Roma Tre University
Ph.D. in Computer Science and Automation

Advanced Techniques for Mapping and Cleaning

Donatello Santoro

Advanced Techniques for Mapping and Cleaning

A thesis presented by
Donatello Santoro
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Automation

Roma Tre University
Department of Engineering

May 2014

COMMITTEE:
Prof. Giansalvatore Mecca

REVIEWERS:
Prof. Marcelo Arenas
Prof. Renée J. Miller

This thesis is dedicated to the memory of my father, Antonio.

Abstract

We address the challenging and open problem of bringing together two crucial activities in data integration and data quality, i.e., transforming data using schema mappings, and fixing conflicts and inconsistencies using data repairing. This problem is made complex by several factors. First, schema mappings and data repairing have traditionally been considered as separate activities, and research has progressed in a largely independent way in the two fields. Second, the elegant formalizations and the algorithms that have been proposed for both tasks have had mixed fortune in scaling to large databases. In the thesis, we introduce a very general notion of a mapping and cleaning scenario that incorporates a wide variety of features, like, for example, user interventions. We develop a new semantics for these scenarios that represents a conservative extension of previous semantics for schema mappings and data repairing. Based on the semantics, we introduce a chase-based algorithm to compute solutions. Appropriate care is devoted to developing a scalable implementation of the chase algorithm. To the best of our knowledge, this is the first general and scalable proposal in this direction.

Acknowledgments

Several times during the course of my doctoral studies I have thought about what to write on this page. These years have presented me with a great opportunity to grow both professionally, and personally.

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted for being with me throughout this experience.

Foremost, I would like to express my sincere gratitude to my advisor, Gianni Mecca, for being a brilliant mentor, egging me on with his continuous support, motivation, professionalism and profound knowledge. His guidance has molded my entire academic existence, and expedited the research documented in this dissertation. He has nurtured my growth as a student and a researcher.

I would like to thank my other colleagues at work, Floris Geerts and Paolo Papotti, for their helpful contributions and improvement-inducing suggestions. I learnt a lot from them.

I am also grateful to Prof. Elisa Bertino, who gave me the opportunity to visit the Cyber Center in the Purdue University.

Thanks to all my friends Antonio, Donato, the colleagues of the Laboratorio ICAR who created a great working atmosphere, and my roommate Ankush, who made my American stint memorable.

Last but not least, I thank with love my parents Lina and Antonio, who always believed in me and who gave me the opportunity to study and achieve my goals, teaching me the importance of higher education. Thanks to my siblings, Serena and Flavio, for their support, encouragement, and all the crazy times together which helped me get through the serious times.

Last, and certainly not the least, thanks to my soul mate Carmen, for her love, her incredible patience, and for her support during the good and bad moments of these years. I love you.

Preface

Given the increase in the number and variety of data sources, the development of sophisticated techniques to correlate and integrate data is a crucial requirement in modern data-driven applications. At the same time, it is also well known that data often contain inconsistencies, and that dirty data incurs economic loss and erroneous decisions.

For these reasons, data transformation and data cleaning are two very important research and application problems. Data transformation, or data exchange, relies on declarative *schema mappings* to translate and integrate data coming from one or more source schemas into a different target schema. Data cleaning, or data repairing, uses declarative *data-quality rules* in order to detect and remove errors and inconsistencies from the data.

It is widely recognized that whenever mappings among different sources are in place, there is a strong need to clean and repair data. Despite this need, database research has so far investigated schema mappings and data repairing essentially in isolation.

In this thesis we present the Llunatic mapping and cleaning system, the first comprehensive proposal to handle schema mappings and data repairing in a uniform way. Llunatic is based on the intuition that transforming and cleaning data can be seen as different facets of the same problem, unified by their declarative nature. This declarative approach that allowed us to incorporate unique features into the system and apply it to wide variety of application scenarios.

We show that our proposal generalizes many previous approaches. To the best of our knowledge, this is the first proposal that achieves the level of generality needed to handle three different kinds of problems: traditional mapping problems, traditional data repairing problems, and the new and more articulated category of data translation problems with conflict resolution. We believe that these contributions make a significant advancement with respect

to the state-of-the-art, and may bring new maturity to both schema mappings and data repairing.

This thesis generalizes and extends results that were previously published in conference papers [GMPS13, GMPS14]. We make several advancements with respect to the conference versions:

(*a*) in Section 7 we present a comprehensive treatment of the semantics that extends and generalizes those that were presented earlier in a more synthetic form due to space limitations; given the richness and complexity of the framework, we felt that this was a needed extension;

(*b*) we provide a detailed development of two variants of the chase algorithm (Sections 8 and 9), which stands at the foundations of our mapping and repairing algorithms;

(*c*) we develop full proofs of all theorems (in the Appendix);

(*d*) we develop an in-depth comparison of our semantics to previous semantics (Sections 7.6 and 10), both for data exchange and for data repairing, in order to substantiate our statement that this work extends and unifies many of the previous approaches.

(*e*) we develop these algorithms in a working prototype with a graphical user interface to allow user interaction. We present the system and experimental results in Section 12. We plan to release the prototype under an open-source license on one of the major open-source repositories.

Contents

List of Tables

List of Figures

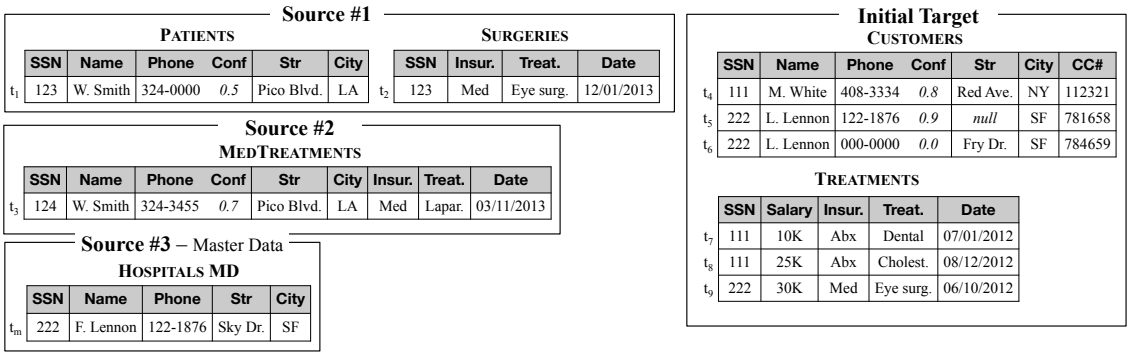
Chapter 1

Introduction

This thesis discusses two important problems in database research, namely to integrate and transform data coming from different repositories using *schema mappings*, and to study the quality of the resulting database using *declarative constraints*. Schema mappings are executable transformations that specify how an instance of a source repository should be translated into an instance of a target repository. A rich body of research has investigated mappings, both with the goal of developing practical algorithms [PVM+02], and nice and elegant theoretical foundations [FKMP05]. However, it is also well known that data often contain inconsistencies, and that dirty data incurs economic loss and erroneous decisions [FG12]. The *data-cleaning* (or *data-repairing*) process consists in removing inconsistencies with respect to some set of constraints over the target database.

We may say that both schema-mappings and data repairing are long-standing research issues in the database community. However, so far they have been essentially studied in isolation. On the contrary, we notice that whenever several possibly dirty databases are put together by schema mappings, there is a very high probability that inconsistencies arise due to conflicts and errors in the source data, and therefore there is even more need for cleaning. In fact, bringing together schema mappings and data repairing is considered an open problem [BFM07]. Solving this problem is far for trivial. To illustrate why, we introduce next our motivating example.

Example 1: [Motivating Example] Consider the data scenario shown in Figure 1.1. Here we have several different hospital-related data sources that must be correlated to one another. The first repository has information about Patients



	SSN	Name	Phone	Conf	Str	City
t_1	123	W. Smith	324-0000	*0.5*	Pico Blvd.	LA

	SSN	Insur.	Treat.	Date
t_2	123	Med	Eye surg.	12/01/2013

	SSN	Name	Phone	Conf	Str	City	Insur.	Treat.	Date
t_3	124	W. Smith	324-3455	*0.7*	Pico Blvd.	LA	Med	Lapar.	03/11/2013

	SSN	Name	Phone	Str	City
t_m	222	F. Lennon	122-1876	Sky Dr.	SF

	SSN	Name	Phone	Conf	Str	City	CC#
t_4	111	M. White	408-3334	*0.8*	Red Ave.	NY	112321
t_5	222	L. Lennon	122-1876	*0.9*	*null*	SF	781658
t_6	222	L. Lennon	000-0000	*0.0*	Fry Dr.	SF	784659

	SSN	Salary	Insur.	Treat.	Date
t_7	111	10K	Abx	Dental	07/01/2012
t_8	111	25K	Abx	Cholest.	08/12/2012
t_9	222	30K	Med	Eye surg.	06/10/2012

Figure 1.1: A Hospital Mapping and Cleaning Scenario.

and Surgeries. The second one about MedTreatments.

Our goal is to move data from the source database into a possibly non-empty target database. The target database organizes data in terms of Customers with their addresses and credit-card numbers, and medical Treatments paid by insurance plans. Notice that the source databases may contain inconsistencies, and possibly come with an associated *confidence* for attributes. The confidence is not mandatory in our approach, i.e., all source attributes may be considered as equally confident, but when present it may help to solve some of the conflicts in the target. In our example, we assume that a confidence of 0.5 has been estimated for the Phone attribute of the first data source, and 0.7 for the second. Additional confidence attributes may also be present in the target database. We also report confidences for the Phone attribute in the target.

A data architect facing this scenario must deal with two different tasks. On the one side, s/he has to develop the mappings to exchange data from the source databases to the target. On the other side, s/he has to devise appropriate techniques to repair inconsistencies that may arise during the process.

Using Mapping to Exchange Data

Let us first discuss task 1, i.e., data exchange via mappings. The desired transformation can be expressed as a set of *tuple generating dependencies (tgds)* [FKMP05].

(1) *S-t Tgds* : We have two s-t tgds, as follows (as usual, universal quantifiers in front of the tgds are omitted):

m_1. Pat(ssn, name, phn, conf, str, city), Surg(ssn, ins, treat, date)
$\rightarrow \exists Y_1, Y_2$: Cust(ssn, name, phn, conf, str, city, Y_1), Treat(ssn, Y_2, ins, treat, date)
m_2. MedTreat(ssn, name, phn, conf, str, city, ins, treat, date)
$\rightarrow \exists Y_3, Y_4$: Cust(ssn, name, phn, conf, str, city, Y_3), Treat(ssn, Y_4, ins, treat, date)

Each tgd states a constraint over the target database. For example, tgd m_2 says that for each tuple in the MedTreatments source table, there must be corresponding tuples in the Customers and Treatments target tables; Y_i are existential variables representing values that are not present in the source database but must be present in the target. The appropriate confidence value is copied into the conf attribute of the target tuples.

Target Constraints

Notice that besides deciding how to populate the target in order to satisfy the s-t tgds above, we must also deal with the problem of generating target instances that comply with target constraints. Traditionally [AHV95], database architects have specified constraints of two forms: inclusion constraints and functional dependencies. These are expressible in data exchange under the form of *target tgds* and target *equality generating dependencies (egds)*.

(2) *Target Tgds*: Target tgds are tgds in which only target symbols appear, and express inclusion constraints that are typically associated with foreign keys. In our example, we have that the SSN attribute in the Treatments table references the SSN of a customer in Customers. This is expressed using the following target tgd:

$$m_3.\ \text{Treat}(\text{ssn}, \text{sal}, \text{ins}, \text{treat}, \text{date}) \rightarrow \exists Y_5, Y_6, Y_7, Y_8, Y_9, Y_{10} : \text{Cust}(\text{ssn}, Y_5, Y_6, Y_7, Y_8, Y_9, Y_{10})$$

(3) *Functional Dependencies (FDs)*: The target database also comes with a number of FDs: $d_1 = (\text{SSN}, \text{Name} \rightarrow \text{Phone})$, $d_2 = (\text{SSN}, \text{Name} \rightarrow \text{CC\#})$ and $d_3 = (\text{Name}, \text{Str}, \text{City} \rightarrow \text{SSN})$ on table Customers. Here, d_1 requires that a customer's social-security number (SSN) and name uniquely determine his or her phone number (Phone). Similarly for d_2 and d_3. As we mentioned, functional dependencies can be expressed as egds as follows:

e_1. Cust(**ssn**, **n**, p, s, c, cc), Cust(**ssn**, **n**, p', s', c', cc') $\rightarrow$ p = p'
e_2. Cust(**ssn**, **n**, p, s, c, cc), Cust(**ssn**, **n**, p', s', c', cc') $\rightarrow$ cc = cc'
e_3. Cust(ssn, **n**, p, **s**, **c**, cc), Cust(ssn', **n**, p', **s**, **c**, cc') $\rightarrow$ ssn = ssn'

Since we don't assume that the target database is empty, in Figure 1.1 we have reported an instance of the target. There, the pair of tuples $\{t_5, t_6\}$ violates both d_1 and d_2; the database is thus dirty.

It is worth noting that, aside from inclusion constraints and functional dependences, the recent literature has shown that more advanced forms of constraints are often necessary in data cleaning applications [FG12]. To model these, new forms of data-quality constraints have been introduced. Here we mention *conditional functional dependencies* [FGJK08], *conditional inclusion dependencies* [FG12], and *editing rules* [FLM$^+$10], among others. We designed our example in such a way to incorporate a flavor of these as well. In particular, we assume two conditional functional dependencies (CFDs):

(4) *CFDs*: A CFD $d_4 = (\mathsf{Insur}[\text{Abx}] \rightarrow \mathsf{Treat}[\text{Dental}])$ on table Treatments, expressing that insurance company 'Abx' only offers dental treatments ('Dental'). Tuple t_8 violates d_4, adding more dirtiness to the target database.

(5) *Inter-table CFDs*: In addition, we also have an *inter-table* CFD d_5 between Treatments and Customers, stating that the insurance company 'Abx' only accepts customers who reside in San Francisco ('SF'). Tuple pairs $\{t_4, t_7\}$ and tuples $\{t_4, t_8\}$ violate this constraint.

Finally, as it is common in corporate information systems [Los09], an additional *master-data table* is available in the source database; this table contains highly-curated records whose values have high accuracy and are assumed to be clean. We also assume an additional constraint to clean target tuples based on values in the master-data table:

(6) *Editing Rules*: our master-data based editing rule, d_6, states that whenever a tuple t in Customers agrees on the SSN and Phone attributes with some master-data tuple t_m in the master-data table Hospitals, then the tuple t must take its Name, Str, City attribute values from t_m, i.e., $t[\mathsf{Name}, \mathsf{Str}, \mathsf{City}] = t_m[\mathsf{Name}, \mathsf{Str}, \mathsf{City}]$. Tuple t_5 does not adhere to this rule as it has a missing street value (NULL) instead of 'Sky Dr.' as provided by the master-data tuple t_m.

In summary, our example is such that:
(a) it requires to map different source databases into a given target database;
(b) it assumes that the target database may be non-empty, and that both the sources and the target instance may contain errors and inconsistencies;
(c) it comes with a variety of data-quality constraints over the target; these include the common inclusion and functional dependencies expressible by target tgds and target egds, but also richer constraints, like conditional dependencies and master-data-based editing rules.

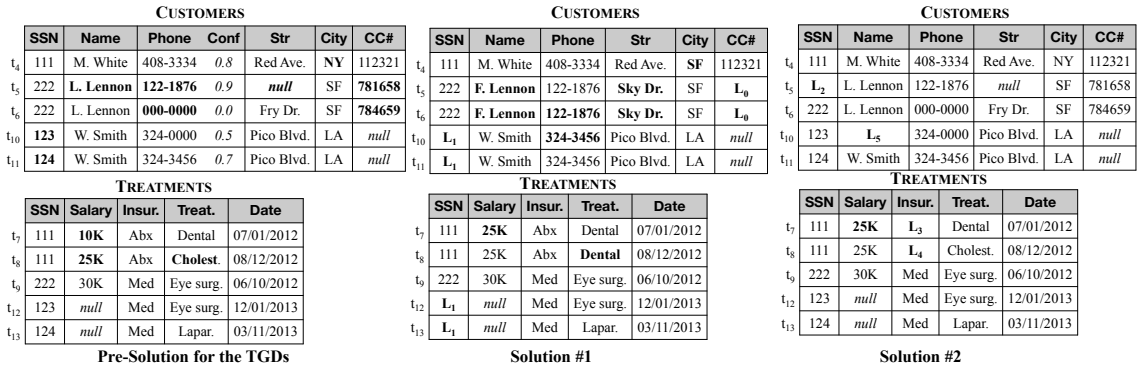
Given the source instances and the target, our goal is to generate an instance of the target database that preserves the mapping and that it is clean wrt target constraints. We start with a number of observations with respect to this problem.

First, we notice that *data exchange* [FKMP05] provides an elegant semantics for the execution of the source-to-target mappings (item (1) in our example). It also incorporates techniques to generate target instances that comply with target tgds and target egds (items (2) and (3)). Data exchange concentrates on *soft violations*. We call a *soft violation* for an egd any violation of the egd that can be removed by replacing one or more null values, either by a constant or by another null value. Consider for example the Customers table in the target database. Assume we have a constraint that states that ssn is a key for the table; we notice that tuples t_5, t_6 violate the constraint. The conflict between street names is a soft violation, since $t_5.\mathsf{Str} = \text{null}$, $t_6.\mathsf{Str} =$ 'Fry Dr.'. On the contrary, as we discussed above, we assume that the data may also contain *hard violations*, i.e., violations that may only be repaired by changing one constant into another constant. In our example this happens with credit card numbers, since $t_5.\mathsf{CC\#} = 781658$, $t_6.\mathsf{CC\#} = 771859$. In case of hard violations, a data exchange scenario has no solution.

In fact, solving hard violations is he primary goal of data repairing algorithms. Early works about database repairing [ABC99] modeled repairs for data quality constraints as sets of tuple-insertions and tuple-deletions. Since tuple-deletions may result in unnecessary loss of information, the recent literature has concentrated on cell changes [FG12], each cell being an attribute of a tuple. In this respect, the recent literature has provided us with a good arsenal of approaches and techniques to repair conflicts. In this thesis, we want to capitalize on this wealth of knowledge about the subject, and investigate the following foundational problem: *what should a database administrator do when facing a complex data-repairing problem that requires to bring together different data-quality constraints, as discussed above?* A second, striking observation about our example is that, despite many studies on the subject, there is currently no way to handle scenarios like the one in our example. The main problem is the lack of a uniform formalism to handle different data repairing constraints. In fact, although repairing strategies exist for each of the individual classes of constraints discussed at items (2), (3), (4), (5) and (6), there is currently no formal semantics for their combination.

The third, important observation, is that simply pipelining existing techniques does not work in general. Indeed, why can't we simply rely on a multi-step process in which: (*i*) we first use the standard semantics of data exchange

[FKMP05] in order to generate an initial instance of the target that is a *pre-solution* for the tgds at items (1), (2), as shown in Figure 1.2. Then, (*ii*) we use a combination of known data repairing algorithms for the constraints at (3), (4), (5), (6), like those in [BFFR05], [BFM07] or [BIG10], to repair the target?



CUSTOMERS

	SSN	Name	Phone	Conf	Str	City	CC#
t_4	111	M. White	408-3334	*0.8*	Red Ave.	**NY**	112321
t_5	222	**L. Lennon**	**122-1876**	*0.9*	***null***	SF	**781658**
t_6	222	L. Lennon	**000-0000**	*0.0*	Fry Dr.	SF	**784659**
t_{10}	**123**	W. Smith	324-0000	*0.5*	Pico Blvd.	LA	*null*
t_{11}	**124**	W. Smith	324-3456	*0.7*	Pico Blvd.	LA	*null*

TREATMENTS

	SSN	Salary	Insur.	Treat.	Date
t_7	111	**10K**	Abx	Dental	07/01/2012
t_8	111	**25K**	Abx	**Cholest**.	08/12/2012
t_9	222	30K	Med	Eye surg.	06/10/2012
t_{12}	123	*null*	Med	Eye surg.	12/01/2013
t_{13}	124	*null*	Med	Lapar.	03/11/2013

Pre-Solution for the TGDs

CUSTOMERS

	SSN	Name	Phone	Str	City	CC#
t_4	111	M. White	408-3334	Red Ave.	**SF**	112321
t_5	222	**F. Lennon**	122-1876	**Sky Dr.**	SF	$\mathbf{L_0}$
t_6	222	**F. Lennon**	**122-1876**	**Sky Dr.**	SF	$\mathbf{L_0}$
t_{10}	$\mathbf{L_1}$	W. Smith	**324-3456**	Pico Blvd.	LA	*null*
t_{11}	$\mathbf{L_1}$	W. Smith	324-3456	Pico Blvd.	LA	*null*

TREATMENTS

	SSN	Salary	Insur.	Treat.	Date
t_7	111	**25K**	Abx	Dental	07/01/2012
t_8	111	25K	Abx	**Dental**	08/12/2012
t_9	222	30K	Med	Eye surg.	06/10/2012
t_{12}	$\mathbf{L_1}$	*null*	Med	Eye surg.	12/01/2013
t_{13}	$\mathbf{L_1}$	*null*	Med	Lapar.	03/11/2013

Solution #1

CUSTOMERS

	SSN	Name	Phone	Str	City	CC#
t_4	111	M. White	408-3334	Red Ave.	NY	112321
t_5	$\mathbf{L_2}$	L. Lennon	122-1876	*null*	SF	781658
t_6	222	L. Lennon	000-0000	Fry Dr.	SF	784659
t_{10}	123	$\mathbf{L_5}$	324-0000	Pico Blvd.	LA	*null*
t_{11}	124	W. Smith	324-3456	Pico Blvd.	LA	*null*

TREATMENTS

	SSN	Salary	Insur.	Treat.	Date
t_7	111	**25K**	$\mathbf{L_3}$	Dental	07/01/2012
t_8	111	25K	$\mathbf{L_4}$	Cholest.	08/12/2012
t_9	222	30K	Med	Eye surg.	06/10/2012
t_{12}	123	*null*	Med	Eye surg.	12/01/2013
t_{13}	124	*null*	Med	Lapar.	03/11/2013

Solution #2

Figure 1.2: Pre-solution and solutions.

