constants in $\vec{a}$ is greater than in $\vec{b}$ (*prefer constants*), or
- number of terms, belonging to the $range(h_i)$ in $\vec{a}$ is greater than in $\vec{b}$ (*reuse images*).

As shown in the next chapter ("System tests and evaluation"), these two simple heuristic rules often suffice to obtain a core after the first successful iteration of the FindCore^E (that is, after an iteration resulting in creation of a proper retraction).

5.2.11 Obtaining the retraction

According to the Theorem 3.2.2, to obtain a retraction r from an arbitrary endomorphism h on some instance B, it suffices first to iterate h some $q < |\text{dom}(B)|$ times until the fix-point is reached, i.e. $h^q(B) = B$. Then, the $g = h^q$ is iterated another c times, where c is polynomial in the $|\text{dom}(B)|$, as the following procedure shows:

Procedure TransformToRetraction

Input: Endomorphism h on some instance B
Output: Retraction r on B s.t. $\forall x, y \in \text{dom}(B)\colon h(x) = h(y)$ implies $r(x) = r(y)$

(1) Set $g := h$;
(2) **while** $|\text{range}(g)| > |\text{range}(h \circ g)|$ **do**
(3) Set $g := h \circ g$
(4) **od**
(5) Set $r := g$
(6) **while** $\exists x \in \text{dom}(r) : r(x) \neq r(r(x))$ **do**
(7) Set $r := g \circ r$
(8) **od**
(9) **return** r.

The correctness of this algorithm follows immediately from the Theorem 3.2.2. The second iteration could have been implemented more efficiently (now we compute g^c performing c iterations; however, Theorem 3.2.2 shows that c could be factored on some k multipliers c_i, $1 \leq i \leq k$; it then suffices to perform $\sum_{1 \leq i \leq k} c_i$ compositions). However, our experiments showed that in practice this part of an algorithm is by far not a bottleneck (taking at least by two orders of magnitude less time than the search and extension of the homomorphism on previous steps of FindCore^E), so we have found improving this procedure unnecessary.

5.3 Implementation

5.3.1 Database manipulation

For database manipulation, like copying of data from source to target, target schema creation, updating and deletion, we used an open-source library, Apache DDL Utils.

Therefore, the target schema must be specified in the XML format used by this library, which is called Turbine XML or Torque XML in the Apache documentation. Though it

supports different types of database objects, as indexes and constraints, we use only the most basic constructions, for specifying table and field names. It is also possible to provide more information within a database schema (like different field data types, public and foreign keys etc.), though everything but table and field names will be ignored. Moreover, only string (`VARCHAR`) fields are currently supported in the target tables. The following XML file represents the target schema in our running example:

```
<!DOCTYPE database SYSTEM "http://db.apache.org/torque/dtd/database.dtd">

<database name="default-source-databse">
    <table name="Course">
        <column name="idc"/>
        <column name="course"/>
    </table>

    <table name="Tutor">
        <column name="idt"/>
        <column name="tutor"/>
    </table>

    <table name="Teaches">
        <column name="id_tutor"/>
        <column name="id_course"/>
    </table>

    <table name="NeedsLab">
        <column name="id_tutor"/>
        <column name="lab"/>
    </table>
</database>
```

The source schema is normally must not be specified, as Apache DDL Utils can obtain it by means of JDBC. We use this library also for creation and updating the target database schema, and for transferring the data between source and target databases. For other operations on the target database, classes of the Spring Framework [spr] are used, which is much more convenient than the standard Java JDBC implementation. The use of Spring Framework for system configuration is further explained in Section 5.3.2.

5.3.2 Configuration

For specifying data exchange scenarios, we use XML configuration files. The schema of the source and target database as well as the STDs and TDs can thus be comfortably represented, and they are cleanly separated from the scenario-independent Java code of the core computation.

For instance, for our running example of this chapter, the source-to-target dependencies

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

are represented by the following XML code fragment:

```
<dx:sourceToTargetDependencies>
    <dx:dependency>
        <dx:premise>
            <BasicUnit course="C"/>
        </dx:premise>
        <dx:conclusion>
            <Course course="C"/>
        </dx:conclusion>
    </dx:dependency>
    <dx:dependency>
        <dx:premise>
            <Tutorial tutor="T" course="C"/>
        </dx:premise>
        <dx:conclusion>
            <Course idc="Idc" course="C"/>
            <Tutor idt="Idt" tutor="T"/>
            <Teaches id_tutor="Idt" id_course="Idc"/>
        </dx:conclusion>
    </dx:dependency>
</dx:sourceToTargetDepenencies>
```

Note that for the first TGD, the variable Idc which is unique for the conclusion, has not to be specified. By default, all the omitted fields are instantiated by fresh unique labeled nulls. The XML consists of two schemata: one, denoted by the prefix `"dx"` sets the structure of dependencies, another is used to specify the premise and conclusion queries. The second schema is flexible: the field/value pairs can be represented either as arguments or as elements, or both: `<Tutorial tutor="T" course="C"/>` denotes the same as `<Tutorial tutor="T"> <course>C</course></Tutorial>`. The target dependencies

3. $\mathsf{Course}(Idc, C) \rightarrow \mathsf{Tutor}(Idt, T), \mathsf{Teaches}(Idt, Idc)$,
4. $\mathsf{Teaches}(Idt, Idc) \rightarrow \mathsf{NeedsLab}(Idt, L)$ and
5. $\mathsf{Course}(Id_1, Course), \mathsf{Course}(Id_2, Course) \rightarrow Id_1 = Id_2$

are specified in the same way:

```
<dx:targetDependencies>
    <dx:dependency>
        <dx:premise>
            <Course idc="Idc"/>
        </dx:premise>
        <dx:conclusion>
            <Tutor idt="Idt"/>
            <Teaches id_tutor="Idt" id_course = "Idc"/>
        </dx:conclusion>
    </dx:dependency>
    <dx:dependency>
        <dx:premise>
            <Teaches id_tutor="Idt"/>
        </dx:premise>
        <dx:conclusion>
            <NeedsLab id_tutor="Idt"/>
        </dx:conclusion>
    </dx:dependency>
```