Unfortunately, this is not feasible. In fact, in the thesis we formally prove that dependencies, i.e., mappings and data quality constraints, interact in such a way that simply pipelining the two semantics often does not return solutions (details are in Section 7). To have a simple intuition of this, consider the pre-solution in Figure 1.2. It satisfies the tgds at items (1) and (2) in our example. However, it contains inconsistencies wrt the key constraints described at item (3) (highlighted in bold) due to conflicts in the initial instances. However, repairing the pre-solution may cause a violation of the tgds and hence the mappings need to be applied again. For example, in order to solve the conflict between tuples $\{t_{10}, t_{11}\}$, that violate d_3, one may want to equate $t_{10}[\mathsf{SSN}]$ and $t_{11}[\mathsf{SSN}]$, for instance changing $t_{11}[\mathsf{SSN}]$ to '123'. The resulting repaired instance will satisfy the constaint d_3, but it will violate the tgd m_3.

Notice also that data quality constraints interact with each other in non-trivial ways. To see this, consider dependencies d_1 and d_5. Suppose we use d_1 to repair tuples t_5, t_6 such that both have phone-number '122-1876'; then, since t_5 and t_6 agree with the master-data tuple t_m, we can use d_5 to fix names, streets and cities, to obtain: (222, F. Lennon, 122-1876, Sky Dr., SF, 781658), for t_5, and (222, F. Lennon, 122-1876, Sky Dr., SF, 784659), for t_6. However,

if, on the contrary, we apply d_5 first, only t_5 can be repaired as before; then, since t_5 and t_6 do not share the same name anymore, d_1 has no violations. We thus get a different result, of inferior quality.

Based on these observations, in this thesis we tackle the complex problem of defining a single, uniform framework for mapping and cleaning, and present LLUNATIC, the first comprehensive proposal to handle schema mappings and data repairing in a uniform way. LLUNATIC is based on the intuition that transforming and cleaning data can be seen as different facets of the same problem, unified by their declarative nature. It is the first system that supports three kinds of scenarios:

Type 1: schema-mapping scenarios in the spirit of [FKMP05], with one or more source databases, and a target database that is related to the sources by a set of schema mappings; integrity constraints can be imposed over the target under the form of inclusion and functional dependencies; given a set of source instances, the goal is to generate a valid instance of the target according to the mappings.

Type 2: data repairing scenarios in the spirit of [FG12], with one target database, possibly multiple *authoritative* source tables with highly curated data, used to model the so called *master-data* [Los09], and a set of data-quality rules over the target; these may include functional dependencies, conditional functional dependencies, editing rules, but also inclusion dependencies and conditional inclusion dependencies; here, we are given an instance of the target that is dirty with respect to the constraints, and want to generate a clean instance.

Type 3: mapping and cleaning scenarios, that combine and generalize the two above; here we have multiple sources that may include master-data, one target, a set of mappings that relate the target to the sources, and a rich set of data quality constraints over the target. Our reference example in Figure 1.1 falls in this category.

The latter kind of scenarios illustrate the main novelty of our approach: on the one side they model the problem of exchanging and repairing data as a single process, and on the other side they conservatively extend and unify type 1 and 2 scenarios, which are the focus of previous approaches.

We want to emphasize, however, that our approach brings some interesting extensions also to type 1 and type 2 scenarios. For example, type 1 scenarios generalize traditional data exchange scenarios since we allow for non empty target databases. Type 2 scenarios generalize what it is typically found in data repairing papers, since the vast majority of papers about data repairing have only considered single tables with functional dependencies and their variants,

and disregarded foreign keys (i.e., inclusion constraints), or the integration among different classes of constraints, like conditional constraints and editing rules.

Our ultimate goal is to devise a single algorithm to generate solutions for all kinds of scenarios listed above. In addition, we want to develop an implementation that is as scalable as possible. Computing solutions in a scalable way is a paramount problem, both in data exchange and data repairing. In fact, computing repairs requires to explore a space of solutions of exponential size wrt the size of the database.

Chapter 2

Contributions

In the thesis, we make several important and nontrivial contributions.

2.1 A Uniform Framework for Mapping and Cleaning

We develop a general framework for mapping and cleaning that can be used to generate solutions to complex data transformation scenarios, and to repair conflicts and inconsistencies among the sources with respect to a very wide class of target constraints.

The framework is a conservative extension of the well-known framework of data exchange, and incorporates most of the features considered in existing algorithms for data repairing; at the same time, it considerably extends its reach in both activities; in fact, on the one side it brings a powerful addition to schema mappings, by allowing for sophisticated conflict resolution strategies, authoritative sources, and non-empty target databases; on the other side, it extends data repairing to a larger classes of constraints, especially inclusion dependencies and conditional inclusion dependencies [FG12] that are very important in the management of referential integrity constraints, and for which very little work exists.

In order to do this, we introduce a language to specify constraints based on *equality generating dependencies (egds)* [BV84] that generalizes many of the data quality constraints used in the literature. Besides standardizing the language to express dependencies, this syntax has two merits. On the one side, it emphasizes the relationship between the data repairing process and the logical language of embedded dependencies, thus suggesting a path to define

a chase-like procedure to repair the database. On the other side, it brings an important extension to the class of constraints that can be handled by the formalism, since it allows one to express inter-table constraints. In the presence of such constraints, previous works had no other alternative than joining the two original tables, and defining standard dependencies over the join result. This, however, is a strong drawback, since it forces to work with large, non-normalized tables, for which data-repairing times are even worse.

2.2 Semantics, Preference Rules and Partial Orders

At the core of the framework stands a novel semantics for the mapping and cleaning scenarios. The definition of such a semantics is far from trivial. The crux of our approach consists in formalizing the process of executing mappings and repairing constraints as the process of *upgrading* the quality of an instance.

In data repairing, whenever a violation to a constraint is detected under the form of conflicting values, a general problem consists in picking-up a "preferred" value to repair the database. Consider FD d_1 in our example. To repair the target database one may want to equate $t_5[\mathsf{Phone}]$ and $t_6[\mathsf{Phone}]$. The FD does not tell, however, to which phone number these attribute values should be repaired: '122-1876' or '000-0000', or even a completely different value. As it happens in this kind of problems, we assume that the Phone attribute values in the Customers table come with a *confidence* (*Conf.*) value ('122-1876' has confidence 0.9 and the dummy value '000-0000' has confidence 0). Relying on confidence is a typical strategy to select preferred values: if we assume that one prefers values with higher confidence, we can repair $t_6[\mathsf{Phone}]$ by changing it to '122-1876'.

There are, however, other possibilities. For example, when working with the Treatments table, we may use dates of treatments to infer the currency of other attributes. If the target database is required to store the most recent value for the salary by FD $d_7 = (\mathsf{SSN} \rightarrow \mathsf{Salary})$, this may lead us to repair the obsolete salary value '10K' in t_7 with the more recent (and preferred) value '25K' in t_8.

Notice that we don't always have a clear policy to choose preferred values. For example, when repairing $t_5[\mathsf{CC\#}]$ and $t_6[\mathsf{CC\#}]$ for FD d_2, there is no information available to resolve the conflict. This means that the best we can do is to "mark" the conflict, and then, perhaps, ask for user-interaction in order to solve it.

Previous works have proposed several algorithms that incorporate strategies to select preferred values, like the ones discussed above. However, these algorithms tend to hard-code the way in which preferred values are used for the purpose of repairing the database. As a consequence, there is no way to incorporate the different strategies in a principled way. As a consequence, aside from the generic notion of a repair as an updated database that satisfies the constraints, it is not possible to say what represents a "good" repair in the general case.

Our solution to this problem builds on two main concepts. First, we show that seeing repairs simply as cell updates is not sufficient. On the contrary, we introduce the new notion of a *cell group*, that is essentially a "partial repair with lineage". Then, we formalize the process of improving the quality of a database by introducing a very general notion of a *partial order* over cell groups; the partial order nicely abstracts all of the most typical strategies to decide when a value should be preferred to another, including master data, certainty, accuracy, freshness and currency. In the thesis, we show how users can easily plug-in their preference strategies for a given scenario into the semantics. Finally, by introducing a new category of values, called *lluns*, we are able to complete the lattice of instances induced by the partial order, and to provide a natural hook for incorporating user feedbacks into the process.

2.3 The Chase and Scalability

We introduce the notion of a *minimal solution* and develop algorithms to compute minimal solutions, based on a parallel-chase procedure. This has the advantage of building on a popular and principled algorithmic approach, but it has a number of subtleties. In fact, our chase procedure is more sophisticated than the standard one, in various respects. To give an example, we realize that in the presence of inconsistencies user inputs may be crucial. To this aim, we introduce a nice abstraction of user inputs and show how it can be seamlessly integrated into the chase. This may pave the way to the development of new tools for data repairing, in the spirit of [CT06].

In addition, scalability is a primary concern of this work. Given the complexity of our chase procedure, addressing this concern is quite challenging. Therefore, we introduce a number of new optimizations to alleviate computing times, making the chase a viable option to exchange and repair large databases. A key ingredient of our solution is the development of an ad-hoc representation systems for chase trees, called *delta relations*.

In addition, we introduce a notion of a *cost manager* as a plug-in for the chase algorithm that selects which repairs should be kept and which ones should be discarded. The cost manager abstracts and generalizes all of the popular solution-selection strategies, including similarity-based cost, set-minimality, set-cardinality minimality, certain regions, sampling, among others. In Example 1, our semantics generates minimal solutions like the two solutions in Figure 1.2, where L_i values represent lluns (confidence values have been omitted); notice that other minimal solutions exist for this example. Cost managers allow users to differentiate between these two solutions, which have completely different costs in terms of chase computation, and ultimately to fine-tune the tradeoff between quality and scalability of the repair process.

In our experiments, we show that the chase engine scales to databases with millions of tuples, a considerable advancement in scalability wrt previous main-memory implementations. In fact, as a major result, we show in our experiments that the chase engine is orders of magnitude faster than existing engines for data exchanges, and show superior scalability wrt previous algorithms for data repairing [BFFR05, BIG10] that were designed to run in main memory.

We compare our semantics to many previous approaches (Sections 7.6 and 10). To the best of our knowledge, this is the first proposal that achieves the level of generality needed to handle three different kinds of problems: traditional mapping problems, traditional data repairing problems, and the new and more articulated category of data translation problems with conflict resolution, as exemplified in Example 1. We believe that these contributions make a significant advancement with respect to the state-of-the-art, and may bring new maturity to both schema mappings and data repairing.

Chapter 3

Extended Dependencies

3.1 Background

Database instances A *schema* $\mathcal{R}$ is a finite set $\{R_1, \ldots, R_k\}$ of relation symbols, with each R_i having a fixed arity $n_i \geq 0$. Let CONSTS be a countably infinite domain of constant values, typically denoted by lowercase letters a, b, c, ... Let NULLS be a countably infinite set of labeled nulls, distinct from CONSTS. Constants are typically denoted by lowercase letters a, b, c, ..., and nulls by N_1,N_2, N_3,.... An *instance* $I = (I_1, \ldots, I_k)$ of $\mathcal{R}$ consists of finite relations $I_i \subset (\text{CONSTS} \cup \text{NULLS})^{n_i}$, for $i \in [1, k]$. Let R be a relation symbol in $\mathcal{R}$ with attributes $A_1, \ldots, A_n$ and I an instance of R. A *tuple* is an element of I and we denote by $t.A_i$ the value of tuple t in attribute A_i. Furthermore, we always assume the presence of *unique tuple identifiers* for tuples in an instance. That is, t_{tid} denotes the tuple with id "tid" in I. A *cell* is a location in I specified by a tuple id/attribute pair $t_{tid}.A_i$. Given two disjoint schemas, $\mathcal{S}$ and $\mathcal{T}$, if I is an instance of $\mathcal{S}$ and J is an instance of $\mathcal{T}$, then the pair $\langle I, J\rangle$ is an instance of $\langle \mathcal{S}, \mathcal{T}\rangle$. The *value* of a cell $t_{tid}.A_i$ in I is the value of attribute A_i in tuple t_{tid}.

Dependencies A relational atom over $\mathcal{R}$ is a formula of the form $R(\overline{x})$ with $R \in \mathcal{R}$ and $\overline{x}$ is a tuple of (not necessarily distinct) variables. A *tuple-generating dependency (tgd)* over $\mathcal{R}$ is a formula of the form $\forall\, \overline{x}\big(\phi(\overline{x}) \rightarrow \exists\, \overline{y}\psi(\overline{x}, \overline{y})\big)$, where $\phi(\overline{x})$ and $\psi(\overline{x}, \overline{y})$ are conjunctions of relational atoms over $\mathcal{R}$. Given two disjoint schemas, $\mathcal{S}$ and $\mathcal{T}$, a tgd over $\langle \mathcal{S}, \mathcal{T}\rangle$ is called a *source-to-target tgd (s-t tgd)* if $\phi(\overline{x})$ only contains atoms over $\mathcal{S}$, and $\psi(\overline{x}, \overline{y})$ only contains atoms over $\mathcal{T}$.

Furthermore, a *target tdg* is a tgd in which both $\phi(\overline{x})$ and $\psi(\overline{x}, \overline{y})$ only contain atoms over $\mathcal{T}$. An *equality generating dependency (egd)* over $\mathcal{T}$ is a formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow x_i = x_j)$ where $\phi(\overline{x})$ is a conjunction of relational atoms over $\mathcal{T}$ and x_i and x_j occur in $\overline{x}$. In the sequel, we omit the universal quantification in front of tgds and egds and simply write $\phi(\overline{x}, \overline{z}) \rightarrow \exists \overline{y}\psi(\overline{x}, \overline{y})$ for tgds, and $\phi(\overline{x}) \rightarrow x_i = x_j$ for egds.

Schema Mapping A *mapping scenario* [FKMP05] (aka a *data-exchange scenario*) is a quadruple $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \Sigma_{st}, \Sigma_t)$, where $\mathcal{S}$ is a source schema, $\mathcal{T}$ is a target schema, Σ_{st} is a set of source-to-target tgds, and Σ_t is a set of target dependencies that may contain target tgds and egds.

An instance $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{T}\rangle$ *satisfies* a tgd or egd following the standard semantics of first-order logic. Given a set Σ of tgds and egds, $\langle I, J\rangle$ satisfies Σ, denoted by $\langle I, J\rangle \models \Sigma$, if it satisfies all dependencies in Σ. A target instance J of $\mathcal{T}$ is a *solution* of $\mathcal{M}$ and a source instance I of $\mathcal{S}$, denoted by $J \in \mathsf{Sol}(\mathcal{M}, I)$, iff $\langle I, J\rangle$ satisfies Σ (following the standard semantics of first-order logic), i.e., $\langle I, J\rangle \models \Sigma_{st}$, and $J \models \Sigma_t$.

Given two instances J, J' over a schema $\mathcal{T}$, a *homomorphism* $h : J \rightarrow J'$ is a mapping from $dom(J)$ to $dom(J')$ such that for each $c \in consts(J)$, $h(c) = c$, and for each tuple $t \in J$ it is the case that $h(t) \in J'$. A solution J is regarded to be *more general* than a solution J', if there exists a homomorphism from J to J'. A *universal solution* for $\mathcal{M}$ and I is a solution that is more general than any other solution [FKMP05]. The set of universal solutions for $\mathcal{M}$ and I is denoted by $\mathsf{USol}(\mathcal{M}, I)$. The *core universal solution* for $\mathcal{M}$ and I is the smallest, in terms of number of tuples, universal solution and is unique up to isomorphism [FKP05].

The *chase* is a procedure to generate universal solutions. We assume the standard definitions of a *chase step, chase sequence*, and *chase result* for mappings, as given in [FKMP05].

3.2 Cleaning EGDs and Extended TGDs

A first, important step in the definition of our framework for mapping and cleaning consists in the definition of a unified language for mappings and data quality constraints. Traditional embedded dependencies are extensively studied and used when integrating and exchanging data. They fall short, however, when it comes to the constraint formalisms used in the context of data quality.

For example, the constraints d_4, d_5 and d_6 described in Example 1 cannot be directly expressed as egds.

To alleviate this problem, we make use of *extended tgds* and *cleaning egds*. Our main extensions to the syntax of ordinary embedded dependencies, is that we freely mix source and target symbols in the premise. In addition, in a cleaning egd, we also consider *equation atoms* of the form $t_1 = t_2$, where t_1, t_2 are either constants in CONSTS or variables.

Egds for our running example are expressed as follows:

e_1. $\mathsf{Cust}(\mathbf{ssn}, \mathbf{n}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{Cust}(\mathbf{ssn}, \mathbf{n}, \mathsf{p'}, \mathsf{s'}, \mathsf{c'}, \mathsf{cc'}) \rightarrow \mathsf{p} = \mathsf{p'}$
e_2. $\mathsf{Cust}(\mathbf{ssn}, \mathbf{n}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{Cust}(\mathbf{ssn}, \mathbf{n}, \mathsf{p'}, \mathsf{s'}, \mathsf{c'}, \mathsf{cc'}) \rightarrow \mathsf{cc} = \mathsf{cc'}$
e_3. $\mathsf{Cust}(\mathsf{ssn}, \mathbf{n}, \mathsf{p}, \mathbf{s}, \mathbf{c}, \mathsf{cc}), \mathsf{Cust}(\mathsf{ssn'}, \mathbf{n}, \mathsf{p'}, \mathbf{s}, \mathbf{c}, \mathsf{cc'}) \rightarrow \mathsf{ssn} = \mathsf{ssn'}$
e_4. $\mathsf{Treat}(\mathsf{ssn}, \mathsf{s}, \mathsf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{ins} = \text{`Abx'} \rightarrow \mathsf{tr} = \text{`Dental'}$
e_5. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathbf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{MD}(\mathbf{ssn}, \mathsf{n'}, \mathbf{p}, \mathsf{s'}, \mathsf{c'}) \rightarrow \mathsf{n} = \mathsf{n'}$
e_6. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathbf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{MD}(\mathbf{ssn}, \mathsf{n'}, \mathbf{p}, \mathsf{s'}, \mathsf{c'}) \rightarrow s = s'$
e_7. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathbf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{MD}(\mathbf{ssn}, \mathsf{n'}, \mathbf{p}, \mathsf{s'}, \mathsf{c'}) \rightarrow c = c'$
e_8. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{Treat}(\mathbf{ssn}, \mathsf{sal}, \mathsf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{ins} = \text{`Abx'} \rightarrow \mathsf{c} = \text{`SF'}$
e_9. $\mathsf{Treat}(\mathbf{ssn}, \mathsf{s}, \mathsf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{Treat}(\mathbf{ssn}, \mathsf{s'}, \mathsf{ins'}, \mathsf{tr'}, \mathsf{d'}) \rightarrow \mathsf{s} = \mathsf{s'}$

Before we formalize this notion, an immediate observation is that constants and equation atoms in the premise can be avoided altogether, by encoding them in additional tables in the source database. Consider dependency e_4 in our example in which two constants appear: 'Abx' in attribute Insur and 'Dental' in attribute Treat. We extend $\mathcal{S}$ with an additional binary source table, denoted by Cst_{e4} with attributes Insur and Treat, corresponding to the "constant" attributes in e_4. Furthermore, we instantiate Cst_{e4} with the single tuple t_{e_4} : (Abx, Dental). Given this, e_4 can be expressed as an egd without constants. Similarly for e_8, as follows:

e'_4. $\mathsf{Treat}(\mathsf{ssn}, \mathsf{s}, \mathbf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{Cst}_{e_4}(\mathbf{ins}, \mathsf{tr'}) \rightarrow \mathsf{tr} = \mathsf{tr'}$
e'_8. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{Treat}(\mathbf{ssn}, \mathsf{sal}, \mathbf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{Cst}_{e_8}(\mathbf{ins}, \mathsf{c'}) \rightarrow \mathsf{c} = \mathsf{c'}$

In general, $\mathcal{S}$ can be extended with such constants tables, one for each CFD, and their source tables contain tuples for the constants used to define the CFD. In other words, these tables coincide with the pattern tableaux associated with the CFDs [FGJK08]. Of course, one needs to provide a proper semantics of egds such that whenever such constant tables are present, egds have the same semantics as CFDs. We give such semantics later in the thesis.

For the moment we formalize the notion of a cleaning egd as follows:

Definition 1 [CLEANING EGD] A *cleaning egd* over schemas $\mathcal{S}, \mathcal{T}$ is a formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow t_1 = t_2)$ where $\phi(\overline{x})$ is a conjunction of relational atoms

over $\langle \mathcal{S}, \mathcal{T} \rangle$, and $t_1 = t_2$ is of the form $x_i = c$ or $x_i = x_j$, for some variables x_i, x_j in $\overline{x}$ and constant $c \in$ CONSTS. Furthermore, at most one variable in the conclusion of an egd can appear in the premise as part of a relation atom over $\mathcal{S}$.

The latter condition is to ensure that the egd specifies a constraint on the target database rather than on the fixed source database. With an abuse of notation, in the following we shall often refer to these cleaning egds simply as egds.

Notice that additional tables that hold constant values are considered as part of the source database. Since they encode a constraint on the data that is explicitly specified by the dependencies, we want to treat them as sources of high reliability, similarly to what happens with master-data.

A similar treatment is also done for tgds. To properly encode data quality constraints, like, for example, conditional inclusion dependencies [BFM07], we need to be able to specify equation atoms in the dependency premise. Suppose, for example, that our target database also contains a DentalProsthesis table, in which we store all prosthesis that have been installed as part of dental treatments. We might want to specify a conditional inclusion dependency of this form:

$$m_4.\, \mathsf{Treat}(\mathsf{ssn}, \mathsf{sal}, \mathsf{ins}, \mathsf{tr}, \mathsf{date}), \mathsf{tr} = \text{`}\mathsf{Dental}' \rightarrow \mathsf{DenProst}(\mathsf{ssn}, \ldots)$$

This is rewritten to get rid of the constant as follows:

$$m_4'.\, \mathsf{Treat}(\mathsf{ssn}, \mathsf{sal}, \mathsf{ins}, \mathsf{tr}, \mathsf{date}), \mathsf{Cst}_{m_4}(\mathsf{tr}) \rightarrow \mathsf{DenProst}(\mathsf{ssn}, \ldots)$$

As usual, Cst_{m_4} is a source table with a single tuple with a single attribute of value 'Dental'. In light of this, we define the notion of an extended tgd:

Definition 2 [EXTENDED TGD] An *extended tgd* over schemas $\mathcal{S}, \mathcal{T}$ is a formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow \exists \overline{y}\psi(\overline{x}, \overline{y}))$ where $\phi(\overline{x})$ is a conjunction of relational atoms over $\langle \mathcal{S}, \mathcal{T} \rangle$, and $\psi(\overline{x}, \overline{y})$ is a conjunction of relational atoms over $\mathcal{T}$.

Notice that extended tgds generalize both traditional s-t tgds and target tgds. Further extensions of dependencies with, e.g., built-in predicates, matching functions and negated atoms, are needed to encode matching dependencies and constraints for numerical attributes [FFP10, CIP13]. We do not consider them in this thesis for simplicity of exposition.

In the following sections, we formalize the notion of a mapping and cleaning scenario with extended tgds and cleaning egds, and then provide a semantics

for it. Before we turn to this, we need to introduce an important result, that motivates the need for a new semantics.

One may wonder why a new semantics is needed after all. Indeed, why can't we simply rely on the standard semantics for tgds [FKMP05], and on known data repairing algorithms, like those in [GMPS13], [BFFR05] or [BIG10]? As an example, let Σ_t be a set of tgds and Σ_e be a set of egds, and I and J instances of a source and target schema, $\mathcal{S}$ and $\mathcal{T}$, respectively. Assume that we simply pipeline the chase of tgds, $\mathsf{chase}^{de}_{\Sigma_t}$, [FKMP05], and a repair algorithm for egds, $\mathsf{repair}_{\Sigma_e}$, treated as functional dependencies, as reported in Figure 3.1.

$\mathsf{pipeline}_{\Sigma_t \cup \Sigma_e}(\langle I, J \rangle)$
  $\langle I, J_{tmp} \rangle := \langle I, J \rangle$;
  while (true)
    $\langle I, J_{tmp} \rangle := \mathsf{chase}^{de}_{\Sigma_t}(\langle I, J_{tmp} \rangle)$;
    $\langle I, J_{tmp} \rangle := \mathsf{repair}_{\Sigma_e}(\langle I, J_{tmp} \rangle)$;
    if $(\langle I, J_{tmp} \rangle \models \Sigma_t \cup \Sigma_e)$ return $Sol := J_{tmp}$;
  end while

Figure 3.1: The pipeline algorithm.

Unfortunately, interactions between tgds and egds often prevent that pipelining the two semantics returns a solution, as illustrated by the following proposition.

Proposition 1 *There exist sets Σ_t of non-recursive tgds, Σ_e of egds, and instances $\langle I, J \rangle$ such that procedure $\mathsf{pipeline}_{\Sigma_t \cup \Sigma_e}(\langle I, J \rangle)$ does not return solutions.*

In addition, as we will show in our experiments, even in those cases in which $\mathsf{pipeline}_{\Sigma_t \cup \Sigma_e}(\langle I, J \rangle)$ does return a solution, its quality is usually rather poor. Even worse, since we are combining two rather different algorithms without a formal semantics, it is not even clear what a "good" solution is. These arguments justify the need for a definition of a notion of a mapping and cleaning scenario, and of a new semantics for it.

Chapter 4

Mapping and Cleaning Scenarios

Our uniform framework for schema mapping and data repairing is centered around the concept of a *mapping & cleaning scenario*. A mapping and cleaning scenario consists essentially of a source schema $\mathcal{S}$, a target schema $\mathcal{T}$, and a set of constraints Σ, that may be extended tgds and cleaning egds. There are however three other main ingredients in a mapping and cleaning scenario, namely *lluns*, *user inputs*, and *partial order specifications*. These are introduced next.

4.1 LLUNs

We assume that the target database may contain values of different kinds. To start, besides constants from CONSTS, we also allow target instances to take values from a third set of values, called *lluns*. Recall from Example 1 that t_5 and t_6 form a violation for the dependency e_2 (customers with equal ssns and names should have equal credit-card numbers), and that the target database could be repaired by equating $t_5.\text{CC\#} = t_6.\text{CC\#}$. However, no information is available as to which value should be taken in the repair. In such case, we repair the target database (for e_2) by changing $t_5.\text{CC\#}$ and $t_6.\text{CC\#}$ into the llun L_0, that is to indicate that we need to introduce a new value that may be either 781658 or 784659, or some other preferred value. In this case, such value is currently unknown and we mark it so that it might be resolved later on into a constant, e.g., by asking for user input.

We denote by $\text{LLUNS} = \{L_1, L_2, \ldots\}$ an infinite set of symbols, called *lluns*, distinct from CONSTS and NULLS. Lluns can be regarded as the opposite of

nulls since lluns carry "more information" than constants. In our approach, they play two important roles: (*i*) they allow us to complete the lattice induced by our partial orders, as it will be discussed in the next section; (*ii*) they provide a clean way to record inconsistencies in the data that require the intervention of users.

4.2 User Inputs

We abstract user inputs by seeing the user as an oracle. More formally:

Definition 3 [User-Input Function] We call a *user-input function* a partial function User that takes as input any set of cells, $\mathcal{C}$, i.e., tuple-attribute pairs with a value, and returns one of the following values, denoted by $\mathsf{User}(\mathcal{C})$:

- v, to denote that the value of the cells in $\mathcal{C}$ should be changed to value $v \in$ consts;
- $\perp$, to denote that changing the cells in $\mathcal{C}$ would represent an incorrect modification to the database, and therefore the change should not be be performed.

Notice that User is by definition a partial function, and therefore it may be undefined for some sets of cells.

4.3 The Partial Order Specification

A key idea in our approach is that the strategy to select preferred values and repair conflicts should be factored-out of the actual repairing algorithm. Our solution to do that is to introduce a notion of a *partial order* over updates to the database. The partial order plays a central role in our semantics, since it allows us to identify when a repair is an actual "upgrade" of the original database.

In the definition of a mapping and cleaning scenario, we assume that a (possibly empty) *specification* of this partial order, Π, is provided by the user. We want users to be able to specify different partial orders for different scenarios in a simple manner. To do this, users may specify preference rules by providing an assignment Π of so-called *ordering attributes* to $\mathcal{T}$.

We say that an attribute A of $\mathcal{T}$ has *ordered values* if its domain $\mathcal{D}_A$ is a partially ordered set. To specify which values should be preferred during the repair of the database, users may associate with each attribute A_i of $\mathcal{T}$ a

partially ordered set $\mathcal{P}_{A_i} = \langle D, \leq \rangle$. The poset $\mathcal{P}_{A_i}$ associated with attribute A_i may be the empty poset, or its domain $\mathcal{D}_{A_i}$ if A_i has ordered values, or the domain of a different attribute $\mathcal{D}_{A_j}$ that has ordered values. In the latter case, we call A_j the *ordering attribute* for A_i. Intuitively, $\mathcal{P}_{A_i}$ specifies the order of preference for values in the cells of A_i. A *partial-order specification* is an assignment of ordering attributes to attributes in $\mathcal{T}$, denoted by Π.

In our example, the Date attribute in the Treatments table, and the confidence column, Conf, in the Customers table have ordered values. For these attributes, we choose the corresponding domain as the associated poset (i.e., we opt to prefer more recent dates and higher confidences). Other attributes, like the Phone attribute in the Customers table, have unordered values; we choose Conf as the ordering attribute for Phone (a phone number will be preferred if its corresponding confidence value is higher). Notice that there may be attributes, like Salary in Treatments, that have ordered values but the natural ordering of values does not coincide with the desired notion of a preferred value. Here, we may rather prefer most recent salaries and hence use Date as the ordering attribute for Salary. Finally, attributes like ssn will have an empty associated poset, i.e., all constant values are equally preferred.

Below is the assignment Π of ordering attributes in our example (attributes not listed have an empty poset):

$$\Pi = \left\{ \begin{array}{lcl} \mathcal{P}_{\text{Customers.Conf}} & = & \mathcal{D}_{\text{Customers.Conf}} \\ \mathcal{P}_{\text{Treatments.Date}} & = & \mathcal{D}_{\text{Treatments.Date}} \\ \mathcal{P}_{\text{Customers.Phone}} & = & \mathcal{D}_{\text{Customers.Conf}} \\ \mathcal{P}_{\text{Treatments.Salary}} & = & \mathcal{D}_{\text{Treatments.Date}} \\ \mathcal{P}_{\text{Customers.cc\#}} & = & \emptyset \end{array} \right\}$$

Notice that a similar treatment can also be done for the source schema $\mathcal{S}$. More specifically, we assume two source schemas, $\mathcal{S}$ and $\mathcal{S}_a$. This second schema, $\mathcal{S}_a$, is a set of *authoritative tables* that provide clean and reliable information as input for the repairing process. This include master-data tables and constant tables introduced to remove constants from dependencies (as discussed in Section 3). Authoritative tables are considered all as equally reliable, since they contain "certified" tuples. On the contrary, a partial order specification can be specified on the attributes of $\mathcal{S}$ in order to state preference relations on source values as well.

The role of the partial order specification is to induce a partial order for the cells of the initial instance, $\langle I, J \rangle$. This will be detailed in Section 7.1.

With this in mind, we introduce the notion of a mapping and cleaning scenario as follows.

Definition 4 [MAPPING&CLEANING SCENARIO] Given a domain $\mathcal{D} = \text{CONSTS} \cup \text{NULLS} \cup \text{LLUNS}$, a *mapping & cleaning scenario* over $\mathcal{D}$ is a tuple $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$, where:

1. $\mathcal{S} \cup \mathcal{S}_a$ is the source schema; $\mathcal{T}$ is the target schema; $\mathcal{S}_a$ is the set of *authoritative source tables*;
2. Σ_t is a set of *extended tgds*, as defined in Section 3.1;
3. Σ_e is a set of *cleaning egds*, as defined in Section 3.1;
4. Π is a partial-order specification for attributes of $\mathcal{S}$ and $\mathcal{T}$;
5. User is a partial function as defined in Definition 3.

If the set of tgds, Σ_t, is empty, $\mathcal{MC}$ is called a *cleaning scenario*.

It is readily verified that Example 1 can be regarded as an instance of a mapping & cleaning scenario. Given a mapping & cleaning scenario and instance $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{S}_a, \mathcal{T}\rangle$, the goal is to compute a set of repair instructions of the target that upgrades the initial dirty target instance J and satisfies the dependencies in Σ_t and Σ_e. We assume that: (i) the source instance, I, only contains constants from CONSTS, and is immutable, i.e., its cells cannot be changed; (ii) the target instance, J, may contain constants from CONSTS and nulls from NULLS. Later on during the repair process, lluns can be used to update the target. Notice that Π and User can be empty, i.e., users may decide to provide no additional information but the set of dependencies to satisfy. However, whenever they are not empty, we expect the semantics to use this input to "improve" the quality of the repairs. We next describe the necessary tools to achieve this goal.

Chapter 5

Cell Groups and Updates

Given instance $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{S}_a, \mathcal{T}\rangle$, we represent the set of changes made to update the target database J in terms of *cell groups*. Cell groups are crucial in defining our semantics.

As the name suggests, they are essentially groups of *cells*, i.e., locations in a database specified by tuple/attribute pairs $t_{\mathsf{tid}}.A_i$. For example, t_5.CC# and t_6.CC# are two cells in the Customers table. As we have previously seen, to repair inconsistencies, different cells are often updated *together*, i.e., they are either changed all at the same time or not changed at all. For example, t_5.CC# and t_6.CC# are both modified to the same llun value in solution #1 in Figure 1.2. Cell groups capture this by specifying a set of target cells, called *occurrences* of the group, and a value to update them.

However, cell groups are more sophisticated than that since they also model relationships among target and source values. In some cases the target cells to repair receive their value from tuples in the source database; consider Example 1 and dependency e_6. When repairing t_5, cell t_5.Street gets the value 'Sky Dr.' from cell t_m.Street in the master-data table. Since these source cells contain highly reliable information, it is important to keep track of the relationships among changes to target cells and values in the source. To do this, a cell group carries provenance information about the repair in terms of associated cells of the source database. We call these source cells the *justifications* for the cell group.

In addition, cell groups model different ways to modify the target database. In our approach, this can be done in two ways: (*i*) by changing cell values to enforce egds; (*ii*) by adding new tuples as an effect of enforcing tgds. We there-

fore need to record into cell groups also new cells added to the target database by tgds. These will be part of the occurrences, but need to be distinguishable, since they do not have a value in the original target instance.

Furthermore, notice that there are two different strategies to remove violations for a dependency. The ones we have discussed so far are called *forward changes*, since they amount to changing cells to satisfy the conclusion of an egd. Consider egd e_2 that states that customers with equal SSNs must have equal credit-card numbers. The forward change to solve a violation equates the values of the CC# attribute of the conflicting tuples. However, as an alternative, one may introduce *backward changes* to falsify the premise instead. In our example, this requires to set the value of attribute ssn in either of the conflicting tuples to a llun value to say that the original value is dirty, and should be changed to something else that we currently do not know. Special care is needed when we want to use cell groups to backward-change the value of a cell, as will be discussed in the following section. For this, we clearly separate cell groups with backward changes from the others.

We specify an update by providing the original target database together with the set of cell groups encoding the modification. In other words, cell groups can be seen as partial updates with lineage. Notice how they also provide an ideal basis to plug-in user-specified changes. In fact, our user function, User, works with cell groups to modify their values when needed.

These observations are captured by the following definitions. Let $\langle I, J\rangle$ be an instance of $\langle \mathcal{S}, \mathcal{S}_a, \mathcal{T}\rangle$. We denote by *cells*$(I)$, *auth-cells*$(I)$ and *cells*(J) the set of all cells in the source tables in I, the set of cells in authoritative tables in I, and the set of cells in the target tables in J, respectively. Let *new-cells*(J) denote the (infinite) set of cells corresponding to all tuples over $\mathcal{T}$ that are not in J. Intuitively, *new-cells*(J) represents possible insertions in J needed to satisfy tgds. We assume that each cell in *new-cells*(J) is initialized each to a different null value from NULLS.

Definition 5 [CELL GROUP] A *cell group* g over $\langle I, J\rangle$ is a quadruple $\langle v, occ(g), just(g), isBckw(g)\rangle$, where:

1. $v = val(g)$ is a value in CONSTS $\cup$ NULLS $\cup$ LLUNS;
2. $occ(g)$ is a finite set of cells in *cells*$(J) \cup$ *new-cells*(J), called the *occurrences* of g; of these, we call *target-cells*(g) the ones that appear in J, and *new-cells*(g) the new ones from *new-cells*(J);
3. $just(g)$ is a finite set of cells in *cells*(I), called the *justifications* of g; of these, we call *auth-cells*(g) the ones that belong to authoritative tables;

4. *isBckw*(g) is a boolean value; if *isBckw*$(g) = true$ we say that g has *backward changes*.

We denote by *cells*(g) the set *occ*$(g) \cup$ *just*(g). In writing cell groups, we find it useful to adopt the following notation. First, a cell group g can be read as "change (or insert) the cells in *occ*(g) to value *val*(g), justified by the values in *just*(g)". We therefore shall often write a cell group as $g = \langle v \rightarrow$ *occ*(g), *by just*$(g)\rangle$ for the sake of readability. We shall use the symbol bckw to indicate that *isBckw*$(g) = true$. In addition, we use superscripts to denote new and authoritative cells. Subscripts are used to report the value of a cell in the original databases, I or J, so that the modifications specified by cell groups are easier to interpret. Following are some examples of cell groups.

Example 2: Consider a sample scenario with a source table $S(A, B)$ and a target table $T(A, B, C)$, and two constraints:

$$m_1 : S(x, y) \rightarrow \exists z : T(x, y, z)$$
$$m_2 : T(x, y, z), T(x, y', z') \rightarrow y = y'$$

Given an initial instance $t_1 : S(1, 2), t_2 : S(1, 3)$, suppose T is empty. The insert of tuple t_3 with values from t_1 into the target (according to m_1) may be expressed using the following cell groups:

$$g_1 : \langle 1 \rightarrow \{t_3.\mathsf{A}^{new}\}, by\ \{t_1.\mathsf{A}_{[1]}\}\rangle$$
$$g_2 : \langle 2 \rightarrow \{t_3.\mathsf{B}^{new}\}, by\ \{t_1.\mathsf{B}_{[2]}\}\rangle$$
$$g_3 : \langle N_1 \rightarrow \{t_3.\mathsf{C}^{new}\}, by\ \emptyset\}\rangle$$

Similarly, the insert of tuple t_4 with values from t_2 can be modeled by three more cell groups $g_4 - g_6$. Whenever a set of cell groups creates new tuples in the target, it generates a new set of tuples that we call ΔJ. In this example: $\Delta J = \{t_3 : T(1, 2, N_1), t_4 : T(1, 3, N_2)\}$. Notice how new cells may either contain the copy of values coming from source cells or new, labeled nulls. Constant values from the source are recorded in cell groups by means of justifications. On the contrary, new cells with a null value have empty justifications.

If we update the target with $g_1 - g_6$, the two new tuples t_3, t_4 violate the egd. A cell group that enforces the egd over cells $t_3.B, t_4.B$ is as follows:

$$g_7 : \langle L_1 \rightarrow \{t_3.\mathsf{B}^{new}, t_4.\mathsf{B}^{new}\}, by\ \{t_1.\mathsf{B}_{[2]}, t_2.\mathsf{B}_{[3]}\}\rangle$$

Otherwise, we may remove the violation by backward-changing cell $t_3.A$:

$$g_8 : \langle L_2 \rightarrow \{t_3.\mathsf{A}^{new}\}, by\ \{t_1.\mathsf{A}_{[1]}\}, \mathsf{bckw}\rangle$$

Example 3: Consider now our motivating Example 1. Here are further examples of cell groups:

$$g_1 : \langle L_0 \rightarrow \{t_5.\mathsf{CC}\#_{[781658]}, t_6.\mathsf{CC}\#_{[784659]}\},\ by\ \emptyset\rangle$$
$$g_2 : \langle Dental \rightarrow \{t_8.\mathsf{Treat}_{[Cholest]}\},\ by\ \{t_{e_4}.\mathsf{Treat}^{auth}_{[Dental]}\}\rangle$$
$$g_3 : \langle Med \rightarrow \{t_{12}.\mathsf{Insur}^{new}\},\ by\ \{t_2.\mathsf{Insur}_{[Med]}\}\rangle$$
$$g_4 : \langle null \rightarrow \{t_{12}.\mathsf{Salary}^{new}\},\ by\ \emptyset\rangle$$
$$g_5 : \langle L_3 \rightarrow \{t_7.\mathsf{Insurance}_{[Abx]}\},\ by\ \emptyset, \mathsf{bckw}\rangle\}$$

As you can see, g_1 changes two conflicting cells of the target database into a llun value. The justification in g_2 can be intuitively explained by seeing that the cell is repaired according to egd e_4, stating that company 'Abx' only provides dental treatments. Recall that in our approach this is expressed by encoding constants as additional authoritative tables in the source database, Cst_{e_4} in this example, with a single tuple t_{e_4}. Given this, we can express e_4 without constants as:

$$e'_4.\ \mathsf{Treat}(\mathsf{ssn}, \mathsf{s}, \mathbf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{Cst}_{e_4}(\mathbf{ins}, \mathsf{tr'}) \rightarrow \mathsf{tr} = \mathsf{tr'}$$

Cell groups g_3 and g_4 add new cells to the target, to insert (part of) tuple t_{12}. Notice how, in g_3, the value for the new cell is stored as a justification. Finally, g_5 is an example of a cell group that backward-changes a cell to satisfy dependency e_8.

We consider cell groups to be undistinguishable up to the renaming of nulls and lluns. In fact, we say that g *is equal to* g' if $occ(g) = occ(g'), just(g) = just(g'), isBckw(g) = isBckw(g')$, and: (i) both have the same constant value, i.e., $val(g) = val(g') \in$ CONSTS, or (ii) $val(g), val(g')$ are equally informative, i.e., they are both nulls or lluns.

We define an *update* to an instance $\langle I, J\rangle$ as a set of cell groups. More specifically,

Definition 6 [UPDATE] An *update* Upd of J is a set of cell groups over $\langle I, J\rangle$ such that there exists a set of tuples ΔJ, distinct from J, for which:

1. for each $g = \langle v, occ(g), just(g)\rangle$ in Upd, $occ(g) \subseteq cells(J \cup \Delta J)$. That is, cell groups in Upd are restricted to updates of cells in $J \cup \Delta J$.

2. each cell in J occurs at most once in Upd, and each cell in ΔJ occurs exactly once in Upd. That is, two cell groups in Upd cannot update the same cell in the target, and ΔJ is completely specified by Upd.

3. each null (resp. llun) value occurs at most once as a value of a cell group in Upd. That is, null and llun values uniquely identify the cells in which they appear.

We denote by $\mathsf{Upd}(J)$ the target instance obtained by adding to J the tuples in ΔJ, and then changing the values in the new instance as specified by Upd.