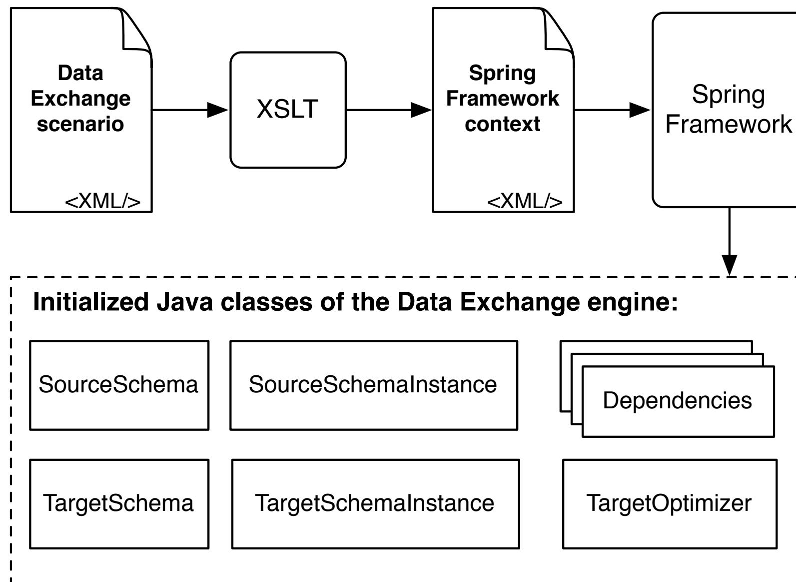


Figure 5.6: Configuration of the system.

```
    <dx:dependency>
        <dx:premise>
            <Course id_course="Id1" course="C"/>
            <Course id_course="Id2" course="C"/>
        </dx:premise>
        <dx:conclusion>
            <dx:eq dx:a="Id1" dx:b="Id2"/>
        </dx:conclusion>
    </dx:dependency>
</dx:targetDepenencies>
```

Also in the premise query, if a variable occurs only once, and also not present in the conclusion it can be omitted (variable C in the TGD #3, and Idc in the TGD #4). Equality is specified by the tag `dx:eq` with two obligartory attributes `dx:a` and `dx:b` for the left- and right-hand sides of the equation respectively.

The free XML schema is convenient for specifying queries, but not for further processing. To facilitate XSL transformations, configuration data is encoded in the internal format, better suited for XSLT. Thus, the last target TGD (dependency #4) looks in the following way in the intermediate XML schema (the attribute `partId` corresponds to partition id: see Section 5.2.4):

```
<dependency id="4">
    <premise>
        <tuples>
            <tuple rel="Teaches">
                <term field="id_tutor" isRelevant="true" partId="5">Idt</term>
                <term field="id_course" isRelevant="false" partId="4">Idc</term>
            </tuple>
        </tuples>
    </premise>
```

```
    <conclusion>
        <tuples>
            <tuple rel="NeedsLab">
                <term field="id_tutor" partId="5">Idt</term>
                <term field="lab" isVar="true" partId="3">Idc</term>
            </tuple>
        </tuples>
    </conclusion>
</dependency>
```

Based on the data schema and dependies, the java classes are constructed and initialized. To facilitate this, we use the Spring Framework [spr]. This framework allows to set up properties of Java objects and the dependencies between them via XML files, called contexts. We obtain such a context from the above described configuration file with the help of XSL transformations. Fig. 5.6 illustrates this.

5.3.3 Main system classes

Here we present the essential part of the class diagram of the system (Fig. 5.7), and outline the functionality of each class, along with the way they cooperate.

- `DataExchangeScenario` — a class representing the system configuration, either as individual properties, or via XML fragments, fitted for further XSL transformations.
- `Schema` — a relational schema representation.
- `TupleGeneratingDependency`, `EqualityGeneratingDependency` — the subclasses of the `Dependency` class, representing individual constraints. Provide convenience methods for analyzing the dependency structure: variables shared by the premise and the conclusion, compatible fields, etc.
- `SchemaInstance` — represents the schema instance. Most of the functionality is implemented in the subclasses: `SourceSchemaInstance` and `ParentSchemaInstance`.
- `SourceSchemaInstance` — determines the relevant objects of the source database for the data exchange session. Used by the `SourceToTargetLink` object.
- `TargetSchemaInstance` is responsible for construction and manipulating the target schema instance. Implements the chase.
- `SourceToTargetLink` — ensures that the source data is accessible from the target one. As the most straightforward scenario, a bulk copy of the source tables into the target database is performed. If a target database supports linking external data sources, the copying can be avoided.
- `TargetOptimizer` — computes the core of a canonical universal solution (computed and encapsulated by the `TargetSchemaInstance` object).
- `DbTool` is responsible for the platform-specific database operations (creating/dropping objects, translating data types, handling auto-incremental values etc.) Implements a `SqlStatementProvider` interface, constructing SQL statements from the XML scenario fragments with the help of XSLT.
- `Driver` — parses command line and runs the data exchange process.