Example 4: Consider our Example 2 above. Following are two updates $\mathsf{Upd}_1, \mathsf{Upd}_2$ that enforce the constraints:

$$\Delta J = \{t_3 : T(1, 2, N_1), t_4 : T(1, 3, N_2)\}$$
$$\mathsf{Upd}_1 = \{g_1, g_3, g_4, g_6, g_7\}$$
$$\mathsf{Upd}_2 = \{g_2, g_3, g_4, g_5, g_6, g_8\}$$

Example 5: Consider the table Treatments from Example 1 and corresponding table in solution #1 shown in Figure 1.2. It is easy to see that the repaired instance can be seen as an update. For example, it can be regarded as $\mathsf{Upd}_1(\mathsf{Treatments})$. Following are some cell groups from Upd_1:

$$g_1 : \langle \mathit{Dental} \rightarrow \{t_8.\mathsf{Treat}_{[\mathit{Cholest}]}\},\ \mathit{by}\ \{t_{e_4}.\mathsf{Treat}^{\mathit{auth}}_{[\mathit{Dental}]}\}\rangle,$$
$$g_2 : \langle \mathit{25K} \rightarrow \{t_7.\mathsf{Salary}_{[10K]}, t_8.\mathsf{Salary}_{[25K]}\},\ \mathit{by}\ \emptyset\rangle,$$
$$g_3 : \langle L_1 \rightarrow \{t_{10}.\mathsf{SSN}^{new}, t_{11}.\mathsf{SSN}^{new}, t_{12}.\mathsf{SSN}^{new}, t_{13}.\mathsf{SSN}^{new}\},$$
$$\mathit{by}\ \{t_1.\mathsf{SSN}_{[123]}, t_2.\mathsf{SSN}_{[123]}, t_3.\mathsf{SSN}_{[124]}\}\rangle,$$
$$g_4 : \langle \mathit{null} \rightarrow \{t_{12}.\mathsf{Salary}^{new}, t_{13}.\mathsf{Salary}^{new}\},\ \mathit{by}\ \emptyset\rangle,$$
$$g_5 : \langle \mathit{Med} \rightarrow \{t_{12}.\mathsf{Insur}^{new}\},\ \mathit{by}\ \{t_2.\mathsf{Insur}_{[\mathit{Med}]}\}\rangle,$$
$$g_6 : \langle \mathit{Eye\ surg.} \rightarrow \{t_{12}.\mathsf{Treat}^{new}\},\ \mathit{by}\ \{t_2.\mathsf{Treat}_{[\mathit{Eye\ surg}]}\}\rangle,$$
$$g_7 : \langle \mathit{12/01/2013} \rightarrow \{t_{12}.\mathsf{Date}^{new}\},\ \mathit{by}\ \{t_2.\mathsf{Date}_{[12/01/2013]}\}\rangle,$$
$$g_8 : \langle \mathit{Med} \rightarrow \{t_{13}.\mathsf{Insur}^{new}\},\ \mathit{by}\ \{t_3.\mathsf{Insur}_{[\mathit{Med}]}\}\rangle,$$
$$g_9 : \langle \mathit{Lapar.} \rightarrow \{t_{13}.\mathsf{Treat}^{new}\},\ \mathit{by}\ \{t_3.\mathsf{Treat}_{[\mathit{Lapar}]}\}\rangle,$$
$$g_{10} : \langle \mathit{03/11/2013} \rightarrow \{t_{13}.\mathsf{Date}^{new}\},\ \mathit{by}\ \{t_3.\mathsf{Date}_{[03/11/2013]}\}\rangle$$

Clearly, other updates are possible. For example, to resolve e_4 one may consider changing the value of the cell t_8.INSURANCE into a new llun value L_3, i.e., an unknown value that improves 'Abx'. The following update, Upd_2, follows the same approach to satisfy all dependencies, and yields the solution #2 shown in Figure 1.2:

$$g'_1 : \langle L_2 \rightarrow \{t_5.\mathsf{SSN}_{[222]}\},\ \mathit{by}\ \emptyset, \mathsf{bckw}\rangle,$$
$$g'_2 : \langle L_3 \rightarrow \{t_7.\mathsf{Insurance}_{[\mathit{Abx}]}\},\ \mathit{by}\ \emptyset, \mathsf{bckw}\rangle,$$
$$g'_3 : \langle L_4 \rightarrow \{t_8.\mathsf{Insurance}_{[\mathit{Abx}]}\},\ \mathit{by}\ \emptyset, \mathsf{bckw}\rangle\}$$
$$g'_4 : \langle L_5 \rightarrow \{t_{10}.\mathsf{Name}^{new}\},\ \mathit{by}\ \{t_1.\mathsf{Name}_{[\mathit{W.\ Smith}]}\}, \mathsf{bckw}\rangle,$$

We say that two updates coincide if their cell groups are identical, up to the renaming of nulls and lluns. We may assume, without loss of generality, that an update is always *complete*, i.e., it specifies values for the entire target instance.

Definition 7 [COMPLETE UPDATE] An update Upd for $\langle I, J\rangle$ is *complete* if each cell of J occurs in a cell group in Upd.

Indeed, any update Upd can be turned into a complete update Upd', as follows:

- initially, we let $\mathsf{Upd}' = \mathsf{Upd}$;
- for each cell c of J that is not changed by Upd, if $val(c) \in$ CONSTS, then we add to Upd' the cell group $\langle val(c) \rightarrow \{c\}, by\ \emptyset\rangle$;
- for each cell c of J that is not changed by Upd, if $val(c) \in$ NULLS and $val(c)$ does not occur in Upd, then we add to Upd' the cell group of c with value $val(c)$, occurrences consisting of all cells of J in which $val(c)$ occurs and empty justifications.

From now on, and without loss of generality, we always assume an update to be complete, and we blur the distinction between an update Upd and the instance $\mathsf{Upd}(J)$ obtained by applying Upd to J. Observe that the initial target instance J can be seen as $\mathsf{Upd}_\emptyset(J)$ where $\mathsf{Upd}_\emptyset$ denotes the trivial update, i.e., no modifications are made.

Chapter 6

The Partial Order of Cell Groups

Cell groups and updates set the stage for the central notion of our semantics: upgrades. As we mentioned, our key intuition is that an update to the database is acceptable only when there is the guarantee that it "improves" the quality of the target. An essential tool, in this respect, is the *partial order* over cell groups and, in turn, updates. In order to do this, we resort to a *hierarchy* of partial orders.

We start with a simple partial order for values. This partial order states that constants are more informative than nulls, and lluns are more informative than constants.

Definition 8 [Partial Order of Values] Given two values $v_1, v_2 \in \text{NULLS} \cup \text{CONSTS} \cup \text{LLUNS}$, we say that v_2 *is more informative* than v_1, if v_1 and v_2 are of different types, and one of the following holds: (i) $v_1 \in \text{NULLS}$, i.e., the first value is a null value; or (ii) $v_2 \in \text{LLUN}$, i.e., the second value is a llun. On the contrary, two values of the same type are *equally informative*.

6.1 The Partial Order of Cells

Based on the *partial order specification*, Π, provided by the user as part of a scenario, we now derive a partial order $\preceq_\Pi$ of the cells in the original database instances, $\langle I, J\rangle$, plus the set of *new-cells*(J). This is crucial to plug-in arbitrary preference strategies during the repair process.

Definition 9 [Partial Order of Cells] Given a partial-order specification Π, and an instance $\langle I, J\rangle$, we can define a corresponding partial order $\preceq_\Pi$ for

the set of cells $\mathcal{C} = \mathit{cells}(I) \cup \mathit{cells}(J) \cup \mathit{new\text{-}cells}(J)$ as follows. For any pair of cells $c_1, c_2 \in \mathcal{C}$ we say that $c_1 \preceq_\Pi c_2$ iff either $c_1 = c_2$ or one of the following holds:

1. $\mathit{val}(c_1) \in$ NULLS, and $\mathit{val}(c_2) \in$ CONSTS;
2. $c_1 \notin \mathit{auth\text{-}cells}(I)$, while $c_2 \in \mathit{auth\text{-}cells}(I)$;
3. c_1 and c_2 are ordered according to Π, that is: $c_1 = t_1.A_1$, $c_2 = t_2.A_2$ in $\langle I, J\rangle$, and both are constants in CONSTS; then, assume the ordering attributes for A_1 and A_2, called A_1', A_2' have the same poset, i.e., $\mathcal{P}_{A_1'} = \mathcal{P}_{A_2'}$; call v_1', v_2' the values of cells $t_1.A_1', t_2.A_2'$. Then, $c_1 \preceq_\Pi c_2$ iff $v_1 = v_2$ or $v_1' < v_2'$ according to $\mathcal{P}_{A_1'} = \mathcal{P}_{A_2'}$.

The following proposition states that relation $\preceq_\Pi$ is in fact a partial order for cells (the proof is in Appendix 14):

Proposition 2 *The binary relation $\preceq_\Pi$ as specified in Definition 9 is a partial order.*

As we have seen, cell groups are made essentially of target cells (occurrences) and source cells (justifications). We expect cell groups to "improve" and "generalize" the values that appear in $\mathit{occ}(g) \cup \mathit{just}(g)$, according to the partial order over cells; in order to do this, we shall introduce a notion of an *upper-bound value* of a set of cells.

6.2 How To Handle Backward Changes and User Inputs

Before we turn our attention to that, we need to discuss how to handle backward changes to the database and user inputs. Backward changes disrupt the intuition discussed in the previous section about the value of a cell group. In fact, a backward cell-group essentially states that one or more of its occurrences have values that are considered invalid, and should be replaced by a llun. Therefore, the values of these cells should not be considered when computing upper bounds. Similarly, user inputs trump any other value, and represent values of the highest priority.

To seamlessly introduce this notion into the semantics, we introduce a new category of cells, called *meta-cells*. More specifically, we assume a set of meta cells, *meta-cells*, with two elements:

- the *invalid cell*, $c_\Box$, with value $\Box$;

- the *user cell*, $c_\top$, with value $\top$.

First, we introduce the notion of a cell group *with user inputs* as a cell group g such that there exists a subset of $\mathcal{C}_{sub}$ of $occ(g) \cup just(g)$ such that $\mathsf{User}(\mathcal{C}_{sub})$ is defined and it is equal to a constant value v. Then, we associate a possibly empty set *meta-cell*(g) with each cell group g to properly encode its backward flag and user inputs, as follows:

1. if g has user inputs, then *meta-cell*(g) is the user cell $c_\top$; otherwise:
2. if *isBckw*(g) is true, then *meta-cell*(g) is the invalid cell $c_\square$;
3. if *isBckw*(g) is false, then *meta-cell*(g) is empty.

We extend the partial order of cells introduced in Definition 9 to incorporate meta cells in a straightforward way. More specifically:

- the invalid cell $c_\square$ is incomparable to any cell in *cells*$(J) \cup$ *new-cells*(J); it remains true that any authoritative cell c_a is such that $c_\square \preceq_\sqcap c_a$;
- the user cell $c_\top$ is the highest cell in the partial order of cells, i.e., for each cell c, $c \preceq_\sqcap c_\top$.

In essence, the invalid cell is incomparable wrt to non-authoritative cells. As a consequence, it forces the upper-bound value to a llun, unless there are authoritative cells that may fix it. The user cell always trumps any other cell.

It is quite straightforward to prove that this extension of Definition 9 is still a partial order.

6.3 Valid Cell Groups and Valid Updates

We notice that our semantics is centered around the fact that arbitrary updates to the target are forbidden. We therefore introduce the notion of a *valid cell group*. Intuitively, a valid cell group should have a value that "generalizes" and "improves" the ones of its occurrences and justifications. We formalize this notion as the *upper-bound value* of a cell group:

Definition 10 [Upper-Bound Value] Given a set of cells $\mathcal{C}$ in *cells*$(I) \cup$ *cells*$(J) \cup$ *new-cells*$(J) \cup$ *meta-cells*, we define the *upper-bound value* of $\mathcal{C}$, denoted by *lub-val*$(\mathcal{C})$, as follows:

1. if the user-cell, $c_\top \in \mathcal{C}$, then consider the set of cells $\mathcal{C}' = \mathcal{C} - \{c_\top\}$; then *lub-val*$(\mathcal{C}) = \mathsf{User}(\mathcal{C}')$, if this is defined and equal to a constant; if $\mathsf{User}(\mathcal{C}')$ is not defined, then *lub-val*$(\mathcal{C})$ is a fresh llun value L_i;

2. otherwise, if all cells in $\mathcal{C}$ have a null value, then *lub-val*$(\mathcal{C})$ is a fresh null value N_i;

3. otherwise, consider the set *maximal-cells*$(\mathcal{C})$ of maximal elements in $\mathcal{C}$ wrt $\preceq_\sqcap$. If all cells in *maximal-cells*$(\mathcal{C})$ have exactly the same value $v \in$ CONSTS, then *lub-val*$(\mathcal{C})$ is exactly v;

4. otherwise *lub-val*$(\mathcal{C})$ is a fresh llun value L_j.

Whenever *lub-val*$(\mathcal{C})$ comes from an authoritative cell we say that it is an *authoritative value*. If it comes from the user function, User, we say that it is a *user value*.

Notice that, throughout the definition of the semantics, we always refer to the original value of a cell in *cells*$(I) \cup$ *cells*$(J) \cup$ *new-cells*(J), regardless of any modification that will be done to the target database to satisfy the constraints. This is a very important feature of our approach: to find solutions we compute upper-bound values with respect to a partial order of cells, $\preceq_\sqcap$, that is fixed in advance and cannot change. This guarantees that the repair process does not interfere with the strategy to pick-up preferred values for cell groups, and therefore no termination or confluence problems may arise [CFY13] (further details on this aspect are provided in Section 10).

We can now formalize the notion of a *valid cell group* as a cell group whose value is either the upper-bound value of the corresponding cells, or a generalization thereof.

Definition 11 [VALID CELL GROUP] A cell group g is called a *valid cell group* if it is not refused by the user function, i.e., it is not the case that $\mathsf{User}($*cells*$(g)) = \bot$, and in addition:

- either *val*(g) is equal to the upper-bound value *lub-val*(g) = *lub-val*$($*occ*$(g) \cup$ *just*$(g) \cup$ *meta-cell*$(g))$ (in this case, we say that g is *strict*);
- or *val*(g) is more informative than *lub-val*(g) (and we say that g is *non strict*).

A *valid* update is an update made exclusively of valid cell groups. From now on, we shall only consider valid updates to the database.

Example 6: Following is a set of valid cell groups from our motivating example. Cell groups $g_1 - g_4$ are valid and strict.

$$g_1 : \langle L_0 \rightarrow \{t_5.\mathsf{CC\#}_{[781658]}, t_6.\mathsf{CC\#}_{[784659]}\}, by\ \emptyset\rangle$$

The value of g_1 is a llun, because g_1 has two occurrences with different values, and we have no preference rules that states than one is an improvement over the other.

$$g_2 : \langle Med \rightarrow \{t_{12}.\mathsf{Insur}^{new}\}, by\ \{t_2.\mathsf{Insur}_{[Med]}\}\rangle$$

In g_2 we have two cells. One is a new cell that is to be inserted into the target. Therefore, the source justification provides the value for g_2.

$$g_3 : \langle null \rightarrow \{t_{12}.\mathsf{Salary}^{new}\}, by\ \emptyset\rangle$$

Cell group g_3 is an example of a cell group that only has null cells. Therefore its upper-bound value is a null value.

$$g_4 : \langle 25K \rightarrow \{t_7.\mathsf{Salary}_{[10K]}, t_8.\mathsf{Salary}_{[25K]}\}, by\ \emptyset\rangle$$

Finally, g_4 has two occurrences that are ordered according to the partial order specification (one salary is more recent than the other). The upper-bound value is chosen accordingly.

All cell groups so far have a value that is strict, i.e., it coincides with the upper-bound value of the involved cells. However, we also consider as valid cell groups that overgeneralize these cells, for example:

$$g_4' : \langle L_1 \rightarrow \{t_7.\mathsf{Salary}_{[10K]}, t_8.\mathsf{Salary}_{[25K]}\}, by\ \emptyset\rangle$$

By comparing g_4 and g_4' it is easy to see that g_4' is not a "minimal" way of upgrading the database, since it is introducing an unnecessary llun into the target.

Following are additional cell groups with authoritative cells and backward changes.

$$g_5 : \langle Dental \rightarrow \{t_8.\mathsf{Treat}_{[Cholest]}\}, by\ \{t_{e_4}.\mathsf{Treat}^{auth}_{[Dental]}\}\rangle$$

The cells in g_5 have a clear upper bound, namely the authoritative source cell from t_{e_4}. Therefore, the upper-bound value coincides with the value of $t_{e_4}.\mathsf{Treat}$, and the cell group is strict.

All cell groups so far had the backward flag set to false. As a consequence, their meta-cell coincided with the bottom cell, which has no effect on the upper-bound value. On the contrary, consider:

$$g_6 : \langle L_3 \rightarrow \{t_7.\mathsf{Insurance}_{[Abx]}\}, by\ \emptyset, \mathsf{bckw}\rangle$$

Cell group g_6 is different from the cell groups above since it is a backward change. Its meta-cell is the invalid cell, $c_\square$. As a consequence, its strict value is a llun, since there is no maximal cell in $\mathit{occ}(g_6) \cup \mathit{just}(g_6) \cup \mathit{meta\text{-}cell}(g_6)$.

Example 7: Consider table $R(A, B)$ with three dependencies:

- an FD $A \to B$, encoded by $e_0 : R(x, y), R(x, y') \to y = y$;
- a CFD $A[a] \to B[v_1]$, stating that whenever $R.A$ is equal to "a", $R.B$ should be equal to "v_1"; this is encoded by egd $e_1 : R(x, y), c1(x, z) \to y = z$;
- a second CFD $A[a] \to B[v_2]$, that contradicts the first one and states that whenever $R.A$ is equal to "a", $R.B$ should be equal to "v_2"; this is encoded by $e_2 : R(x, y), c2(x, z) \to y = z$.

Assume R contains two tuples: $t_1 : R(a, 1), t_2 : R(a, 2)$. Here, $c1, c2$ are authoritative source tables with a single tuple t_{c1}, t_{c2} each, encoding the patterns in the CFDs: $t_{c1} : (A : a, B : v_1), t_{c_2} : (A : a, B : v_2)$. Since the two CFDs clearly contradict each other, previous approaches [FG12] would fail to give a solution to this example. Nevertheless, later on we will provide a semantics for this case.

Following is a set of cell groups, with an indication of their validity. Here we assume that the partial order specification Π states that cells of A with higher values are to be preferred over ones with smaller ones, and $\mathsf{User}(\mathit{cells}(g_4)) = k$.

$$
\begin{array}{cc}
g_1 = \langle 1 \to \{t_1.\mathsf{B}_{[1]}\},\ by\ \emptyset\rangle & \mathit{valid} \\
g_1' = \langle L_1 \to \{t_1.\mathsf{B}_{[1]}\},\ by\ \emptyset\rangle & \mathit{valid} \\
g_2 = \langle 2 \to \{t_1.\mathsf{B}_{[1]}, t_2.\mathsf{B}_{[2]}\},\ by\ \emptyset\rangle & \mathit{valid}\ (\Pi) \\
g_2' = \langle 3 \to \{t_1.\mathsf{B}_{[1]}, t_2.\mathsf{B}_{[2]}\},\ by\ \emptyset\rangle & \mathit{non\ valid} \\
g_3 = \langle v_1 \to \{t_1.\mathsf{B}_{[1]}, t_2.\mathsf{B}_{[2]}\},\ by\ \{t_{c_1}.\mathsf{B}^{auth}_{[v_1]}\}\rangle & \mathit{valid} \\
g_4 = \langle k \to \{t_1.\mathsf{B}_{[1]}, t_2.\mathsf{B}_{[2]}\},\ by\ \{t_{c_1}.\mathsf{B}^{auth}_{[v_1]}, t_{c_2}.\mathsf{B}^{auth}_{[v_2]}\}\rangle & \mathit{valid}(\mathsf{User}) \\
g_5 = \langle L \to \{t_1.\mathsf{B}_{[1]}, t_2.\mathsf{B}_{[2]}\},\ by\ \{t_{c_1}.\mathsf{B}^{auth}_{[v_1]}, t_{c_2}.\mathsf{B}^{auth}_{[v_2]}\}\rangle & \mathit{valid}
\end{array}
$$

In the case of g_4, we look for the maximal cell in $\mathit{occ}(g_4) \cup \mathit{just}(g_4) \cup \mathit{meta\text{-}cell}(g_4)$. Since the user function is defined, the maximal cell consists of the user cell $c_\top$, and the strict value is exactly $\mathsf{User}(\mathit{occ}(g_4) \cup \mathit{just}(g_4))$.

Notice that g_2' is not valid, since its value is neither the upper-bound value of the cells (constant 3 does not appear anywhere in its occurrences), nor a generalization of them. As such, in our approach it represents an unjustified way of repairing the database and is discarded.

With this in mind, we shall now lift this partial order over cells to a partial order $\preceq_{\Pi,\mathsf{User}}$ over cell groups, that ultimately will tell us when an update upgrades the database more than another.

6.4 The Partial Order of Cell Groups: A Simplified Case

Since the definition of the partial order over cell groups has a number of subtleties, let us first discuss a simple case in which we ignore some of the features of our approach, namely: authoritative tables (i.e., $\mathcal{S}_a$ is empty), and user inputs (i.e., User is also empty). Despite these simplifications, the fragment of the semantics we consider here represents an interesting combination of tgds, egds with hard-conflict resolution, forward and backward changes, and declarative preference rules specified through the partial-order specification.

This simplified case allows us to introduce the main intuitions behind the notion of upgrades, without giving all of the technical details that are needed to handle our framework in its generality. We will extend the definitions to the general case in the next paragraph.

Solutions to mapping & cleaning scenarios (formalized in Section 7) are updates made of valid cell groups that represent "upgrades" to the original target instance. However, we are also interested in *minimal solutions*. Intuitively, these will be composed only of strict cell groups that minimally upgrade the database and do not introduce over-generalizations of the original values. We now lift the partial order over cells, $\preceq_{\Pi}$, to a partial order over valid cell groups and updates. This is based on the following intuitions:

- it is natural to say that a cell group g' is an improvement wrt a cell group g if its cells "carry more information" according to the partial order of cells and values. Given the definition of a strict value, this happens when the set of occurrences and justifications of g' contains those of g. Therefore we state that g' can only be preferred over g if this *containment property* among their cells is satisfied. When the containment property is not satisfied, these cell groups represent incomparable ways to modify a target instance, and therefore cannot be ordered;

- in addition, given two cell groups g, g' with comparable occurrences and justifications, we need that the value set by g' is "at least as good" as the value set by g.

More formally:

Definition 12 [Simplified Partial Order for Cell Groups] Given two valid cell groups g and g', we say that $g \preceq_{\sqcap} g'$ iff either g is equal to g', or:

1. the cell-containment property is satisfied: $occ(g) \subseteq occ(g')$, $just(g) \subseteq just(g')$, and $isBckw(g')$ is true if $isBckw(g)$ is true;
2. in addition, one of the following is true:
   a) $val(g) \in$ NULLS (in this case, $val(g')$ may be either a null, or a constant, or a llun);
   b) $val(g') \in$ LLUNS (in this case, $val(g)$ may be either a null, or a constant, or a llun);
   c) $val(g) = val(g') \in$ CONSTS – i.e., both are equal constants – and g, g' are non-strict;
   d) $val(g) \in$ CONSTS, $val(g') \in$ CONSTS and both cell groups are strict, i.e., $val(g) = lub\text{-}val(occ(g) \cup just(g))$ and $val(g') = lub\text{-}val(occ(g') \cup just(g'))$.

We notice that there are alternative and more compact formulations for this notion. However, we prefer to state the definition in this form because we believe it makes it more readable, and allows us to clarify its content using an analogy to data exchange. In data exchange, solutions are compared to one another via homomorphisms in which: (i) nulls can be mapped to one another; (ii) nulls can be mapped to constants, but not the other way round; (iii) each constant can only be mapped to itself. In our setting, updates are compared via the partial order, with a similar inspiration:

1. items (a)-(b) state conditions analogous to $(i), (ii)$ in our setting, where three different kind of values are present;
2. items (c)-(d) handle constants, which require special attention; item (c) is analogous to (iii); we say that two non-strict cell groups, g and g', can be compared if they satisfy the containment property, and change the cells to the same constant (notice that g' may change a larger number of cells wrt g);
3. items (d) is peculiar to our framework, and incorporates the preference relation over cells we have introduced in Definitions 9, 10; in essence, it states that the constant values of g and g' may also be different from one another, provided that they are obtained as upper-bound values of the cells in the two cell groups; due to the containment property, this

guarantees that the value of g' is "at least as good" as (possibly an improvement over) the one of g.

Also in this case, $\preceq_\Pi$ is a partial order over valid cell groups:

Proposition 3 *Relation $\preceq_\Pi$ among valid cell-groups over $\langle I, J\rangle$ as specified in Definition 12 is a partial order.*

This partial order over cell groups can be easily lifted into a partial order over updates, and provides the basis to formalize the crucial notion of an *upgrade* in Section 7.1.

6.5 The Partial Order of Cell Groups: The General Case

When we move to the general case, things become more involved. In fact, we now have a much more ambitious goal, that it, to unify into a single notion the many different facets of the process of improving cells in a database. We want that the partial order of cell groups still encodes the rules discussed in the previous section, i.e (i) constant values are an improvement over null values, and lluns are an improvement over constants, (ii) cells in the input databases are ordered according to the partial order specification, (iii) unjustified changes to the database are forbidden.

However, in addition to these we need to incorporate a number of new rules: (iv) values coming from authoritative tables are an improvement over other values; (v) ultimately, users may step in and specify values which are to be considered higher up than anything else in our partial order.

To provide the general definition, given a mapping & cleaning scenario $\mathcal{M}$ and an instance $\langle I, J\rangle$, we partition valid cell groups over $\langle I, J\rangle$ in the following subsets:

- $user_{\mathcal{M},\langle I,J\rangle}$, the set of cell groups with user inputs, i.e., they are such that the user function, User, is defined over a subset of their cells; this is further divided in $user\text{-}strict_{\mathcal{M},\langle I,J\rangle}$, $user\text{-}nonstr_{\mathcal{M},\langle I,J\rangle}$, depending if a cell group takes the strict value or a generalization;
- $auth_{\mathcal{M},\langle I,J\rangle}$, the set of cell groups with no user inputs, whose justifications contain authoritative cells; this is further divided in $auth\text{-}strict_{\mathcal{M},\langle I,J\rangle}$, $auth\text{-}nonstr_{\mathcal{M},\langle I,J\rangle}$, similarly to the previous case;
- $std_{\mathcal{M},\langle I,J\rangle}$, the set of cell groups with no user inputs, nor authoritative cells.

We assign an order to these sets, such that:

$$std_{\mathcal{M},\langle I,J\rangle} < auth\text{-}strict_{\mathcal{M},\langle I,J\rangle} < auth\text{-}nonstr_{\mathcal{M},\langle I,J\rangle} < user\text{-}strict_{\mathcal{M},\langle I,J\rangle} < user\text{-}nonstr_{\mathcal{M},\langle I,J\rangle}$$

The general partial order of cell groups is an extension of the one introduced in Definition 12, as follows:

Definition 13 [Partial Order over Cell Groups] Given two valid cell groups g and g', we say that $g \preceq_{\Pi,\mathsf{User}} g'$ iff either g is equal to g', or:

1. the cell-containment property is satisfied: $occ(g) \subseteq occ(g')$, $just(g) \subseteq just(g')$, and $isBckw(g')$ is true if $isBckw(g)$ is true;
2. in addition, one of the following is true:
   a) g belongs to a subset that is less than the subset of g' in the order above;
   b) $g, g' \in user_{\mathcal{M},\langle I,J\rangle} \cup auth_{\mathcal{M},\langle I,J\rangle}$, and they belong to the same subset;
   c) $g, g' \in std_{\mathcal{M},\langle I,J\rangle}$ and the conditions of Definition 12 hold, i.e.:
      i. $val(g) \in$ NULLS;
      ii. $val(g') \in$ LLUNS;
      iii. $val(g) = val(g') \in$ CONSTS and g, g' are non-strict;
      iv. $val(g) \in$ CONSTS, $val(g') \in$ CONSTS and both cell groups are strict.

Consider Example 7. Cell groups are ordered as follows:

$$g_1 \preceq_{\Pi,\mathsf{User}} g_2 \preceq_{\Pi,\mathsf{User}} g_3 \preceq_{\Pi,\mathsf{User}} g_4 \preceq_{\Pi,\mathsf{User}} g_5$$

First, $g_1 \preceq_{\Pi,\mathsf{User}} g_2$ because we have a conflict wrt the FD $A \to B$, and we know that higher values are preferable for cells $t_1.B, t_2.B$ in R. However, we have a CFD that requires that both cells should be equal to v_1, and therefore g_3 is an improvement over g_2. Unfortunately, in this example we also have a contradicting CFD, that states that the value should be v_2 instead. Our semantics accommodates for these inconsistencies by resorting to lluns, as stated by g_4. Lluns, however, need at some point to be resolved by asking for user input; in our example, if the user provides the value k for the llun, we have a final group g_5 that is an improvement over all the others. It is also true that $g_1 \preceq_{\Pi,\mathsf{User}} g_1'$, since L_1 is more informative than 1.

Proposition 4 *Relation $\preceq_{\Pi,\mathsf{User}}$ among valid cell-groups is a partial order.*

What are Lluns, in the End? The role and the importance of lluns should now be apparent. While lluns are nothing more than symbols from a distinguished set, like constants and nulls, their use in conjunction with cell groups makes them a powerful addition to the semantics. Not only they allow us to complete the lattice of cell-groups and updates, but, when appearing inside cell-groups, they also provide important lineage information to support users in the delicate task of resolving conflicts. Consider again Example 7. The cell group $\langle L \rightarrow \{t_1.\mathsf{B}_{[1]}, t_2.\mathsf{B}_{[2]}\}, by\ \{t_{c_1}.\mathsf{B}^{auth}_{[v_1]}, t_{c_2}.\mathsf{B}^{auth}_{[v_2]}\}\rangle$ is a clear indication that it was not possible to fully resolve the conflicts, and therefore user interventions are needed to complete the repair. In addition, the cell-group provides complete information about the conflict, both in terms of which target cells – and therefore which original values – were involved, and also in terms of source values that justify the change.

Chapter 7

Semantics

7.1 Upgrades

We are now ready to formalize the notion of an upgrade. To do this, we must be able to compare updates with each other. We therefore introduce one final tool, called *id mappings*. This is a way to map updates – which by tgds may add new tuples with completely new ids to the target – to one another.

Definition 14 [Id Mapping] Let Upd and Upd' be two updates over $\langle I, J\rangle$. An *id mapping* h_{id} from Upd to Upd' maps tuple ids appearing in Upd, denoted by $\mathsf{tids}(\mathsf{Upd})$, into those appearing in Upd', $\mathsf{tids}(\mathsf{Upd}')$.

Id mappings can be extended to cells in a straightforward way. Given a cell group $g = \langle v \rightarrow \mathit{occ}(g)\ \mathit{by}\ \mathit{just}(g), \mathit{isBckw}\rangle$ we denote its image according to h_{id} by $h_{id}(g) = \langle v \rightarrow h_{id}(\mathit{occ}(g))\ \mathit{by}\ h_{id}(\mathit{just}(g)), \mathit{isBckw}\rangle$, if $h_{id}(g)$ is a valid cell group. Otherwise we say that its image is undefined. With this in mind, we introduce the concept of an upgrade:

Definition 15 [Upgrade] Given two updates Upd and Upd' over $\langle I, J\rangle$ and a partial order $\preceq_{\Pi,\mathsf{User}}$ on cell groups, we say that Upd' *upgrades* Upd, denoted by $\mathsf{Upd} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}'$, if there exists an id mapping h_{id} from the tuple ids in Upd to the tuple ids in Upd' such that for each cell group $g \in \mathsf{Upd}$ there exists a cell group $g' \in \mathsf{Upd}'$ and $h_{id}(g) \preceq_{\Pi,\mathsf{User}} g'$.

In the rest of the thesis, we consider that an update Upd' is preferable to Upd whenever $\mathsf{Upd} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}'$. Notice that $\preceq_{\Pi,\mathsf{User}}$ is a preorder for updates, and not a partial order. It is not, in fact, antisymmetric: there may exist updates Upd and Upd' such that $\mathsf{Upd} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}'$, $\mathsf{Upd}' \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$, and $\mathsf{Upd} \neq \mathsf{Upd}'$

(examples are in the next section). It is possible to show that, in the case of cleaning scenarios where no tgds are present, $\preceq_{\Pi,\mathsf{User}}$ is also a partial order.

7.2 Solutions

In this section, we formalize the semantics of a mapping and cleaning scenario. The key challenge here is to develop a new semantics for mappings and cleaning constraints that is a conservative extension of the semantics for mappings in [FKMP05], and of those of data repairing in [FG12]. We address this challenge by leveraging the notions of cell groups, partial order and upgrades.

Intuitively, a solution of a mapping and cleaning scenario is a set of repair instructions of the target (represented by cell groups) that upgrades the initial dirty target instance and that satisfies the dependencies in Σ_t and Σ_e. We notice however that updating the database to enforce egds may disrupt the standard notion satisfaction of the tgds. To handle this, we shall adopt a revised notion satisfaction for dependencies, called of *satisfaction after upgrades*. We first give the formal definition of a solution.

Definition 16 [SOLUTION] Given a *mapping&cleaning* scenario $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$, and an input instance $\langle I, J\rangle$, a *solution* for $\mathcal{MC}$ over $\langle I, J\rangle$ is a valid update Upd s.t.:

1. $J \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$, i.e., Upd upgrades the initial target instance;
2. $\langle I, \mathsf{Upd}(J)\rangle$ *satisfies after upgrades* $\Sigma_t \cup \Sigma_e$ under $\preceq_{\Pi,\mathsf{User}}$.

Let us now turn to the definition of satisfaction after upgrades. Let Upd be an update over $\langle I, J\rangle$. Clearly, if $\langle I, \mathsf{Upd}(J)\rangle$ satisfies an edg or tgd in the standard semantics, nothing needs to be done. Otherwise, we revise the semantics for edgs and tgds in such a way that they remain satisfied as long as we upgrade the database.

7.3 Satisfaction After Upgrades for Egds

Let us first discuss egds. Consider Example 7. Notice that we upgrade the database with cell group g_5 to write a user input, k, into cells $t_1.B, t_2.B$, and obtain two identical tuples $t_1 : R(a, k), t_2 : R(a, k)$. However, the two (contradicting) conditional functional dependencies in this example state that, whenever $R.A$ equals a, $R.B$ must be equal to v_1, v_2, respectively. Therefore, after g_5, the corresponding egds are not satisfied in the standard sense.

We still want to consider these updates as solutions, since they are the result of an "improvement" of values that originally satisfied the dependencies, but were dirty. In order to do this, given a dependency (egd or tgd), variables x, x', and a homomorphism h of the premise into $\langle I, \mathsf{Upd}(J)\rangle$, we need to be able to compare the cell groups associated by h with x, x', to check whether one value, say $h(x)$, is an upgrade for $h(x')$, or vice versa.

Notice that a variable x may have several occurrences in a formula. Homomorphism h maps each occurrence into a cell of the database. We denote by $\mathit{cells}_h(x)$ the set of cells in $\langle I, \mathsf{Upd}(J)\rangle$ associated by h with occurrences of x. Then, we define the notion of a cell group associated by h with x, $g_h(x)$, as the result of merging all cell groups of cells in $\mathit{cells}_h(x)$.

Definition 17 [Cell Group for Variable] Given a dependency $d : \phi(\bar{x}) \rightarrow \exists \bar{y} : \psi(\bar{x}, \bar{y})$, and a homomorphism h of $\phi(\bar{x})$ into $\langle I, \mathsf{Upd}(J)\rangle$, for each variable $x \in \bar{x}$ we define the *cell group associated by h with x* as the cell group $g_h(x) = \langle h(x), \mathit{occ}, \mathit{just}, \mathit{isBckw}\rangle$, where:

- *occ* (resp. *just*) is the union of all occurrences (resp. justifications) of the cell groups in Upd for cells in $\mathit{cells}_h(x)$;
- in addition, *just* contains all cells in $\mathit{cells}_h(x)$ that belong to the source tables in I;
- in addition, *isBckw* is true if it is true in any of the cell groups for cells in $\mathit{cells}_h(x)$.

Consider the sample scenario in example 2. Given the initial instance, the s-t tgd $m : S(x, y) \rightarrow \exists z : T(x, y, z)$, and the homomorphism h that maps x into constant 1 and y into 2, the cell groups for these variables are $g_h(x) : \langle \mathit{1} \rightarrow \emptyset, \mathit{by}\ \{t_1.\mathsf{A}_{[1]}\}\rangle$ and $g_h(y) : \langle \mathit{2} \rightarrow \emptyset, \mathit{by}\ \{t_2.\mathsf{B}_{[1]}\}\rangle$. Consider now the target instance ΔJ and the egd $e : T(x, y, z), T(x, y', z') \rightarrow y = y'$. In this case, consider homomorphism h that maps, among others, variable x into constant 1; the cell group associated by h with x is $g_h(x) : \langle \mathit{1} \rightarrow \{t_3.\mathsf{A}^{new}, t_4.\mathsf{A}^{new}\}, \mathit{by}\ \{t_1.\mathsf{A}_{[1]}, t_2.\mathsf{A}_{[1]}\}\rangle$.

Based on the notion of cell group for a variable, we are now ready to introduce the notion of *satisfaction after upgrades* for egds:

Definition 18 [Satisfaction After Upgrades - Egds] Given an egd $e : \forall \overline{x}\ \phi(\overline{x}) \rightarrow x = x'$, an instance $\langle I, J\rangle$, and an update Upd, we say that $\langle I, \mathsf{Upd}(J)\rangle$ *satisfies after upgrades* e wrt the partial order $\preceq_{\Pi,\mathsf{User}}$ if, whenever there is a homomorphism h of $\phi(\overline{x})$ into $\langle I, \mathsf{Upd}(J)\rangle$, then (i) either the value

of $h(x)$ and $h(x')$ are equal (standard semantics), or (ii) it is the case that the cell groups associated by h with x, x' are comparable wrt $\preceq_{\Pi,\mathsf{User}}$, i.e., either $g_h(x) \preceq_{\Pi,\mathsf{User}} g_h(x')$ or $g_h(x') \preceq_{\Pi,\mathsf{User}} g_h(x)$.

7.4 Satisfaction After Upgrades for Tgds

A similar notion holds for tgds. To give an example, consider our motivating Example 1. The s-t tgd m_1 uses source tuples t_1, t_2 from source #1 to generate tuple t_{10} : *(123, W. Smith, 324-0000, Pico Blvd., LA, null)* into the target; we call t_{10} the *canonical update* for m_1 and t_1, t_2. After egds have been enforced, however, the tuple is upgraded in several ways, and becomes *(L_1, W. Smith, 324-3456, Pico Blvd., LA, null)*. Again, after the changes the target instance does not satisfy tgd m_1 in the standard sense.

Consider a tgd $m : \forall\overline{x}(\phi(\overline{x}) \rightarrow \exists\overline{y}\ (\psi(\overline{x}, \overline{y})))$ that is not satisfied by $\langle I, \mathsf{Upd}(J)\rangle$. Let h be a homomorphism of $\phi(\overline{x}, \overline{z})$ into $\langle I, \mathsf{Upd}(J)\rangle$ that cannot be extended to a homomorphism h' of $\psi(\overline{x}, \overline{y})$ into $\langle I, \mathsf{Upd}(J)\rangle$. We now want to regard m as being *satisfied after upgrades* whenever $\mathsf{Upd}(J)$ is an upgrade of the *canonical update* for m and h.

Intuitively, the canonical update Upd_h^{can} represents the "standard way" to satisfy the tgds, defined as follows:

Definition 19 [Canonical Update] Let h_{can} be the *canonical homomorphism* that extends h by injectively assigning a fresh labeled null with each existential variable. Consider the new instance $J_{can} = J \cup h_{can}(\psi(\overline{x}, \overline{y}))$, obtained by adding to J the set of tuples in $\Delta J = h_{can}(\psi(\overline{x}, \overline{y}))$, each with a fresh tuple id. Then, Upd_h^{can} is such that:

1. $J_{can} = \mathsf{Upd}_h^{can}(J)$;
2. Upd_h^{can} coincides with Upd when restricted to *cells*(J);
3. it contains a cell group $g_{h_{can}}(x)$ over $\langle I, J\rangle$ for each variable $x \in \bar{x}$, where $g_{h_{can}}(x)$ denotes the cell group associated by h_{can} with variable x;
4. it contains a cell group $g_{h_{can}}(y)$ over $\langle I, J\rangle$ for each variable $y \in \bar{y}$, with value $h_{can}(y)$, occurrences equal to $cells_{h_{can}}(y)$, all of them new cells, empty justifications and no backward flag.

Consider again example 2, the s-t tgd m and the homomorphism h defined above. The canonical update Upd_h^{can} contains the following cell groups: g_1 :

$\langle 1 \rightarrow \{t_3.\mathsf{A}^{new}\}, by\ \{t_1.\mathsf{A}_{[1]}\}\rangle$, $g_2 : \langle 2 \rightarrow \{t_3.\mathsf{B}^{new}\}, by\ \{t_1.\mathsf{B2}\}\rangle$ and $g_3 : \langle N_1 \rightarrow \{t_3.\mathsf{C}^{new}\}, by\ \emptyset\}\rangle$.

Definition 20 [Satisfaction After Upgrades - Tgds] Given a tgd $m : \forall\overline{x}(\phi(\overline{x}) \rightarrow \exists\overline{y}\ (\psi(\overline{x},\overline{y})))$, and an update Upd, we say that $\langle I, \mathsf{Upd}(J)\rangle$ *satisfies after upgrades* m under partial order $\preceq_{\Pi,\mathsf{User}}$ if, whenever there is a homomorphism h of $\phi(\overline{x})$ into $\langle I, \mathsf{Upd}(J)\rangle$, then (i) either m is satisfied by $\langle I, \mathsf{Upd}(J)\rangle$ in the standard sense, or (ii) $\mathsf{Upd}_h^{can} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$.

This concludes the formalization of the notion of a solution as given in Definition 16.

7.5 Minimal Solutions

We are interested in solutions that are *minimal*, i.e., they do not contain unneeded target tuples and upgrade the initial target instance as little as possible. To quantify minimality we leverage $\preceq_{\Pi,\mathsf{User}}$ to decide when one update Upd' *strictly upgrades* another update Upd, denoted by $\mathsf{Upd} \prec_{\Pi,\mathsf{User}} \mathsf{Upd}'$. More specifically, $\mathsf{Upd} \prec_{\Pi,\mathsf{User}} \mathsf{Upd}'$ if:

- $\mathsf{Upd} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}'$, but not the other way around; or
- $\mathsf{Upd} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}'$, according to id mapping h_{id}, $\mathsf{Upd}' \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$, according to id mapping h'_{id}, and h'_{id} is surjective while h_{id} is not surjective.

Definition 21 [Minimal Solutions] A *minimal solution* for a mapping and cleaning scenario is a solution Upd such that there exists no solution Upd' such that $\mathsf{Upd}' \prec_{\Pi,\mathsf{User}} \mathsf{Upd}$.

Consider Example 2. Figure 7.1 reports some of the updates and solutions for this example, along with a diagram of the "upgrades" relationship among them. This example clearly shows that $\preceq_{\Pi,\mathsf{User}}$ is a preorder for updates, and not a partial order (see, for example, $\mathsf{Upd}_3, \mathsf{Upd}_7$).

Two minimal solutions for our motivating example are the solutions #1 and #2 in Figure 1.2. These can be made non-minimal by adding unneeded tuples, or unnecessary changes.

7.6 Restrictions and Relationship to Other Semantics

We stated several times that our semantics generalizes the ones previously introduced for mapping scenarios [FKMP05] on one side, and data repairing

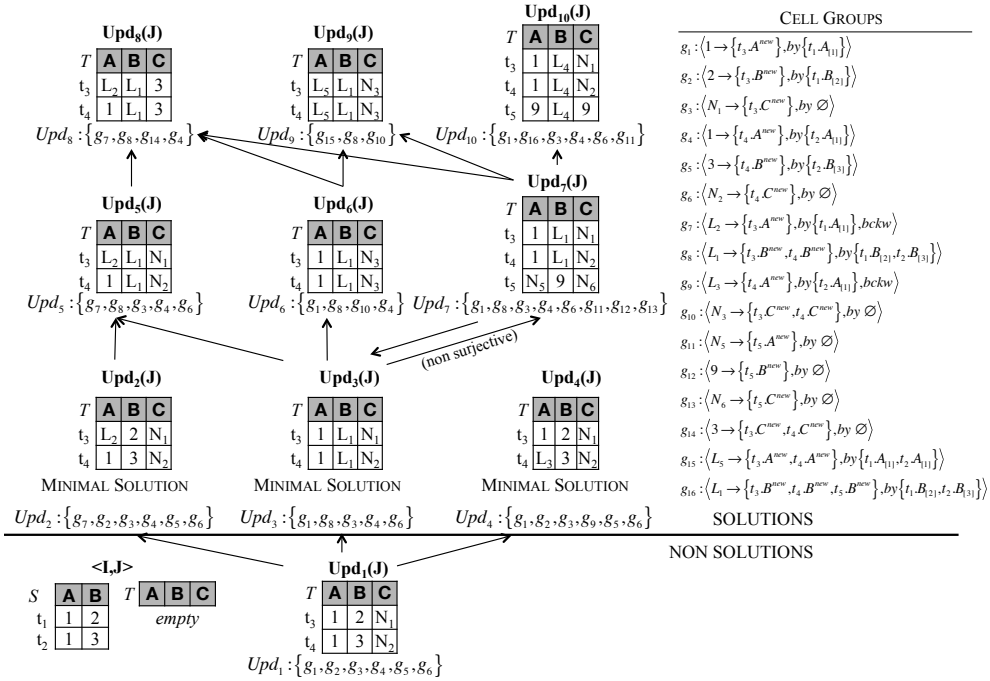


Figure 7.1: A diagram of solutions for Example 2.