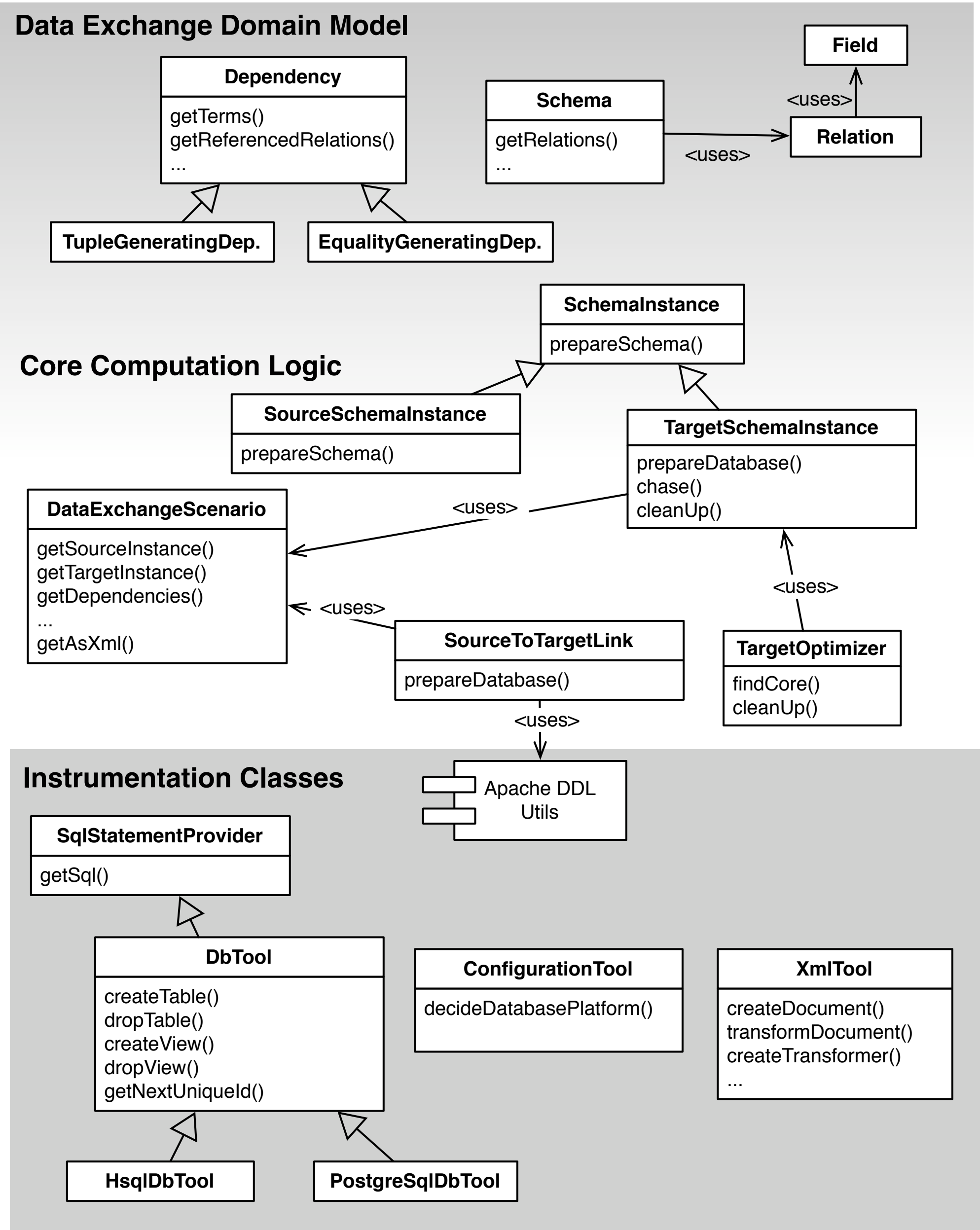


Figure 5.7: Main classes of the system.

Chapter 6

System tests and evaluation

We have implemented the prototype of a data exchange engine with core computation support. This chapter presents the test results of our system, as well as their evaluation. Testing the algorithm on different data sets revealed predictable limitations of the current implementation: as labeled nulls are modeled by views with large amount of joins (see Section 5.2.1), queries against them are much slower than against usual tables.

A rough estimation for data exchange scenarios that can be tackled by the current implementation, is 10 tables, with no more than 100 rows per input table *and* total of 1000 of labeled nulls generated by the chase. Though very far from the requirements of practical applications, these performance bounds allow to run diverse data exchange scenarios and to Analise the performance trends varying dependencies and the input. We believe that the current implementation as it is (i.e. using a database back-end and simulating labeled nulls with views), can still be improved to handle times as much data as it does now. However, a more detailed bottleneck analysis and most probably further algorithmic improvements are needed to approach any practically relevant database sizes. For a prototype, our main goal was to develop a system which is easy to run and to configure and which provides for easy inspection of internal data transfers for the debugging purposes.

The test configuration for this chapter:

- Hardware: MacBook Core 2 Duo 2GHz and 2Gb RAM.
- Operating System: Mac OS X 1.4 Tiger.
- Target database: PostgreSQL 8.1.
- Source database: Hypersonic SQL.

We start with a test set with tiny relations of arity less or equal than 3, and including TGDs that introduce redundant tuples to be removed at the core computation phase. In the following test sets, we create redundancy in target relations by adding *incomplete duplicates* to the source data, i.e., the variants of ground source tuples with some constants replaced by nulls (see the previous chapter for the discussion). The second test set concentrates on normalization and denormalization, and the third test set first splits a table into smaller slices and than restructures them.

For this chapter, we adopt the following naming convention: if a term starts with a capital letter, it denotes a variable (sometimes we also use subscripts), otherwise, the term is a constant. For example, in the dependency $S(A, C_1) \rightarrow \exists C_m\, P(A, b, C_1, C_m)$ A, C_1 and C_m are variables, and b is a constant.

6.1 Redundant tuples

The first test set was built to be as simple as possible. It deals with five small relations: F/1, S/1, R/3, P/2 and Q/2. The fields of each relation are named alphabetically (e.g. S/1 has the only field "a", Q/2 has fields "a" and "b" etc.). F/1 belongs to the source schema, and the rest are target relations. In all tests data transfer proceeds as shown by

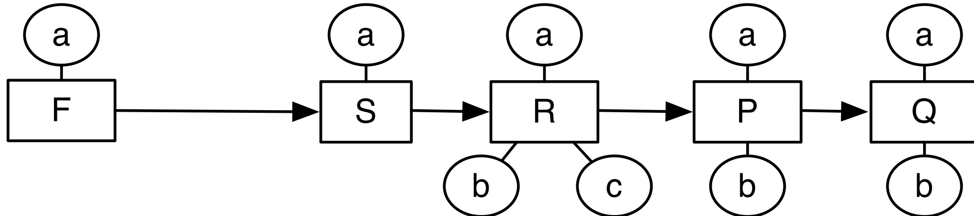


Figure 6.1: Schema of the first test set.

arrows. The source-to-target constraint is always the same: $F(X) \to S(X)$.

6.1.1 3-level target TGDs

The set Σ_t of target constraints in this case is the following (we do not use the relation $Q/2$ in this test set; also note that according to our naming convention, w denotes a constant):

1. $S(A) \to \exists X, Y, Z \ R(w, A, X) \wedge R(X, Y, Z)$
2. $R(A, B, C) \to \exists X, Y \ P(C, X) \wedge P(Y, X)$

Here the second rule generates redundant tuples in the relation P/2. In principle, it is possible to analyze the dependencies before chasing them in order to prune such "meaningless" tuples. A possible solution could be to compute a *core of a query* in the conclusion of a TGD, and use it for chase instead of the original dependency conclusion. It is easy to show, that tuples generated by those atoms of a TGD, which are not part of the core of its conclusion, can never be present in the core of the canonical solution.

However, we deliberately use suboptimal dependencies for our tests, just to provide enough work for the core computation algorithm, and, consequently, more material for the performance analysis. The possible optimization would be elimination the second atom $P(Y, X)$ from the conclusion of the last TGD, since there is a homomorphism from $P(Y, X)$ onto $P(C, X)$; variables carried from the premise act as constants in computing the core of such a conclusion query.

The results of the first test are presented in table 6.1. The obscure abbreviations in the header of this and the following similar tables need to be explained:

#SRC	total number of tuples in source relations
#VAR	number of variables (labeled nulls) generated by a chase
#OCCUR	number of variable occurences
#TGT	total number of tuples generated by a chase in target relations
#COREVAR	number of variables (labeled nulls) in a core
#1^{st}IT	number of variables, mapped out after the first iteration of the algorithm
CHASE	chase running time time in seconds
CORE	core computation time time in seconds

Table 6.1: Run-time of the 3-level TGD test case

#Src	#Var	#Occur	#Tgt	#CoreVar	$\#1^{st}$It	Chase, s.	Core, s.
10	70	120	70	50	20	2	10
30	210	360	210	150	60	4	17
50	350	600	350	250	100	7	31
70	490	840	490	350	140	5	52
80	560	960	560	400	160	6	67
100	700	1200	700	500	200	8	104

The graphical representation of the performance results are presented below, on Fig. 6.3. As follows from the results summary in Table 6.1, the core was found already after the first successful iteration of the algorithm (the sum of the columns #CoreVar and $\#1^{st}$It equals the total number of variables). This was possible due to the usage of the mapping choosing heuristics: *reuse images* and *prefer constants* (see Section 5.2.10). Let us illustrate by example how one of these heuristics works.

Let the source instance contain the only fact $F(1)$. Then STD generates the fact $S(1)$ in the target instance, and the target chase then creates the R- and P-facts, as shown on Fig. 6.2. The canonical universal solution consists of the following facts:

$$T = \{S(1),\ R(w,1,X_1),\ R(X_1,Y_1,Z_1),\ P(X_1,X_2),\ P(Y_2,X_2),\ P(Z_1,X_3),\ P(Y_3,X_3)\}$$

Both R-facts must necessarily be in the core, as well as the two P-facts sharing a variable with them, but the algorithm has to exclude the facts $P(Y_2,X_2)$ and $P(Y_3,X_3)$. Fig. 6.2 shows the iteration of a FindCore^E, which tries to build a proper endomorphism h, such that $h(Y_2) = h(X_1) = X_1$. The "reuse images" heuristic helps to get rid of the variable Y_3 already at the lifting step (step 7 of FindCore^E, procedure Extend).

6.1.2 4-level target TGDs

In this test we add another target relation $Q/2$, and the set Σ_t of target constraints grows appropriately:

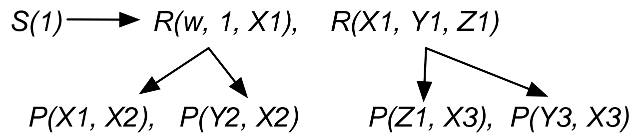
1. $S(A) \to \exists X,Y\ R(A,w,X) \wedge R(A,Y,X)$
2. $R(A,w,C) \to \exists X,Y\ P(C,X) \wedge P(Y,X)$
3. $P(A,B) \to \exists X,Y\ Q(A,X) \wedge Q(Y,X)$

The number of variables in this example doesn't change significantly, as now the second rule creates two times less tuples as in the previous test. This allows to evaluate the increase of processing time caused solely by the introduction of a new rule. We could also add a TGD that doesn't introduce any variables, i.e. a *full* TGD, but such full TGDs are ignored by the homomorphism extension routine, and thus do not contribute to the core computation time.

Again in this example, we were able to find the core already with the first run of the algorithm.

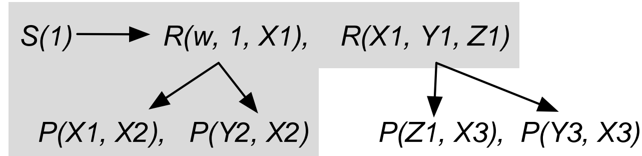
The test demonstrates a clear increase of the core computation time. This is partially due to a slightly greater number of labeled nulls and target tuples generated by the chase, but there is also another reason: the greater depth of a variable, the higher cost of trying to unify it with another term (step 3 of the FindCore^E algorithm in Chapter 4).

Canonical solution T:



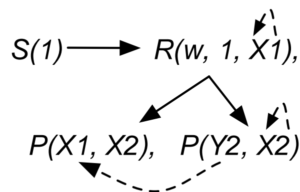
Trying to map Y2 → X1

1: $T_{Y2,X1}$



2: Find *h*: $T_{Y2,X1}$ → T

R(X1, Y1, Z1)

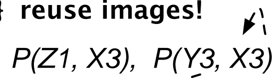


3: Extend *h* to *h'*: T → T, i.e, find images for X3 and Y3

Possible extensions:

— {X3 → X3 , Y3 → Y3}

— {X3 → X3 , Y3 → Z1} **reuse images!**



Z1 was already used

The core of T:

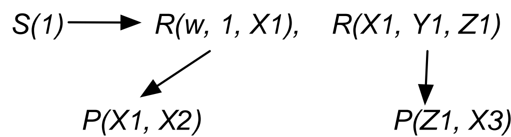


Figure 6.2: "Reuse images" heuristic prefers Z_1 over Y_3 on step 3

Table 6.2: Run-time of the 4-level TGD test case

#Src	#Var	#Occur	#Tgt	#CoreVar	$\#1^{st}$It	Chase, s.	Core, s.
10	80	150	90	30	50	3	13
30	240	400	270	90	150	2	25
50	400	750	450	150	250	7	44
70	560	1050	630	210	350	4	75
80	640	1200	720	240	400	5	95
100	800	1500	900	300	500	16	140

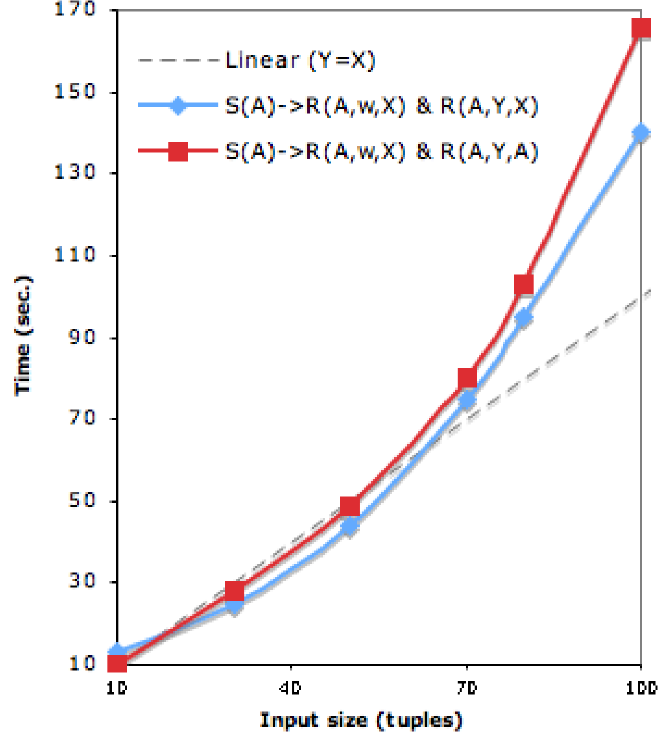


Figure 6.3: Core computation time as a function of input size, 4-level TGDs test set

However, currently the algorithm picks the variables for elimination at random. The bigger depth of a chosen variable, the more tuples are needed to check existence of an endomorphism that maps it out, and the larger joins are needed to perform such a check.