[BFFR05, BFM07, BIG10] on the other side. Our goal in this section is to introduce two restrictions of our setting, the notion of a *mapping scenario*, and the notion of a *cleaning scenario*. We will investigate the relationship between mapping & cleaning scenarios and existing repair strategies later on, in Section 10, after we have introduced our operational chase-based semantics.

7.7 Mapping Scenarios and Data Exchange

To start, we compare our semantics to the one of data exchange. Recall that traditionally [FKMP05] a *data-exchange scenario* has been defined as a quadruple $\mathcal{M}_{de} = \{\mathcal{S}, \mathcal{T}, \Sigma_{st}^{de}, \Sigma_{t}^{de}\}$, where $\mathcal{S}$ and $\mathcal{T}$ are the source and target schemas, Σ_{st}^{de} a set of s-t tgds, and Σ_{t}^{de} a set of target constraints that includes target tgds and target egds. Notice how we use superscripts to emphasize that $\mathcal{M}_{de}$

only contains standard dependencies, as defined in Section 3.1.

We see mapping scenarios as a restriction of mapping and cleaning. Recall the general definition of a mapping and cleaning scenario as a tuple $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$. Recall that dependencies here are the extended tgds and cleaning egds defined in Section 3. Given a data-exchange scenario $\mathcal{M}_{de}$, we now introduce its *associated mapping scenario* $\mathcal{M}$ as a mapping & cleaning scenario over the same source and target schema, with the following restrictions:

- $\mathcal{S}_a$, the authoritative schema is empty;
- Σ_t is the set of standard s-t tgds in Σ_{st}^{de} and the set of standard target tgds in Σ_t^{de};
- Σ_e is the set of standard egds in Σ_t^{de};
- Π and User are empty;
- finally, the set of lluns, LLUNS, is also empty, i.e., we only allow for constants and labeled nulls in instances.

When restricted to this notion of a mapping scenario, many of the notions in our semantics are greatly simplified. First, a cell group becomes simply a set of occurrences and justifications. The partial order over the cells of $\langle I, J \rangle$ only states that constants are preferable to labeled nulls, and therefore a cell group g' is preferable to g another if the containment property among their cells is satisfied, and either both have the same value, or the value of g' is a constant, and the one of g is a null.

Interestingly, our semantics is a conservative extension of the one of data exchange, as stated by the following theorem.

Theorem 5 *Every (core) solution of a data exchange scenario corresponds to a (minimal) solution of its associated mapping scenario, and vice versa.*

We consider this an important result, since not only are we preserving the semantics of data exchange, but we are also extending it in a significant way. In fact, we are enriching it with a principled way to handle hard conflicts – i.e., conflicts among constant values – in addition to soft ones addressed in data exchange – the ones that involve nulls.

7.8 Cleaning Scenarios

Let us turn our attention to scenarios with no tgds. We call a *cleaning scenario* a mapping and cleaning scenario in which the set of tgds Σ_t is empty. We discuss them in detail in Section 10. For now, let us summarize some of the properties of cleaning scenarios. First, we notice that, even in the absence of tgds, cleaning egds may still maintain a link between the source and the target database, e.g., to leverage authoritative sources.

Theorem 6 *Given a cleaning scenario* $\mathcal{CS} = \{\mathcal{S}, \mathcal{T}, \Sigma_e, \Pi\}$ *and an input instance* $\langle I, J\rangle$*, there always exists a solution for* $\mathcal{CS}$ *and* $\langle I, J\rangle$*.*

Theorem 7 *Given two solutions* $\mathsf{Upd}, \mathsf{Upd}'$ *for a scenario* $\mathcal{CS}$ *over instance* $\langle I, J\rangle$*, one can check* $\mathsf{Upd} \preceq_\Pi \mathsf{Upd}'$ *in* $O(n + kmlog(m))$ *time, where* n *is the number of cells in* J*,* k *is the maximum number of cell groups in* Upd*,* Upd'*, and* m *is the maximum size of a cell group in* Upd*,* Upd'*.*

Chapter 8

The Chase

In order to generate solutions for mapping & cleaning scenarios, we resort to a variant of the traditional chase procedure for egds and tgds [FKMP05]. However, we revise and extend the standard chase substantially in order to achieve the following goals:

(*i*) our first goal is that the chase is correct, i.e., it computes solutions for our semantics; a key property, in this respect, is that chase steps preserve cell groups and generate actual upgrades of the target database; as a consequence, in the definition we shall make extensive use of cell groups and the partial order;

(*ii*) the second important property we want is that the chase explores all possible strategies to satisfy the dependencies, this implies that an egd may be chased both *forward*, to satisfy its conclusion, or *backward*, to falsify its premise; this, in turn, means that we need to consider a *disjunctive chase*, which generates a tree of alternative solutions;

(*iii*) finally, we have a strong concern for scalability: we want that the chase provides a basis to implement a scalable engine that scales nicely to large databases. This requirement has a tricky interaction with the expressibility one above, since we need to find the appropriate balance between generality and scalability.

To simplify the presentation and ease the reading, we shall proceed in two steps. In this section, we introduce a first, simplified version of the chase that we show to be correct. Then, we revise the definition in the next section to show how to strike a balance with scalability.

To start, recall that we also want to incorporate user inputs in the process. As a consequence, our chase generates a tree of solutions by three main kind of steps: (a) chasing egds (forward and backward) with cell groups; (b) chasing tgds with cell groups; (c) correcting cell groups or refuting updates by means of user inputs. Notice that we don't backward-chase tgds, since in most cases this would require to delete tuples from the original database, a modification that is not allowed in our setting.

We fix a mapping & cleaning scenario $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \preceq_\Pi, \mathsf{User}\}$ and instances I of $\mathcal{S} \cup \mathcal{S}_a$ and J of $\mathcal{T}$. Given a (possibly empty) update Upd of J, a dependency d (tgd or egd) is said to be *applicable* to $\langle I, \mathsf{Upd}(J)\rangle$ with homomorphism h if h is a homomorphism of the premise of d into $\langle I, \mathsf{Upd}(J)\rangle$ that violates the conditions for $\langle I, \mathsf{Upd}(J)\rangle$ to satisfy after upgrades d. Recall that, given a homomorphism h of a formula $\phi(\bar{x})$ into $\langle I, \mathsf{Upd}(J)\rangle$, we denote by $g_h(x)$ the cell group associated by h with variable x.

8.1 Chase Step for Tgds

The definition of a chase step for an extended tgd is a natural generalization of the standard notion of a chase step for a standard tgd [FKMP05], the main difference being the need to properly update cell groups.

Definition 22 [Chase Step for TGDs] Given an extended tgd $m : \forall\overline{x}\ (\phi(\bar{x}) \rightarrow \exists\bar{y} : (\psi(\bar{x}, \bar{y})))$ in Σ_t applicable to $\langle I, \mathsf{Upd}(J)\rangle$ with homomorphism h, by *chasing* m on $\langle I, \mathsf{Upd}(J)\rangle$ with h we generate a new update Upd' obtained from Upd by:

(i) removing all cell groups $g_h(x)$ that are present in Upd, for all $x \in \bar{x}$;

(ii) adding the new cell groups in the canonical update Upd_h^{can}.

In symbols we write $\mathsf{Upd} \rightarrow_{m,h} \mathsf{Upd}'$, where $\mathsf{Upd}' = \mathsf{Upd}_h^{can}$.

In our running Example 1, the canonical update for tgd m_1 over $\langle I, J\rangle$ contains, for the table $\mathsf{Treatments}$, the following new cell groups:

$$
\begin{aligned}
&g_1 : \langle \mathit{123} \rightarrow \{t_{12}.\mathsf{SSN}^{new}\},\ \mathit{by}\ \{t_1.\mathsf{SSN}_{[\mathit{123}]}, t_2.\mathsf{SSN}_{[\mathit{123}]}\}\rangle\\
&g_2 : \langle \mathit{null} \rightarrow \{t_{12}.\mathsf{Salary}^{new}\},\ \mathit{by}\ \emptyset\rangle\\
&g_3 : \langle \mathit{Med} \rightarrow \{t_{12}.\mathsf{Insur}^{new}\},\ \mathit{by}\ \{t_2.\mathsf{Insur}_{[\mathit{Med}]}\}\rangle\\
&g_4 : \langle \mathit{Eye\ surg.} \rightarrow \{t_{12}.\mathsf{Treat}^{new}\},\ \mathit{by}\ \{t_2.\mathsf{Treat}_{[\mathit{Eye\ surg}]}\}\rangle\\
&g_5 : \langle \mathit{12/01/2013} \rightarrow \{t_{12}.\mathsf{Date}^{new}\},\ \mathit{by}\ \{t_2.\mathsf{Date}_{[\mathit{12/01/2013}]}\}\rangle,
\end{aligned}
$$

8.2 Chase Step for Egds

In order to introduce the notion of a chase step for egds we need a few preliminary definitions. We first introduce the notions of *witness* and *witness variable* for an egd. Intuitively, the witness variables are those variables upon which the satisfiability of the dependency premise depends; these are all variables that have more than one occurrence in the premise, i.e., they are involved in a join or in a selection.

Definition 23 [Witness] Let $e : \forall\overline{x}\ (\phi(\overline{x}) \rightarrow x = x')$ be an egd. A *witness variable* for e is a variable $x \in \bar{x}$ that has multiple occurrences in $\phi(\bar{x})$. For a homomorphism h of $\phi(\bar{x})$ into $\langle I, \mathsf{Upd}(J)\rangle$, we call a *witness*, w_h for e and h, the vector of values $h(\bar{x}_w)$ for the witness variables $\bar{x}_w$ of e.

Consider, for example, dependency e_9 in Example 1 (we omit some of the variables for the sake of conciseness): e_9. $\mathsf{Treat}(\mathbf{ssn}, \mathsf{s}, \ldots), \mathsf{Treat}(\mathbf{ssn}, \mathsf{s'}, \ldots) \rightarrow \mathsf{s} = \mathsf{s'}$. Assume that the target instance of $\mathsf{Treatments}$ contains tuples $t_7 = (ssn : 111, salary : 10K, \ldots)$, $t_8 = (ssn : 111, salary : 25K, \ldots)$. We have a homomorphism h that maps the first atom of e_9 into t_7, and the second one into t_8. In this case, the witness variable, i.e., the variable that imposes the constraint that the two tuples have the same SSN, is *ssn*, and its value is 111.

Definition 24 [Merge of a set of Cell Groups] Given a set of cell groups, G, we define $merge_{\preceq_{\Pi,\mathsf{User}}}(G)$ the cell group $\langle v, occ, just, isBckw\rangle$, where (i) *occ* (resp. *just*) is the union of all occurrences (resp. justifications) of the cell groups in G, (ii) *isBckw* is true if it is true in any of the cell groups in G, (iii) v is the strict value of $merge_{\preceq_{\Pi,\mathsf{User}}}(G)$.

Definition 25 [Chase Step for EGDs] Given a cleaning egd $e : \forall\overline{x}\ (\phi(\overline{x}) \rightarrow x = x')$ in Σ_e applicable to $\langle I, \mathsf{Upd}(J)\rangle$ with homomorphism h, by *forward chasing* e on $\langle I, \mathsf{Upd}(J)\rangle$ with h we generate a new update Upd_f obtained from Upd by:
(i) removing $g_h(x)$ and $g_h(x')$;
(ii) adding the new cell group $merge_{\preceq_{\Pi,\mathsf{User}}}(g_h(x) \cup g_h(x'))$.
In symbols $\mathsf{Upd}_f = \mathsf{Upd} - \{g_h(x), g_h(x')\} \cup g_f$.

By *backward chasing* e on $\langle I, \mathsf{Upd}(J)\rangle$ with h we try to falsify the premise in all possible ways. To do this, we generate a number of new updates as follows: for each witness variable $x_i \in \bar{x}_w$ of e, and each cell $c_j \in cells_h(x_i)$ that belongs to the image $h(R(\ldots))$ of a relational atom appearing in $\phi(\bar{x})$, consider the corresponding cell group according to Upd, $g_{ij} = g_{\mathsf{Upd}}(c_j)$, where by $g_{\mathsf{Upd}}(c)$ we denotate the cell-group of cell c according to Upd. If:

(*i*) $val(g_{ij}) \in$ CONSTS, i.e., the cell has a constant value, and

(*ii*) *auth-cells*$(g_{ij}) = \emptyset$, i.e., g_{ij} has no authoritative justifications,

then, we generate a new update Upd_{bij} obtained from Upd by changing g_{ij} to another cell group g'_{ij} that has same occurrences and justifications, backward flag set, and a new llun L_{ij} as a value. In symbols $\mathsf{Upd}_{b_{ij}} = \mathsf{Upd} - \{g_{ij}\} \cup \{g'_{ij}\}$

We simultaneously consider all these chase steps, in parallel, and write $\mathsf{Upd} \rightarrow_{e,h} \mathsf{Upd}_f, \mathsf{Upd}_{b0}, \mathsf{Upd}_{b1}, \ldots, \mathsf{Upd}_{bn}$, where Upd_f and Upd_{b0}, Upd_{b1}, $\ldots, \mathsf{Upd}_{bn}$ are the updates generated by the forward and backward chase step, respectively.

Observe that we do not backward-chase cells in two cases: (*i*) they contain nulls or lluns; in fact, nulls and lluns are essentially placeholders, and there is no need to replace a placeholder by another one, since this is does not represent an upgrade; (*ii*) they have a authoritative justification from the source; since we use the source to model high-reliability data, we consider it unacceptable to disrupt a value coming from the source in favor of a llun.

To see an example, tuples t_7 and t_8 in the pre-solution in Figure 1.2 violate egd e_9, stating that SSN implies Salary on the table Treatments. Here the witness variable is *ssn*, and the $cells_h(ssn)$ are $t_7.\mathsf{SSN}$ and $t_8.\mathsf{SSN}$, with cell groups $g_{\mathsf{Upd}}(t_7.\mathsf{SSN}) : \langle 111 \rightarrow \{t_7.\mathsf{SSN}_{[111]}\}, by\ \emptyset\}\rangle$ and $g_{\mathsf{Upd}}(t_8.\mathsf{SSN}) : \langle 111 \rightarrow \{t_8.\mathsf{SSN}_{[111]}\}, by\ \emptyset\}\rangle$. Moreover, the cell groups for variables x and x' are $g_h(x) : \langle 10K \rightarrow \{t_7.\mathsf{Salary}_{[10K]}\}, by\ \emptyset\}\rangle$ and $g_h(x') : \langle 25K \rightarrow \{t_8.\mathsf{Salary}_{[25K]}\}, by\ \emptyset\}\rangle$. Notice that the two cell groups are incomparable, and therefore we have a violation.

The chase procedure generates three different updates for the violation:

(a) Upd_f that in the place of $g_h(x)$ and $g_h(x')$ contains the new cell group $g_f : \langle 25K \rightarrow \{t_7.\mathsf{Salary}_{[10K]}, t_8.\mathsf{Salary}_{[25K]}\}, by\ \emptyset\}\rangle$ ($25K$ is more recent than $10K$ as a salary, and therefore it is preferred); as you can see, the least upper bound is constructed in such a way that it contains the union of occurrences and the union of justifications of the two conflicting groups;

(b) Upd_{b_1} that in the place of $g_{\mathsf{Upd}}(t_7.\mathsf{SSN})$ contains its backward cell group $g_{b_1} : \langle L_6 \rightarrow \{t_7.\mathsf{SSN}_{[111]}\}, by\ \emptyset, \mathsf{bckw}\}\rangle$;

(c) Upd_{b_2} that in the place of $g_{\mathsf{Upd}}(t_8.\mathsf{SSN})$ contains its backward cell group $g_{b_2} : \langle L_7 \rightarrow \{t_8.\mathsf{SSN}_{[111]}\}, by\ \emptyset, \mathsf{bckw}\}\rangle$;

8.3 Chase Step for User Inputs

In addition to these steps, in our approach we also want users to be able to correct updates and provide inputs. This may happen in two ways: either by changing the values of cell groups (typically either lluns or nulls), or by refusing an update because it is considered to be incorrect (for example because an egd has been backward chased when the right update according to the user is the forward one). We therefore introduce a notion of a chase step for user inputs.

Definition 26 [Chase Step for User Inputs] Given an update Upd of J, we say that User *applies* to a group $g \in \mathsf{Upd}$ if $\mathsf{User}(\mathit{cells}(g))$ is defined, and returns a value that is different from $\mathit{val}(g)$.

We say that User *refuses* Upd if $\mathsf{User}(\mathit{cells}(g)) = \bot$. If User refuses Upd, we mark Upd as *invalid*. Otherwise, we denote by $\mathsf{User}(\mathsf{Upd})$ the update obtained from Upd by changing any cell group $g \in \mathsf{Upd}$ such that User applies to g to a new cell group g_{User} obtained from g by changing $\mathit{val}(g)$ to $\mathsf{User}(\mathit{cells}(g))$.

Chasing $\langle I, \mathsf{Upd}(J)\rangle$ with User either marks $\langle I, \mathsf{Upd}(J)\rangle$ as invalid, or generates a new update in which cell groups have been changed according to User. In the latter case, we write $\mathsf{Upd} \rightarrow_U \mathsf{User}(\mathsf{Upd})$.

Consider again the Upd_1 in Example 5. If $\mathsf{User}(\{t_1.\mathsf{SSN}, t_2.\mathsf{SSN}, t_3.\mathsf{SSN}\}) = 123$, chasing $\langle I, \mathsf{Upd}_1(J)\rangle$ with User generates a new update that contains, in the place of g_3, the new cell group $g_{\mathsf{User}} : \langle \mathit{123} \rightarrow \{t_{10}.\mathsf{SSN}^{new}, t_{11}.\mathsf{SSN}^{new}, t_{12}.\mathsf{SSN}^{new}, \quad t_{13}.\mathsf{SSN}^{new}\}, \mathit{by}\ \{t_1.\mathsf{SSN}_{[\mathit{123}]}, t_2.\mathsf{SSN}_{[\mathit{123}]}, t_3.\mathsf{SSN}_{[\mathit{124}]}\}\rangle$.

8.4 Chase Tree

Given sets of tgds and egds Σ_t, Σ_e, a chase of $\langle I, J\rangle$ with Σ_t, Σ_e and User, denoted by $\mathsf{chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$ is a tree whose root is $\langle I, J\rangle$, i.e., the empty update, and for each valid node Upd, the children of Upd are the updates $\mathsf{Upd}_0, \mathsf{Upd}_1, \ldots, \mathsf{Upd}_n$ such that, for some $d \in \Sigma_t, \Sigma_e$ and some h, it is the case that $\mathsf{Upd} \rightarrow_{d,h} \mathsf{Upd}_0, \mathsf{Upd}_1, \ldots, \mathsf{Upd}_n$, or $\mathsf{Upd} \rightarrow_{\mathsf{User}} \mathsf{Upd}_0$. The leafs are valid updates Upd_ℓ such that there is no dependency applicable to $\langle I, \mathsf{Upd}_\ell(J)\rangle$ with some homomorphism h. Any leaf in the chase tree is called a *result* of the chase of $\langle I, J\rangle$ with Σ_t, Σ_e.

Note that, as usual, the chase procedure is sensitive to the order of application of the dependencies. In our chase tree, we consider all possible orders in parallel. Different orders of application of the dependencies may lead to different chase sequences and therefore to different results.

8.5 Correctness, Termination, and Complexity

We next show that the chase, if it terminates, always generates solutions of mapping & cleaning scenarios. We also show that the chase does not always terminate in general.

Theorem 8 *Given a mapping & cleaning* $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$*, instances* I *of* $\mathcal{S} \cup \mathcal{S}_a$ *and* J *of* $\mathcal{T}$*, and oracle* User*, the chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *may not terminate after a finite number of steps. If it terminates, it generates a finite set of results, each of which is a solution for* $\mathcal{M}$ *over* $\langle I, J\rangle$*. Even if the chase terminates, not every minimal solution is generated.*

We can prove that, as soon as the tgds are non recursive, then the chase terminates. This result is far from trivial, since, as we discussed, egds interact quite heavily with tgds by updating values in the database. We conjecture that this result can be extended to more sophisticated termination conditions for tgds [GST11].

Theorem 9 *Given a mapping & cleaning* $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$*, instances* I *of* $\mathcal{S} \cup \mathcal{S}_a$ *and* J *of* $\mathcal{T}$*, and oracle* User*, if* Σ_t *is a set of weakly-acyclic tgds, then the chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *terminates after a finite number of steps, and each leaf in the chase tree is a solution for* $\mathcal{MC}$*.*

We next show that for cleaning scenarios, the chase procedure always generates solutions, i.e., it is sound, and it terminates after a finite number of steps.

Theorem 10 *Given a cleaning scenario* $\mathcal{CS} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \emptyset, \Sigma_e, \Pi, \mathsf{User}\}$ *and an instance* $\langle I, J\rangle$*, the chase of* $\langle I, J\rangle$ *with* Σ_e *(i) terminates; (ii) it generates a finite set of results, each of which is a solution for* $\mathcal{CS}$ *over* $\langle I, J\rangle$*.*

It is well-known [Ber11] that a database can have an exponential number of solutions, even for a cleaning scenario with a single FD and when no backward chase steps are allowed.

Theorem 11 *Given a cleaning scenario* $\mathcal{CS} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \emptyset, \Sigma_e, \Pi, \mathsf{User}\}$ *and an instance* $\langle I, J\rangle$*,* $\mathcal{CS}$ *may have at most an exponential number of solutions over* $\langle I, J\rangle$*, and each solution is computed in a number of steps that is polynomial in the size of* $\langle I, J\rangle$*.*

Chapter 9

A Revised Chase

Let us now reconsider the definition of a chase step for egds. As we mentioned above, our goal is to define a chase procedure that provides a basis for an efficient implementation, possibly by pruning unnecessary computations.