This can he illustrated by a slight modification of the first TGD:

$$S(A) \rightarrow \exists X, Y \ R(A, w, X) \wedge R(A, Y, X)$$

is changed to

$$S(A) \rightarrow \exists X, Y \ R(A, w, X) \wedge R(A, Y, A)$$

(note the change in the last atom). Now the tuples of $R/3$, produced by the last atom of the dependency 2 can no longer be mapped on those generated by the atom $R(A, w, X)$. The algorithm can only discover this by a systematic search, which results in loss of speed (see Fig. 6.3).

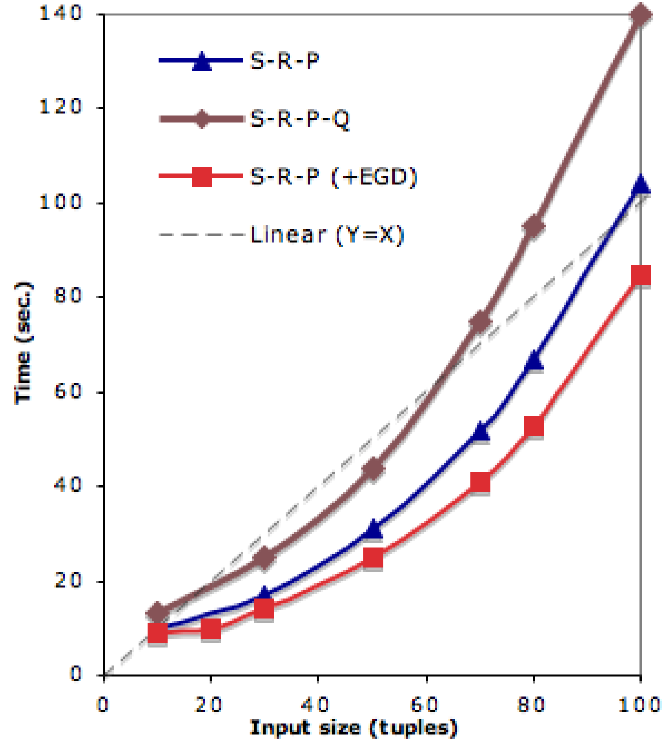


Figure 6.4: Core computation time vs. size of the input

6.1.3 3-level target TGDs and an EGD

In this test case, the set Σ_t of target constraints includes an EGD, turning tuples generated by the first TGD to be equal.

1. $S(A) \rightarrow \exists X, Y \;\; R(A, w, X) \wedge R(A, Y, X)$
2. $R(A, B, C) \rightarrow \exists X, Y \;\; P(B, X) \wedge P(Y, X)$
3. $R(A, B_1, C_1) \wedge R(A, B_2, C_2) \rightarrow B_1 = B_2 \wedge C_1 = C_2$

The EGD reduces the number of labeled nulls, and tuples in the target instance, leaving less work for core computation. So we see a better performance in comparison with the first example with the similar target TGDs (see Table 6.1 and Table 6.3). Fig. 6.4 summarizes the results of this test set.

Table 6.3: Run-time of the 3-level TGD test case

#Source	#Var	#Occur	#Target	Chase, sec.	Core, sec.
10	50	70	60	3	9
20	100	140	120	4	10
30	150	210	180	5	14
50	250	350	300	8	25
70	350	490	420	10	41
80	400	560	480	11	53
100	500	700	600	11	85

6.2 Normalization and denormalization

This test set works with two database schemata: a denormalized and a normalized one. A denormalized schema shown on Fig. 6.2 consists of a single table *ArticleHeap*, enumerating articles, authors, and publication details.

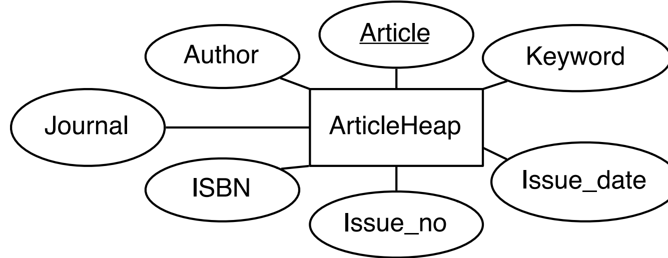


Figure 6.5: Denormalized schema: article heap

A normalized schema includes 7 relations, as shown on Fig. 6.2. Each table has an an *Id* field used for establishing references. There is an option of the system which allows to instantiate those *Id* fields with unique values after computing the core, thus "materializing" the references. This allowed us to run the test in the reverse direction, performing denormalization of data. We do not include charts for denormalization, since it is less relevant with respect to core computation: minimization of a single table is almost trivial compared to the normalization case. Though such an option is useful to to verify the correctness of transformations.

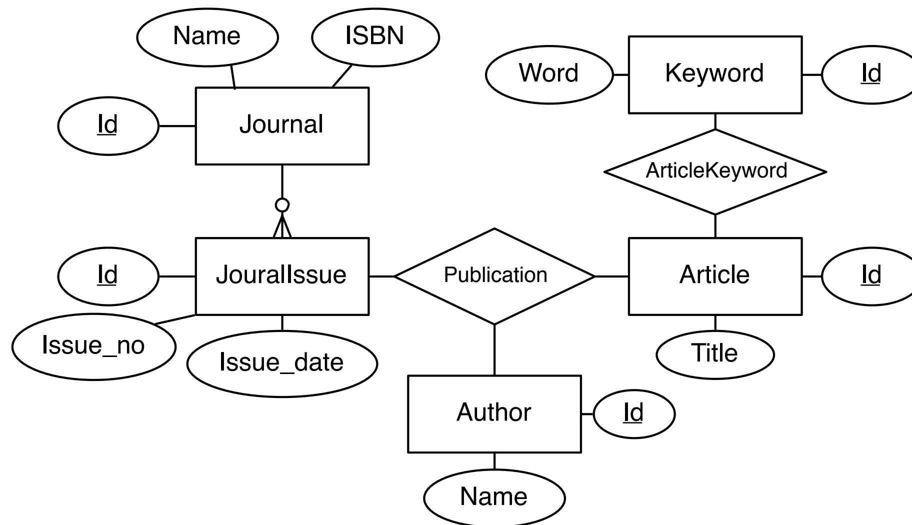


Figure 6.6: Normalized article database

The following two lengthy source-to-target dependencies are used for normalization:

1. $ArticleHeap(Title, Auth, JN, ISBN, IssNo, IssDate, KW) \rightarrow$
   $\exists ArtId,\ \exists AuthId,\ \exists JnId,\ \exists IssId,\ \exists PubId$
   $(Article(ArtId, Title) \wedge Author(AuthId, Name) \wedge$
   $Journal(JnId, JN, ISBN) \wedge JournalIssue(IssId, JnId, IssDate) \wedge$
   $Publication(PubId, ArtId, AuthId, IssId))$

2. $ArticleHeap(Title, Auth, JN, ISBN, IssNo, IssDate, KW) \rightarrow$
$\exists ArtId, \exists KWId\ (Article(ArtId, Title) \wedge Keyword(KWId, KW) \wedge$
$ArticleKeyword(ArtId, KWId)\ )$

Since all key values are generated by the source-to-target chase, every new key is a unique labeled null. For example, two authors with the same name will have different key values, shared with a *Publication* fact (the *AuthorId* field), which in its turn references facts of *JournalIssue* and *Article* etc. Thus, unless two tuples in the source relation were equal, no mapping exist between any of the correspondent tuples in the target relations. To enforce equal identifiers for equal target tuples we need EGDs:

1. $Author(Id_1, Name) \wedge Author(Id_2, Name) \rightarrow Id_1 = Id_2$
2. $Keyword(Id_1, Word) \wedge Keyword(Id_2, Word) \rightarrow Id_1 = Id_2$
3. $Article(Id_1, Title) \wedge Article(Id_2, Title) \rightarrow Id_1 = Id_2$
4. $Journal(Id_1, Name, ISBN) \wedge Journal(Id_2, Name, ISBN) \rightarrow Id_1 = Id_2$
5. $JournalIssue(Id_1, JournalId, Date, No) \wedge$
   $JournalIssue(Id_2, JournalId, Date, No) \rightarrow Id_1 = Id_2$
6. $Publication(Id_1, ArticleId, AuthorId, IssueId) \wedge$
   $Publication(Id_2, ArticleId, AuthorId, IssueId) \rightarrow Id_1 = Id_2$

There are no target inclusion dependencies asserting the correctness of foreign keys. But since we know that target relations are filled exclusively by the two source-to-target dependencies, which set the references right, we omitted them.

After introduction of target EDGs, however, the chase alone now produces a minimal solution, i.e., the core. Nevertheless, we consider this example relevant from the practical point of view. Generation of id values and foreign keys is a quite probable application of labeled nulls. Normally, these key values do not occur in other, non-key fields. Since we use field partitioning (see Section 5.2.4) for endomorphism search, these "key" variables are separated from the rest of the domain, and even if other, "non-key" nulls are present in the target database, the core computation for them will not be hindered by these key variables. Imagine such a scenario, when keys are generated by the labeled nulls, but there are also other null values possible. How big is the impact of this id-generation strategy on the overall core computation time? This test allows to answer this question.

As we see on Fig. 6.7, the impact of labeled null-based id generation on core computation is of the same order of magnitude as the chase (in fact, core checking is faster). Considering the usual discrepancy between the chase time and the core computation time (see Tables 6.1, 6.2 and 6.4), this is rather a mild effect.

6.3 Introducing a new relation with nulls

The last test set combines normalization and schema restructuring. As in the previous example, the source schema consists only of a single relation symbol, *ProjHeap* (Fig. 6.8). The target schema has two layers: first operates only with input data, the second introduces the *Dept* (i.e. department) relation between the employee and the projects he is responsible for. This is by no means a sensical transformation (for instance, a connection between cities and a projects is lost with such source-to-target dependencies), rather yet another test case for performance benchmarking.

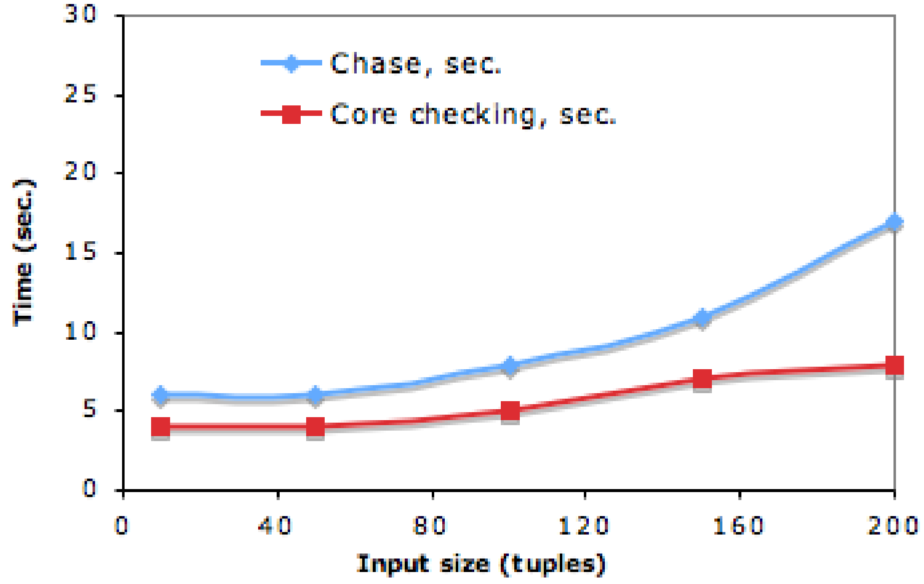


Figure 6.7: Times of core checking and chase on the article database

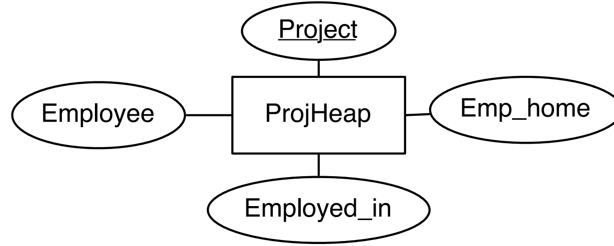


Figure 6.8: Project heap: the source schema

There is only one source-to-target dependency:

$$ProjHeap(Empl, LivesIn, WorksIn, Proj) \rightarrow \\ EmpCity(Empl, WorksIn) \wedge EmpProj(Empl, Proj) \wedge Home(Empl, LivesIn)$$