Example 8: To explain scalability issues, consider a simple functional dependency $A \rightarrow B$ over relation $R(A, B, C)$, with tuples $t_1 = R(1, 2, x)$, $t_2 = R(1, 2, y)$, $t_3 = R(1, 4, z)$, $t_4 = R(2, 5, w)$, $t_5 = R(2, 6, v)$.

In our definition so far, violations to dependencies are analyzed in a tuple-oriented fashion, i.e., by considering two tuples at a time. However, this is highly inefficient, in some cases. Consider first the forward update, i.e., the one that only changes values of the B attribute. It can be seen that eventually the B value of t_1, t_2, t_3 will all become equal. If we analyze violations in a tuple-oriented fashions, it will take several chase steps to realize this. Our goal, on the contrary, is to group and fix all violations together. In the literature [BFFR05, FG12] this has been formalized by means of *equivalence classes*.

However, we also have other solutions to generate, namely those that backward chase the value of the A attribute of some of the tuples above. We need a way to generate these chase steps as well. Therefore, we define our chase step in such a way that: (i) it considers all tuples in one homomorphism class together; (ii) it allows to specify whether any single tuple should be backward or forward chased, or even left untouched.

In order to do this, we need a number of preliminary definitions.

Recall the definition of a witness and witness variable given in Section 8. Consider our FD $A \rightarrow B$ over relation $R(A, B, C)$ introduced above. This becomes an extended egd of the form e. $\mathsf{R}(\mathsf{a}, \mathsf{b}, \mathsf{c}), \mathsf{R}(\mathsf{a}, \mathsf{b'}, \mathsf{c'}) \rightarrow \mathsf{b} = \mathsf{b'}$. Consider

tuples $t_1 = R(1,2,x)$, $t_2 = R(1,2,y)$, $t_3 = R(1,4,z)$, $t_4 = R(2,5,w)$, $t_5 = R(2,6,v)$. Here, the witness variable is **a**, and we have several homomorphisms, some with witness 1 and some others with witness 2. Our goal is to group homomorphisms with equal witnesses together.

Definition 27 [Homomorphism Class] Given an update Upd, and an egd $e : \forall\overline{x}\ (\phi(\overline{x}) \rightarrow x = x')$, let $\bar{x}_w \subseteq \bar{x}$ be the witness variables of e. An *equivalence* class for Upd and e, $\mathcal{H}$, is a set of homomorphisms of $\phi(\bar{x})$ into $\langle I, \mathsf{Upd}(J)\rangle$ such that all $h_i \in \mathcal{H}$ have equal witness values $h_i(\bar{x}_w)$.

Notice that homomorphism classes induce classes of tuples in a natural way. In our simple example above, the tuples are partitioned into two homomorphism classes, as follows: $ec_1 = \{t_1, t_2, t_3\}$ (with witness 1) and $ec_2 = \{t_4, t_5\}$ (with witness 2).

To identify a violation, we look for different values in the conclusion of e. To see an example, consider the homomorphism class ec_1 (witness 1), composed of the three tuples $\{t_1, t_2, t_3\}$: to identify the violation, we notice that they have two different values for the B attribute, 2 and 4, respectively. To formalize this, we introduce the notion of *witness groups* and *conclusion groups* for an homomorphism class $\mathcal{H}$:

Definition 28 [Witness Groups, Conclusion Groups] Given an homomorphism class $\mathcal{H}$ for Upd and egd e: (i) we call *witness groups*, $\mathsf{w\text{-}groups}_{\mathcal{H}}$, the set of cell groups associated by homomorphisms in $\mathcal{H}$ with the witness variables, $\bar{x}_w$; (ii) we call *conclusion groups*, $\mathsf{c\text{-}groups}_{\mathcal{H}}$, the set of cell groups associated by homomorphisms in $\mathcal{H}$ with the conclusion variables, x, x', of e:

$$\mathsf{w\text{-}groups}_{\mathcal{H}} = \{g_h(x_w) \mid h \in \mathcal{H}, x_w \in \bar{x}_w\}$$
$$\mathsf{c\text{-}groups}_{\mathcal{H}} = \{g_h(x) \mid h \in \mathcal{H}\} \cup \{g_h(x') \mid h \in \mathcal{H}\}$$

An homomorphism class for Upd and e generates a *violation* if it has at least two conclusion groups with different values and such that there is no ordering among them, i.e, there exist $g_1, g_2 \in \mathsf{c\text{-}groups}_{\mathcal{H}}$ such that $val(g_1) \neq val(g_2)$ and neither $g_1 \preceq_{\Pi,\mathsf{User}} g_2$ nor $g_2 \preceq_{\Pi,\mathsf{User}} g_1$. In this case, we say that e is *applicable* to $\langle I, \mathsf{Upd}(J)\rangle$ with $\mathcal{H}$.

In order to rework the notion of a chase step for an egd, we introduce the notion of a *repair strategy* for an homomorphism class. This will provide a hook to introduce our optimizations in the chase.

We are now ready to define the notion of a chase step for an egd. Our goal is to define the chase in such a way that it is as general as possible, but at the same time it allows to plug-in optimizations to tame the exponential

complexity. In order to do this, we introduce the crucial notion of a *repair strategy* for an homomorphism class, which provides the hook to introduce the notion of a cost manager in the next section.

Definition 29 [Repair Strategy] A *repair strategy*$rs_{\mathcal{H}}$ for an homomorphism class $\mathcal{H}$ is a mapping from the set of conclusion cell-groups, $\mathsf{c\text{-}groups}_{\mathcal{H}}$ of Upd and $\mathcal{H}$, into the set $\{f, b, u\}$ (where f stands for "forward", b for "backward" and u for "unaffected"). We call the *forward groups*, $\mathsf{forw\text{-}g}_{rs_{\mathcal{H}}}$, of $rs_{\mathcal{H}}$ the set of groups g_i such that $rs_{\mathcal{H}}(g_i) = f$, the *backward groups*, $\mathsf{back\text{-}g}_{rs_{\mathcal{H}}}$, those such that $rs_{\mathcal{H}}(g_i) = b$ and the *unaffected groups*, $\mathsf{equ\text{-}g}_{rs_{\mathcal{H}}}$, those such that $rs_{\mathcal{H}}(g_i) = u$, i.e., those that we don't want to repair in that particular chase step.

For each backward group $g \in \mathsf{back\text{-}g}_{rs_{\mathcal{H}}}$ and for each target cell $c_i \in g$ such that c_i is a cell of a relation in ϕ, we assume that the repair strategy $rs_{\mathcal{H}}$ also identifies (whenever this exists) one of the witness cells in $\mathsf{w\text{-}groups}_{\mathcal{H}}$ to be backward-repaired. This cell, denoted by $\mathsf{w\text{-}cell}_{rs_{\mathcal{H}}}(c_i)$, must be such that:

(i) there exists an homomorphism $h \in \mathcal{H}$ that covers both c_i and $\mathsf{w\text{-}cell}_{rs_{\mathcal{H}}}(c_i)$;

(ii) the corresponding cell group g_i according to Upd has a constant value, i.e., $val(g_i) \in$ CONSTS;

(iii) the corresponding cell group g_i has empty justifications, i.e., $just(g_i) = \emptyset$.

Repair strategies for homomorphism classes are the main building block to define how to generate chase steps for egds.

Definition 30 [Chase Step Strategy] Given a mapping & cleaning $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$, and an update Upd of J, a *chase step strategy css* is a triple $\{e, \mathcal{H}, rs_{\mathcal{H}}\}$, where e is an egd applicable to $\langle I, \mathsf{Upd}(J)\rangle$ with homomorphism class $\mathcal{H}$, and $rs_{\mathcal{H}}$ is a repair strategy for $\mathcal{H}$.

Notice that for a given update Upd and egd e, several different chase step strategies may exist: once for each different $rs_{\mathcal{H}}$ of any $\mathcal{H}$ that generates a violation for Upd and e. We denote by css_{Upd} the set of all possible *css* for Upd. Each chase step is defined based on a specific chase step strategy.

In our simple example 8, the dependency e is applicable to $\langle I, \mathsf{Upd}_{\emptyset}(J)\rangle$ with homomorphism class ec_1. The three conclusion groups are $g_1 : \langle 2 \rightarrow \{t_1.\mathsf{B}_{[2]}\}, by\ \emptyset\rangle$, $g_2 : \langle 2 \rightarrow \{t_2.\mathsf{B}_{[2]}\}, by\ \emptyset\rangle$ and $g_3 : \langle 4 \rightarrow \{t_3.\mathsf{B}_{[4]}\}, by\ \emptyset\rangle$.

Some possible repair strategies are: $rs_1 = \{f, f, f\}$, i.e. we choose to repair in a forward way all these conclusion groups; $rs_2 = \{f, b, f\}$, where we want to equate the first and third conclusion group, and repair backward the second one, for which the witness cell to be backward repaired is $t_2.\mathsf{A}$; $rs_3 = \{u, f, f\}$,

here we choose to leave unaffected the first conclusion group. The associated chase step strategies are $css_1 = \{e, ec_1, rs_1\}$, $css_2 = \{e, ec_1, rs_2\}$ and $css_3 = \{e, ec_1, rs_3\}$.

Definition 31 [CHASE STEP FOR EGDS] Given a mapping & cleaning $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$, and an update Upd of J. For each chase step strategy $css = \{e, \mathcal{H}, rs_{\mathcal{H}}\}$, a *chase step* generates a new update Upd_{css} defined as follows:

(*i*) to start, we initialize $\mathsf{Upd}_{css} = \mathsf{Upd}$

(*ii*) then, we replace all forward groups in $rs_{\mathcal{H}}$ by their least upper bound:

$$\mathsf{Upd}_{css} = \mathsf{Upd}_{css} - \mathsf{forw\text{-}g}_{rs_{\mathcal{H}}} \cup merge_{\preceq_{\Pi,\mathsf{User}}}(\mathsf{forw\text{-}g}_{rs_{\mathcal{H}}})$$

(*iii*) finally, we add the backward updates, i.e, for each backward group $g \in \mathsf{back\text{-}g}_{rs_{\mathcal{H}}}$, and cell $c_i \in occ(g)$, we replace $g_i = g_{\mathsf{Upd}}(\mathsf{w\text{-}cell}_{rs_{\mathcal{H}}}(c_i))$ by the cell group $g_i' = \langle L_i \rightarrow occ(g_i), by\ \emptyset, \mathsf{bckw}\rangle$ (where L_i is a new LLUN), as follows:

$$\mathsf{Upd}_{css} = \mathsf{Upd}_{css} - \{g_i\} \cup \{g_i'\}$$

Note that g' is the immediate successor of g_i according to $\preceq_{\Pi,\mathsf{User}}$.

We say that a chase step strategy css is *valid* for Upd if the chase of css in Upd generates an update Upd' such that Upd' differs from Upd in at least one cell group.

Given Upd, each valid chase step strategy $css^i \in css_{\mathsf{Upd}}$ generates a different step, Upd_{css^i}. We simultaneously consider all these chase steps, in parallel, and write

$$\mathsf{Upd} \rightarrow_{css_{\mathsf{Upd}}} \mathsf{Upd}_{css^0}, \mathsf{Upd}_{css^1} \ldots, \mathsf{Upd}_{css^n}$$

In our example, the chase of the chase step strategies css_1, css_2 and css_3 over $\langle I, \mathsf{Upd}_\emptyset(J)\rangle$ generates the following updates:
(*i*) Upd_{css_1} contains, in the place of conclusion groups g_1, g_2 and g_3, the new cell group $merge_{\preceq_{\Pi,\mathsf{User}}}(g_1, g_2, g_3) = \langle L_1 \rightarrow \{t_1.\mathsf{B}_{[2]}, t_2.\mathsf{B}_{[2]}, t_3.\mathsf{B}_{[4]}\}, by\ \emptyset\rangle$;
(*ii*) in order to generate Upd_{css_2} we need to forward chase g_1 and g_3, and change in a backward way the cell group associated to the cell $t_2.\mathsf{A}$ in $\mathsf{Upd}_\emptyset(J)$. The result is an update with two cell groups: $merge_{\preceq_{\Pi,\mathsf{User}}}(g_1, g_3) = \langle L_2 \rightarrow \{t_1.\mathsf{B}_{[2]}, t_3.\mathsf{B}_{[4]}\}, by\ \emptyset\rangle$ and $g_b = \langle L_3 \rightarrow \{t_2.\mathsf{A}_{[1]}\}, by\ \emptyset, \mathsf{bckw}\rangle$.
(*iii*) in Upd_{css_3} we leave untouched g_1, and we merge g_2 and g_3 in the new cell group $merge_{\preceq_{\Pi,\mathsf{User}}}(g_2, g_3) = \langle L_2 \rightarrow \{t_2.\mathsf{B}_{[2]}, t_3.\mathsf{B}_{[4]}\}, by\ \emptyset\rangle$;

Definition 32 [Chase Tree] Given a mapping & cleaning $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$, a chase is a tree whose root is $\langle I, J\rangle$, i.e., the empty update, and for each valid node Upd, the children of Upd are the updates $\mathsf{Upd}_0, \mathsf{Upd}_1, \ldots, \mathsf{Upd}_n$ such that one of the following conditions holds:
(a) for some $e \in \Sigma_e$ it is the case that $\mathsf{Upd} \rightarrow_{css_{\mathsf{Upd}}} \mathsf{Upd}_0, \mathsf{Upd}_1 \ldots, \mathsf{Upd}_n$;
(b) for some $m \in \Sigma_t$ and some h, it is the case that $\mathsf{Upd} \rightarrow_{d,m} \mathsf{Upd}_0$; or
(c) $\mathsf{Upd} \rightarrow_{\mathsf{User}} \mathsf{Upd}_0$.

The leafs are valid updates Upd_ℓ such that there is no dependency or user inputs applicable to $\langle I, \mathsf{Upd}_\ell(J)\rangle$. Any leaf in the chase tree is called a *result* of the chase of $\langle I, J\rangle$ with $\Sigma_t, \Sigma_e, \mathsf{User}$.

We denote by $\mathsf{revised\text{-}chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$ the chase tree obtained by this revised chase procedure.

Theorem 12 *Consider the chase tree* $\mathsf{chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$*, generated by the chase of* $\mathcal{MC}$ *over* $\langle I, J\rangle$ *as defined in Section 8. If the chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *terminates, then the revised chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *also terminates. In this case, the revised chase procedure generates a chase tree* $\mathsf{revised\text{-}chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$ *such that for any node in* $\mathsf{chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$*, there is an identical node in* $\mathsf{revised\text{-}chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$*.*

9.1 Introducing the Cost Manager

The chase procedure defined in the previous section provides an elegant operational semantics for cleaning scenarios. However, as argued above, computing all solutions has very high complexity, which makes the chase often impractical. In this section, we introduce a key component to improve the scalability of the chase, namely the *cost manager*.

Chasing at the equivalence-class level is more efficient than chasing at the tuple level, but by itself it does not reduce the total number of solutions, and ultimately the complexity of the whole chase process. In fact, previous proposals have chosen many different and often ad-hoc ways to reduce the complexity by discarding some of the solutions in favor of others. Among these we mention various notions of minimality of the updates [ABC99] [BFFR05] [BIG10], certain regions [FLM$^+$10], and sampling [BIG10]. We propose to incorporate these pruning methods into the chase process in a more principled and user-customizable way by introducing a component, called the *cost manager*.

Definition 33 [Cost Manager] Given a mapping & cleaning $\mathcal{M}$ and instance $\langle I, J\rangle$, a *cost manager* for $\mathcal{M}$ and $\langle I, J\rangle$ is a predicate CM over chase step

strategies to be used during the chase of the egds. For each chase step strategy *css*, it may either accept it (CM(*css*) = *true*), or refuse it (CM(*css*) = *false*).

During the chase of the egds, we shall systematically make use of the cost manager. Whenever we need to chase an homomorphism class, we only generate updates corresponding to repair strategies accepted by the cost manager. The standard cost manager is the one that accepts all chase step strategies, and may be used for very small scenarios. As an alternative, our implementation offers a rich library of cost managers. Among these, we mention the following, that have been used in experiments:

– a *maximum size* cost manager (SN): it accepts repair strategies as long as the number of leaves in the chase tree (i.e., the updates produced so far) are less than N; as soon as the size of the chase tree exceeds N, it accepts only the first one of them, and rejects the rest; as a specific case, the S1 cost manager only considers one order of application of the dependencies, and ignores other permutations;

– a *forward-only* cost manager (FO): it accepts forward-only repair strategies, and rejects those that perform backward updates;

– a *sampling* cost manager (SPLK): it randomly accepts repair strategies, until k solutions have been generated;

– a *certain-region* cost manager (CTN): it incorporates the notion of a *certain region* [FLM$^+$10] in the target, i.e., a set of attributes $\mathcal{A}$ that are considered "fixed", i.e., reliable, and cannot be changed; it refuses all chase steps in which changes are made to attributes in $\mathcal{A}$.

Notice that combinations of these strategies are possible, to obtain, e.g., a FO-S1 or a SPL50-FO-S5 cost manager. The FO-S5, for example, discards backward changes and, in addition, it considers five different permutations of the dependencies. In the following, we shall always assume that a cost manager has been selected in order to perform the chase.

Chapter 10

Comparison to Data Repairing Semantics

We have already shown in Section 7.6 that our general notion of a mapping and cleaning scenario can be restricted to mapping scenarios on one side, and cleaning scenarios, on the other side, and that our semantics is a conservative extension of data exchange.

In this section, we want to develop the comparison of mapping and cleaning scenarios to some of the semantics that have been proposed for data repairing [BFFR05, CFG$^+$07, BFM07, BIG10]. Since the partial order stands at the core of our approach, we also consider a few other recent proposals [SCM12, CFY13] that have dealt with notions of preference in connection with data quality constraints.

10.1 The Minimum Cost Algorithm

Let us start our discussion with early works by Fan and others on repairing by cell-modifications [BFFR05], and algorithms for conditional dependencies [CFG$^+$07, BFM07]. We find it useful to start by comparing our framework to the algorithm by Bohannon and others [BFFR05] using a simple data repairing problem where we are given a database table with schema $\mathcal{T}$, and a set of functional dependencies F over it, and an instance J of $\mathcal{T}$ that is dirty wrt F. We want to repair J by using the semantics given in [BFFR05], which we call the *minimum cost semantics*. The main ideas behind this semantics are the following:

- a *repair* is any instance of J' that is the result of updating J and satisfies the constraints in F;
- we seek repairs that *minimally differ* from J; the notion of minimality is based on the cost of updating J to obtain a repair J', according to an appropriate cost function;
- the actual repair algorithm is organized in two phases; in the first one the goal is to build equivalence classes (as introduced in Section 9), i.e., groups of cells that ultimately will become equal according to the dependencies, without actually modifying them; the decision about the value to update the cell is deferred to phase two, when its equivalence class has been determined, and more informed decisions can be taken.

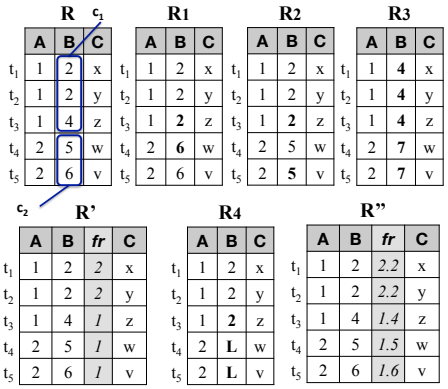


Figure 10.1: A Sample Data Repairing Scenario

Let us consider table $R(A, B, C)$ in Figure 10.1, with a simple functional dependency $d : A \rightarrow B$. The minimum cost algorithm would first build equivalence classes of cells (in our example c_1 for value 1, c_2 for value 2 of the A attribute, also shown in the figure), and then figure out a way to repair them. This is done by minimizing an elaborate cost function [BFFR05] that mixes together various features, like string similarity and confidence values. In its simplest form, however, all updates have the same unitary cost. Thus, to minimize the cost we need to to repair each equivalence class by the value with the highest frequency.

In fact, two minimum cost repairs for our example are R_1, R_2 in Figure 10.1. Notice how we don't have a clear policy to pick up a preferred value for the

second equivalence class, since both values have equal frequency. Repair R_3, on the contrary, is not minimal: by picking 4 as a preferred value for the first equivalence class, it generates an higher number of updates. It is also worth noting that, according to the semantics in [BFFR05], it is perfectly acceptable – although not minimal – to update the cells in c_2 to some arbitrary value, 7, that is different from both 5 and 6.

We now introduce a cleaning scenario that mimics this semantics. We have an empty source database, a target database that coincides with R, and a single egd that encodes the functional dependency d. We assume that no backward changes are allowed, i.e., we use a forward-only cost manager, and the user function is empty. The crux is to properly construct the partial order specification. Suppose we are given a dirty instance R that we want to repair. Our goal is to associate an ordering attribute to attribute B in R. From the theoretical viewpoint, this can be done by building a table R' that is obtained from R by adding a new attribute fr. For each tuple $t \in R$, the value of $t.\mathsf{fr}$ is the frequency of the value of $t.B$, as shown in Figure 10.1. We then specify fr with the natural order of integer numbers as the ordering attribute for B.

In practice, of course, this is not necessary. We may consider fr as a virtual attribute, and compute value frequencies on the fly. It is easy to see that our semantics applied to R' yields the minimal solution R_4 in Figure 10.1. There are a few important things to notice:

(i) there is a relationship (as we also discussed in Section 9) between equivalence classes and cell groups; in fact, in this example two cell groups are generated in the end, the first having occurrences that coincide with c_1, the second with c_2; cell groups, however, are more sophisticated than equivalence classes, since their semantics is such that we don't need to separate the class-construction step from the value-selection one. This is due to the *monotonicity property* of cell groups: they may only increase in size, and at every step the keep upgrading the quality of the target database;

(ii) our semantics is based on the assumption that arbitrary choices are to be avoided, because they correspond to unjustified ways of updating the target. To start, R_3 is not even a solution in our approach: a cell groups that updates $t_4.B$ to 7 would not be a valid cell group, because the constant is not motivated either by some occurrence, justification or user-input. In addition, our chase wouldn't even choose between 5 or 6 for that cell, because the two values are incomparable according to the partial order, and introduce a llun instead, that needs to be resolved later on by the user;

(*iii*) if we wanted to make an explicit choice between R_1 and R_2 without resorting to users, then we need to refine our partial order in such a way that all values for the fr attribute are different. One way to do this is to say that we use both frequency and value, and whenever two values are equally frequent, we pick up the higher one. We did this in table R'' in Figure 10.1. A minimal solution for R'' would be R_1; R_4 is a non-minimal solution for R'', while R_2 is not even a solution in this case.