The set Σ_t consists of six dependencies:

1. $EmpCity(Empl, City) \rightarrow Home(Empl, City)$
2. $EmpCity(Empl, City) \rightarrow \exists Dept\ (EmpDept(Empl, Dept) \wedge DeptCity(Dept, City))$
3. $Home(Empl, City) \rightarrow \exists Dept\ (EmpDept(Empl, Dept) \wedge DeptCity(Dept, City))$
4. $EmpProj(Empl, Proj) \rightarrow \exists Dept\ (EmpDept(Empl, Dept) \wedge ProjDept(Proj, Dept))$
5. $EmpProj(Empl, Proj) \wedge EmpCity(Empl, City) \rightarrow$ $\exists Dept\ (EmpDept(Empl, Dept) \wedge ProjDept(Proj, Dept) \wedge DeptCity(Dept, City))$
6. $EmpDept(Empl, Dept) \rightarrow \exists City\ DeptCity(Dept, City)$

Here the TGD 5 generates a full set of target tuples (knowing the source schema, one might suggest that the rule 5 alone is enough). Without this rule, all tuples generated by TGDs 2 and 4 would persist in the core, as Fig. 6.10 explains. It is a property of cores that can seem counter-intuitive at the first sight. For example, two tuples

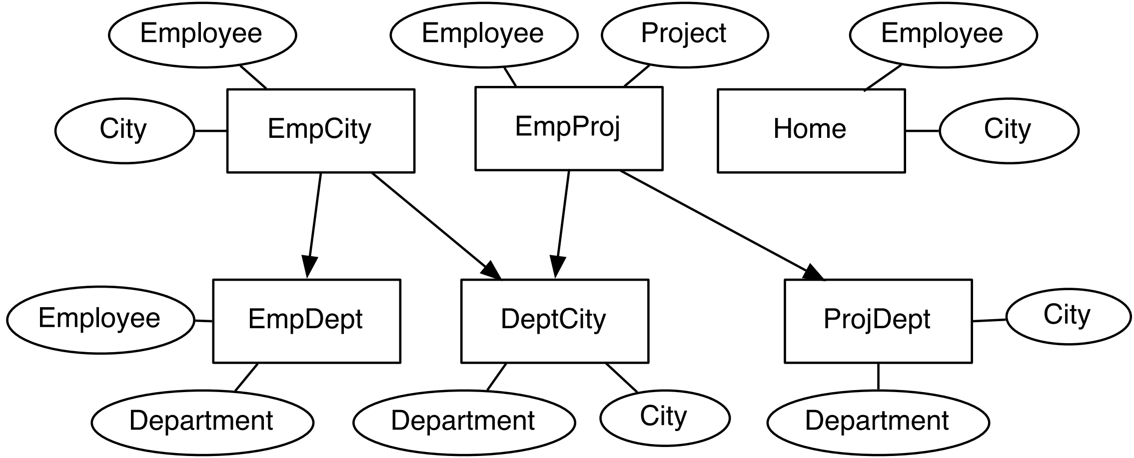


Figure 6.9: Target project database schema

$EmpDept('Mary', D_1)$ and $EmpDept('Mary', D_2)$ cannot be mapped one onto another, since D_1 is shared with a *DeptCity* fact but not with *ProjDept* one, and D_2 is shared with a fact in the *ProjDept* relation. But the two *EmpDept* tuples were induced by the same source relation.

To get rid of D_2 and D_1, we need a set of three facts: *DeptCity*, *EmpDept* and *ProjDept* sharing the same "department" value.

There is certain duality between the roles of EGDs and of the core computation with regard to instance minimization. Both reduce the number of variables, and EGDs can also eventually make two tuples identical. In our example, should we have an EGD

$$EmpDept(Name, Dep_1) \wedge EmpDept(Name, Dep_2) \rightarrow Dep_1 = Dep_2,$$

then D_1 would be immediately identified with D_2, which is something one can intuitively expect, looking on the source data. This is possible, because the employee name is a natural key of the *EmpDept* relation: we assume, that one employee can not work in two different departments at once. However, if this assumption is false, then EGD cannot be applied, while it is still possible to compact the instance with the help of core computation.

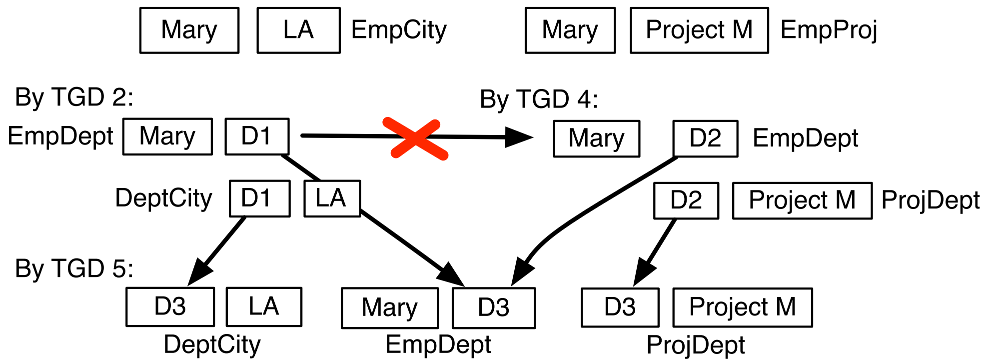


Figure 6.10: Possible homomorphisms between variables of EmpDept relation

At this time, the experiment consisted of two parts. In the first, no input tuple contained null values. This allowed us to test the removal of tuples introduced by all target TGDs other than TGD 5. The results of this experiment are summarized in Table 6.4.

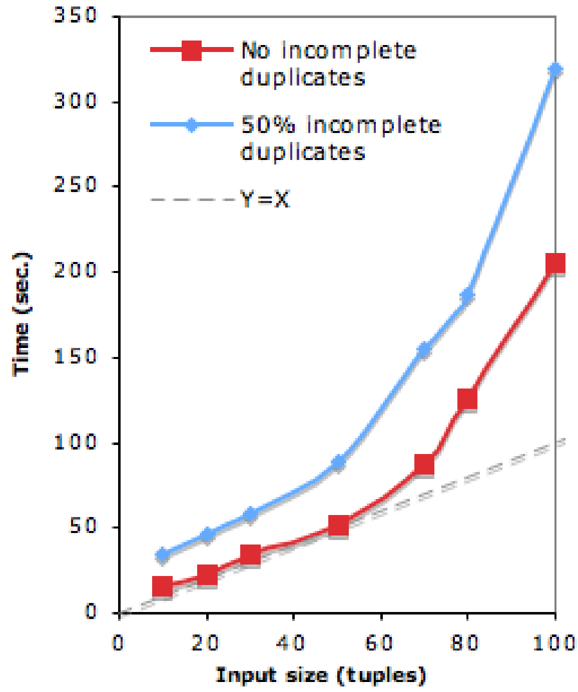


Figure 6.11: Core computation time for the Projects testcase

For the second part, half of the input tuples were incomplete duplicates of some other tuples in the source relation, thus reducing the core size and ensuring more work for the algorithm. For instance, we had

ProjHeap('Mary', 'New York', 'New Jersey', 'Tax consulting')

ProjHeap('Mary',NULL,'New Jersey',NULL)

the second two rows being a duplicate of the first one with some constant values replaced by nulls.

As expected, processing time was higher in the second case (see Fig. 6.11 for comparison). Adding a single duplicate per ground fact, led to some 30% of performance loss.

Table 6.4: Core computation over the Projects database - no incomplete input tuples

#Source	#Var	#Occur	#Target	Chase, sec.	Core, sec.
10	22	51	61	3	16
20	40	90	110	3	22
30	90	210	250	4	35
50	162	396	456	5	52
70	248	594	694	6	87
80	420	1050	1190	15	126
100	460	1140	1300	15	205

6.4 Summary and evaluation of results

In this chapter, we have described a few first tests of a prototype system, supporting core computation for data exchange. Though its current performance is quite far from the real-world requirements, the prototype serves its aim, being a indispensable experimentation tool for a new algorithm. We believe that this was the first step towards creation a practically useful core application. After first series of experiments with the prototype, we can conclude the following:

- Chosen XML-based configuration approach facilitates the rapid development of different data exchange scenarios. Moreover, the usage of a familiar DBMS-based back-end provides for easy monitoring of data transfers, which is important for validation and debugging the data exchange applications.
- The prototype currently tackles scenarios with roughly 10 tables and 100 rows per table.
- From our experiments it became clear, that for many scenarios it is impossible to obtain an intuitively expected solution without target EGDs. Therefore, our algorithmic improvement, concerning direct application of EGDs, is of practical importance.
- We have shown a feasibility of a possible application of labeled nulls, namely, generation of row id values and foreign keys. Due to field partitioning, it does not hinder the core computation performance.
- For some scenarios, already the first approximation found by $\textsc{FindCore}^E$ can be the core itself, or be very close to it. This became possible due to quite simple heuristics used for homomorphism construction.

Based on this results, we outline the future work directions in the next chapter.

Chapter 7

Conclusions

In this thesis we have revisited the core computation, which is an essential problem in the area of data exchange. On the one hand, we have presented an enhanced version of the FindCore algorithm from [GN06], which avoids the simulation of EGDs by TGDs. On the other hand, we have actually implemented this algorithm and drawn first conclusions from the experiments carried out with this implementation. In particular, we have thus identified possible directions for further improvements.

7.1 Future work

As a first task for the future development of the system, we consider a more detailed bottleneck analysis of our current implementation. Our ultimate goal is to build a system making use of cores for industrial-size databases.

Even being polynomial, the core computation procedure seems to be highly computationally expensive, and hence hardly applicable to the large real-world databases. However, it could be useful to develop an algorithm for effective computation of core approximations. In our experiments, already the first successful iteration of the algorithm often produced a very good result.

There exist practical cases, when any reasonable minimization of the target instance is valuable, not necessarily the ultimate one (e.g., to save the disk space used by the target instance, and to improve its query performance). Therefore, we consider the core approximation approach among the most important directions for future development.

Summarizing, the following ideas seem worth investigating to us:

- Dependency preprocessing
  - Compute the cores of the queries in the conclusions of TGDs.
  - Consider using the hypertree decomposition techniques for optimizing dependencies.
  - Source dependencies, which are often available as database metadata, can be used to automatically derive target constraints (especially, functional dependencies, expressed by EGDs).
- The variables with smaller depth must be tried first by the algorithm. In conjunction with heuristics for construction of homomorphisms, this approach proved effective for fast elimination of redundant variables, and consequently, for faster core identification.

- It would be interesting to find a (perhaps, heuristic-based) technique for estimating redundancy in the database. So far, the core computation is a "blind" process, having no cue how far from minimal the current instance is, and which part of an instance contains the most of redundant information. This led to suboptimal performance in the majority of experiments.

Bibliography

[BV84] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC'77*, pages 77–90. ACM Press, 1977.

[Fag82] Ronald Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, 1982.

[FKMP03] Ronald Fagin, Phokion G. Kolaitis, Rene J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *Proc. ICDT'03*, pages 207–224. Springer, 2003.

[FKP03] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: Getting to the core. In *ACM PODS 2003, San Diego, CA*, 2003.

[FS89] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354, New York, NY, USA, 1989. ACM Press.

[GF64] Benrard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[GI91] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

[GN06] Georg Gottlob and Alan Nash. Data exchange: computing cores in polynomial time. In *Proc. PODS'06*, pages 40–49. ACM Press, 2006.

[Got05] Georg Gottlob. Computing cores for data exchange: new algorithms and practical solutions. In *Proc. PODS'05*, pages 148–159. ACM Press, 2005.

[HHH+05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *Proc. SIGMOD'05*, pages 805–810. ACM, 2005.

[HN92] Pavol Hell and Jaroslav Nešetřil. The core of a graph. *Discrete Mathematics*, 109(1-3):117–126, 1992.

[HRO06] Alon Y. Halevy, Anand Rajaraman, and Joann J. Ordille. Data integration: The teenage years. In *Proc. VLDB'06*, pages 9–16. ACM, 2006.

[Kol05] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *Proc. PODS'05*, pages 61–75. ACM, 2005.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246. ACM, 2002.

[Lib06] Leonid Libkin. Data exchange and incomplete information. In *Proc. PODS'06*, pages 60–69. ACM Press, 2006.

[RL90] Ronald L. Rivest and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[spr] Spring framework (www.springframework.org).