Notice that the algorithms in [BFFR05, CFG+07, BFM07] deal with a few more features. We quickly discuss them here.
(*a*) *more sophisticated metrics and confidence values*: it is easy to generalize the frequency approach we have introduced here to incorporate string similarity or some other forms of value distance. In a similar way, as we have shown in our motivating example it is possible to handle confidence values and even more;
(*b*) *conditional functional dependencies and backward chasing* [CFG+07]: CFDs are considered as a source of authoritative values in our approach; they nicely blend with the partial order and need no ad hoc treatment. Similarly for backward chasing;
(*c*) *inclusion dependencies and conditional inclusion dependencies* [BFFR05, BFM07]: these papers develop hoc algorithms to handle the interaction of functional and inclusion dependencies. On the contrary, in our approach their interaction is nicely handled by our chase over cell groups.

10.2 The Sampling Algorithm

To further emphasize the flexibility of our approach, let us consider a second repair algorithm from the literature, that is completely different in spirit from the minimum-cost algorithm discussed above. Beskales and others have proposed a repair algorithm [BIG10] that combines together forward and backward chasing, a random strategy to repair cells, and sampling to reduce complexity. We call this the *sampling algorithm*. In essence, whenever a violation for an FD is detected, this algorithm may randomly decide to forward or backward repair it. It also nondeterministically chooses whether to introduce a variable to repair the conflict, or a random value from the active domain of the dirty table. The space of repairs is sampled in order to generate k repairs that have a minimality property.

We use two different techniques to reproduce this semantics in our approach. The tricky part is to design a partial order that "simulates" the random selection of values. To do this we may proceed along the lines of what we did

with the frequency attribute. In this case, we start with the dirty database, and associate with each attribute A a (virtual) additional attribute $rndA$; for each cell of A, we initialize the corresponding cell of $rndA$ with a randomly assigned value. This guarantees that values will be preferred to each other in a completely random way. Then, we adopt a sampling cost manager, that randomly decides whether to accept or refuse a chase step, until k solutions have been generated.

10.3 Prioritized Repairing

We now want to compare our partial order with different notions of preference that have been recently introduced in connection with data quality constraints. We start by considering the work on prioritized repairing [SCM12]. This research is is inspired by works on preferred models for logic programs, and is somehow different in spirit from our work. While we focus primarily on obtaining preferred solutions by means of a general chase procedure, their focus is on the complexity of repair checking, and on consistent query answering.

There are also significant differences in terms of the language of dependencies, and update strategies. Prioritized repairs consider *subset repairs* (i.e., tuple deletions only), and denial constraints with no constants. While cleaning egds can be extended to capture arbitrary denial constraints, their update primitives are considerably different from the ones we use (cell updates, and no deletions). These differences are such that the two algorithms are quite different in nature.

Nevertheless, our partial order has points of contact with their notion of a prioritized repair, and therefore we find it interesting to compare the two approaches. In this respect, we believe that our partial order over cells and cell groups is more flexible. In fact, prioritized repairs rely on preference orders that are specified over tuples, and lift them to sets of tuples. On the contrary, we specify preference orders over cells, and lift them to cell groups, i.e., sets of cell modifications. This finer granularity of our approach makes our notion of an upgrade more general than their notion of *global optimal repair*.

To see this, consider the simple example in which we have a single table $R(A, B, C)$ with a functional dependency $A \rightarrow B, C$, and a dirty instance $I = \{t_1 : R(a, 1, 4), t_2 = R(a, 2, 3)\}$. Suppose our partial order specification states that, for any attribute, cells with higher values should be preferred to the ones with lower values; this gives us the following minimal solution: $J = \{R(a, 2, 4)\}$.

First, we notice that J is not a repair in their setting, since any of their

repairs must either correspond with $t1$ or $t2$ only (depending on the preference relation on tuples, and therefore on the tuple that is deleted to satisfy the FD). Second, by changing our partial order specification, we can easily simulate their semantics. Suppose, in fact, we say that for attribute B we prefer cells with lower values, while for attribute C the ones with higher values; then we have a minimal solution: $J' = \{t_1 = R(a, 1, 4)\}$ that coincides with their globally optimal repair.

There are a few other restrictions associated with prioritized repairing that we don't need to impose, namely the acyclicity restriction on preference relations, and the notion of Pareto optimality.

Source #1 – Master Data

NBA

	FN	LN	League	Season	Team
t_{m1}	Michael	Jordan	NBA	1994-95	Chicago Bulls
t_{m2}	Michael	Jordan	NBA	2001-02	Washington Wz

Dependencies

$d_1 = (\text{ID} \rightarrow \text{FN, LN})$
$d_2 = (\text{ID, FN} \rightarrow \text{MN})$
$d_3 = (\text{ID, League} \rightarrow \text{TotPts, J, Team})$
$d_4 = (\text{ID, League, J} \rightarrow \text{Rnds})$
$d_5 = (\text{ID, League, Team} \rightarrow \text{Arena})$
$d_6 = t_m[\text{FN, LN, Season='1994-95'}], t[\text{FN, LN}] \rightarrow t_m[\text{League, Team}] = t[\text{League, Team}]$

$\Pi = \{P_{\text{FN}} = D_{\text{MN}},\ P_{\text{J}} = D_{\text{Rnds}},\ P_{\text{TotPts}} = D_{\text{Rnds}},\ P_{\text{Rnds}} = D_{\text{Rnds}}\}$

Initial Target

STATS

	ID	FN	MN	LN	Rnds	TotPts	J	League	Team	Arena
t_1	1	MJ	*null*	*null*	16	424	45	NBA	Chicago	Chicago Stadium
t_2	1	Michael	*null*	Jordan	27	772	23	NBA	Chicago Bulls	United Center
t_3	1	Michael	*null*	Jordan	1	19	45	NBA	Chicago Bulls	United Center
t_4	1	Michael	Jeffrey	Jordan	127	51	45	SL	Birmingham Barons	Regions Park
t_5	2	Larry	Joe	Bird	13	234	33	NBA	Boston Celtics	*null*
t_6	2	Larry	*null*	Bird	45	908	33	NBA	Boston Celtics	TD Garden

Solution #1

STATS

	ID	FN	MN	LN	Rnds	TotPts	J	League	Team	Arena
t_1	1	Michael	Jeffrey	Jordan	27	772	23	NBA	Chicago Bulls	L_1
t_2	1	Michael	Jeffrey	Jordan	127	51	45	SL	Birmingham Barons	Regions Park
t_3	2	Larry	Joe	Bird	45	908	33	NBA	Boston Celtics	TD Garden

Figure 10.2: The Michael Jordan Example

10.4 Relative Accuracy

We conclude our discussion by a discussion of a recent paper by Cao and others [CFY13]. This paper studies a rather specific problem, called by the authors the *accuracy problem* which falls within the reach of *entity resolution* [BGMM$^+$09] rather than constraint-based data repairing. It is formulated as follows: we are given a set of records I_e with the same schema, that correspond to a description of a single real world entity e. These records may have conflicting values, and the goal is to derive a single tuple t_e, which we call the *entity tuple*, with the *most accurate values* for all attributes. Master-data tuples may be used during the process.

One example of this problem is shown in Figure 10.2. Consider for now only tuples $t_1 - t_4$, that refer to Michael Jordan. The goal is to unify them within a single tuple that reflects the most accurate values for season 1994-95.

While their algorithms do not aim at repairing an arbitrary database instance that is dirty wrt a set of constraints, there are some points in common with our approach. The authors develop a language of *accuracy rules* that have two goals:

(*i*) on the one side, they can specify a partial order among target values; they write $t \prec_{\mathsf{rnds}} t'$ to denote that the value in cell $t'.\mathsf{rnds}$ is more accurate than the one in cell $t.\mathsf{rnds}$; this may happen, for example, because $t_1.\mathsf{rnds} < t_2.\mathsf{rnds}$, i.e., tuple t' contains stats that are more current than those in t; this can be stated as follows:

$$a_1 : \forall t_1, t_2 \in \mathsf{Stats} : (t_1.\mathsf{leag} = t_2.\mathsf{leag}, t_1.\mathsf{rnds} < t_2.\mathsf{rnds} \rightarrow t_1 \prec_{\mathsf{rnds}} t_2)$$

in addition, accuracy rules can be used to infer accuracy relationships among attributes:

$$a_2 : \forall t_1, t_2 \in \mathsf{Stats} : (t_1 \prec_{\mathsf{rnds}} t_2 \rightarrow t_1 \prec_{\mathsf{totalPts}} t_2)$$

i.e., the total number of points is more accurate in those tuples that have a more accurate number of rounds;

(*ii*) on the other side, they may be used to correct the entity tuple t_e based on master data tuples, like the ones shown in Figure 10.2 within table NBA.

The authors develop algorithms to dynamically handle the construction of the entity tuple while at the same time deriving the partial order of accuracy among attribute values. The main concern, here, is about the termination and confluence of the process, i.e., if the algorithm terminates, and if it returns the same identical tuple regardless of the order in which accuracy rules are fired. This cannot be guaranteed in all cases.

Let us remark again that this is not a general-purpose data repairing algorithm, since it does not contemplate constraints and makes the strong assumption that all tuples represent a single entity. Still, we find it very useful to compare our approach to this to emphasize a couple of important differences.

The main difference is that our approach to the partial order is immune from the termination and confluence problems discussed in [CFY13]. In fact, the partial-order specification of a mapping & cleaning scenario fixes a partial order of the cells of the initial instance, $\langle I, J \rangle$, which never changes during the chase. In other terms, our algorithm clearly separates the definition of the partial order for cells, that is done once and for all over $\langle I, J \rangle$ before the repair process starts, and the generation of the actual updates using cell groups. This clear separation, along with the monotonicity property of cell groups, guarantees that our chase procedure for cleaning scenarios always terminates

and gives deterministic results (modulo the order of application of egds, which is a totally different problem).

On the contrary, the relative accuracy algorithm adopt a dynamic strategy to derive the partial order by interleaving the firing of accuracy rules and master data rules. Despite the fact that our definition of the partial order is static, we believe that our semantics guarantees most of the benefits of accuracy rules, without the associated shortcomings. The intuition behind this is that accuracy rules in essence do two things: (i) fix a natural order for the values of an attribute, as discussed above for rule a_1; (ii) they propagate this ordering to other attributes, as in rule a_2 above. But this is exactly what our partial order specification does.

To show this, in Figure 10.2 we report a translation of the Michael Jordan example as a cleaning scenario. Recall that the original problem was not a data repairing problem. Therefore we needed to make some changes, to adapt it to our setting. First, the target table Stats now may freely contain data about different players (we added tuples for Larry Bird as well). Second, we needed to specify a set of data quality constraints for this example to state that ID is a key, in order to trigger the repair of tuples for each player, and an editing rule to correct tuples using master-data. In Figure 10.2 we also report our partial order specification for this example, and the minimal solution returned by our algorithm (duplicate tuples have been removed for the sake of readability), which is in line with the expected results reported by Cao and others [CFY13].

In conclusion, the formalism of accuracy rules and ours have different inspirations and are not directly comparable. Loosely speaking, accuracy rules are a very expressive language to encode preference relations, but their dynamic nature is such that they not always interact in the proper way with the entity resolution process.

Our partial order specification is less expressive, but it is static and therefore free from termination and confluence issues: in fact, in cleaning scenarios a solution is always reached. One may wonder how in this example we can achieve basically the same results with our static partial order. The answer is again cell groups. In fact, while the partial order of cells in the original database is fixed and static, the partial order of cell groups naturally evolves during the chase. This evolution obeys the nice law that it is guaranteed to improve the quality of the target in a monotonic way. In light of this, we believe that this example is another proof of the flexibility of our approach.

Chapter 11

Scalability

11.1 Delta Databases

Even with cost managers in place, the parallel nature of our chase algorithm imposes to store a possibly large tree of updates. A naive approach in which new copies of the whole database are created whenever we need to generate a new node in the tree, is clearly inefficient. To solve this problem, we introduce an ad-hoc *representation system* for nodes in our chase trees, called *delta databases*. Delta databases are a formalism to store a finite set of worlds into a single relational database. Intuitively, they allow to store "deltas", i.e., modifications to the original database, rather than entire instances as is done in the naive approach.

Delta relations rely on an attribute-level storage system, inspired by *U-relations* [AJKO08], modified to efficiently store cell groups and chase sequences. More specifically, (*i*) each column in the original database is stored in a separate *delta relation*, to be able to record cell-level changes; (*ii*) chase steps are identified by a function with a prefix property, such that the id of the father of n is a prefix of the encoding of n; this allows to quickly reconstruct the state of the database at any given step, using fast SQL queries; (*iii*) additional tables are used to store cell groups, i.e., occurrences and justifications.

More formally, we introduce a function *stepId*() that associates a string id with each chase step, i.e., with each node in the chase tree, and has the prefix property such that for each n, $stepId(father(n))$ is a prefix of $stepId(n)$. For this, we use the function that assigns the id r to the root, *r.0, r.1, ..., r.n* to its children, and so on.

Definition 34 [DELTA DATABASE] Given a target database schema $\mathcal{R} = \{R_1, \ldots, R_k\}$, a *delta database* for $\mathcal{R}$ contains the following tables: (*i*) a *delta table* $R_i\_A_j$ with attributes (*t_{id}, stepId, value*), for each R_i and each attribute A_j of R_i; (*ii*) a table *occurrences*, with schema (*stepId, value, t_{id}, table, attr*); (*iii*) a table *justifications*, with schema (*stepId, value, t_{id}, table, attr*).

During the chase, we store the whole chase tree into the delta database. We do not perform updates, which are slow, but execute inserts instead. Whenever, at step s, a cell $t_{id}.A$ in table R is changed to value v, we store a new tuple in the delta table R_A with value $(t_{id}, stepId, v)$. Using this representation, it is possible to store trees of hundreds of nodes quite efficiently. In addition, it is relatively easy to find violations using SQL (the actual queries are omitted for space reasons).

In the next section we show how the combination of our advanced chase procedure and its implementation under the form of delta databases scale to large repairing problems with millions of tuples and large chase trees.

11.2 Optimizations to the Chase

Our goal in this section is to introduce new optimizations that guarantee good performance when chasing tgds, and at the same time considerably improve performance on egds.

When Does the Chase Scale?

Of the many variants of the chase, the ones that scale nicely are those that can be implemented as queries in a first-order language, and therefore as SQL scripts. To give an example, consider the s-t tgd $R_1(x, z), R_2(x, v) \rightarrow \exists y : R_3(x, y)$. Assume R_3 is empty. Then, as it was detailed in [tCCKT09], chasing this tgd amounts to run the following SQL statement, where $\mathsf{sk}(x)$ is a *Skolem term* used to generate the needed labeled null:

$$\mathsf{insert\ into}\ R_3\ \mathsf{select}\ x,\ \mathsf{sk}(x)\ \mathsf{from}\ R_1,R_2\ \mathsf{where}\ R_1.x = R_2.x$$

We call this a *batch chase execution*. In fact, chasing s-t tgds, or even the more general *FO-rules* [MPR12] is extremely fast. On the contrary, the chase becomes slow whenever it needs to be executed in a *violation-by-violation* fashion. Unfortunately, our chase procedure does not allow for easy batch-mode executions, because of a crucial factor: during the chase, we need to keep track of cell groups, and properly maintain them. Repairing a violation for either a

tgd or an egd changes the set of cell groups, and therefore may influence other violations.

In our approach, cell-groups are stored during the chase using two additional database tables, one for occurrences, one for justifications.

Consider now the tgd above, and assume also R_1, R_2 are target tables. Suppose our chase is at step s. In our approach, to chase the tgd by literally following the definition of a chase step, we need to do the following: (i) query the target to join R_1, R_2 to find a tuple t that satisfies the premise; (ii) query R_3 to check that t contains a value of x that should actually be copied to R_3; (iii) add the new tuple to R_3; in addition, we also have to properly update cell groups; to do this: (iv) for each cell associated in t with variable x, we need to query tables occurrences and justifications to extract the cell group of the cell, and build the a new cell group as the union of these; (v) store the new cell group for x in tables occurrences and justifications; (vi) do the same for the existentially quantified variable, y. Then, move to the next violation and iterate.

It is easy to see that this amounts to perform several thousands of queries, even for a very small database. More importantly, we are forced to mix queries, operations in main memory, and updates to the database, and send many single statements to the dbms using different connections, with a perverse effect on performance. In the next paragraphs, we develop a number of optimizations that alleviate this problem.

Caching Cell Groups

A key optimization in order to speed up the chase consists in caching cell groups in main memory. This, however, has a number of subtleties. We tested several caching strategies for cell groups. The first, straightforward one, is a typical *cache-aside, lazy loading* strategy, in which a cell group is first searched in the cache; in case it is missing, it is loaded from the database and stored in the cache. As it will be shown in our tests, this strategy is too slow.

Greedy strategies perform better. We tried a *cache-as-sor, greedy* strategy in which the first time a cell group for a step s is requested, we load into the cache all cell groups for s, with two queries (one for occurrences, one for justifications). This strategy works very well for the first few steps. Then, as soon as the chase goes on, for large databases it tends to become slower since the main memory limit is easily reached (no cell group is ever evicted from the cache), and some of the cell groups need to be swapped out to the disk.

Since accessing the file system on disk is slower than querying the database, performances degrade.

To find the best compromise between storage-efficiency and performance, we noticed that our chase algorithm has a high degree of locality. In fact, when chasing node s in the tree to generate its children, only cell groups valid at step s are needed. Then, after we move from s to its first child, s', cell groups of s will not be needed for a while. We therefore designed a *single-step, greedy* caching strategy, that caches cell groups for a single step at a time. In essence, we keep track of the step s currently in the cache; whenever a cell group for a different step s' is requested, we clean the cache and load all cell groups for s'. Our experiments show that this brings excellent improvements in terms of running times.

Chasing Tgds in Batch Mode

A second, major optimization, consists in chasing tgds in batch mode. In essence, we want to clearly separate updates to the dbms (that are much more efficient when are run in batch mode), from the analysis and update of cell groups. To do this, we use a multi-step strategy that we shall explain using our sample tgd above. As a first step, we update the dbms in batch mode. To avoid the introduction of unnecessary tuples, we insert into table R_3 only those tuples that contain values that are not already in R_3, by the following statement:

$$\begin{array}{l}\mathsf{insert\ into}\ R_3\ \mathsf{select}\ x,\ \mathsf{sk}(x)\ \mathsf{from}\ R_1{,}R_2\ \mathsf{where}\\ \qquad R_1.x{=}R_2.x\ \mathsf{and}\ x\ \mathsf{not\ in}\ (\ \mathsf{select}\ x\ \mathsf{from}\ R_3).\end{array}$$

Once all of the needed tuples have been inserted into the database, we maintain cell groups. To do this, we store all values of x that have been copied to R_3 into a violations temporary table. Then, we run the following query, that gives us the cells for which we need to update cell groups:

$$\begin{array}{l}\mathsf{select}\ R_1.x, R_2.x, R_3.x, R_3.y\ \mathsf{from}\ R_1, R_2, R_3\\ \qquad \mathsf{where}\ R_1.x = R_2.x\ \mathsf{and}\ R_2.x = R_3.x\\ \qquad \mathsf{and}\ x\ \mathsf{in}\ (\mathsf{select}\ x\ \mathsf{from\ violations}).\end{array}$$

We scan the result, and properly merge the cell groups. Notice that this step is usually very fast, since we use the cache. Finally, we update the occurrence and justifications table.

Chasing Egds in Batch Mode

We also use an aggressive strategy to chase egds. Generally speaking, violations for egds should be solved one at a time, since they interact with each other. Consider for example this common egd, encoding a conditional functional dependency: $R(x), S(x, y) \rightarrow x{=}y$, where S is source table. Assume the following tuples are present $R(1), S(1, a), S(1, b)$. We first query the database to find out violations using the following query:

$$\mathsf{select}\ x, y\ \mathsf{from}\ R, S\ \mathsf{where}\ R.x = S.x\ \mathsf{and}\ x <> y.$$

This will return two violations, the first arising from $R(1), S(1, a)$, the second from $R(1), S(1, b)$. However, as soon as we repair the first one and change $R.x$ to a, the second violation disappears. To see this, it is necessary to repeat the query and realize that the result is empty.

Despite this, we do not want to process violations one at a time, but rather in batch mode. During the chase, we keep track in main memory of the cell groups that need to be maintained to solve violations. Before writing updates to the database, we check if the resulting set of cell groups is *consistent* with each other, i.e., each cell of the database is changed only once. As soon as we realize that a cell group is not consistent, we discard the update and iterate the query.

Chapter 12

Experimental Result

12.1 Prototype

The proposed algorithms have been implemented in a working prototype of the LLUNATIC system, written in Java. A preliminary release of the system is already available on the project page (`http://db.unibas.it/projects-/llunatic/files`). We plan to release the prototype under an open-source license on one of the major open-source repositories. LLUNATIC comes with a GUI developed using the NetBeans Platform and reported in Figure 12.1, that allows users to easily specify a scenario, explore initial instances and configure the core aspects of the repair process. In particular users can specify the partial order in a declarative way, associating ordering attributes or writing a custom JavaScript code.

The scenario specification is then handled by the tool, which runs a parallel-chase procedure, that generates a chase tree, as shown in Figure 12.1. Leaves in the chase tree are solutions and, for each of them, it is possible to explore all the chase steps, from the root to the leaf, and analyze the modifications, expressed by cell groups. As already said, they not only specify how to change the cells of the database, but also carry full provenance information for the changes, in terms of (a) original values of the cells in the target database; (b) relationships to source and master-data values, if these exist; and (c) user interventions to repair the cells (see Figure 12.1).

The real power of LLUNATIC stands in the possibility to control the solution-generation process in a very fine way. In particular the number of alternative solutions are defined by using the cost manager, and the interaction with the

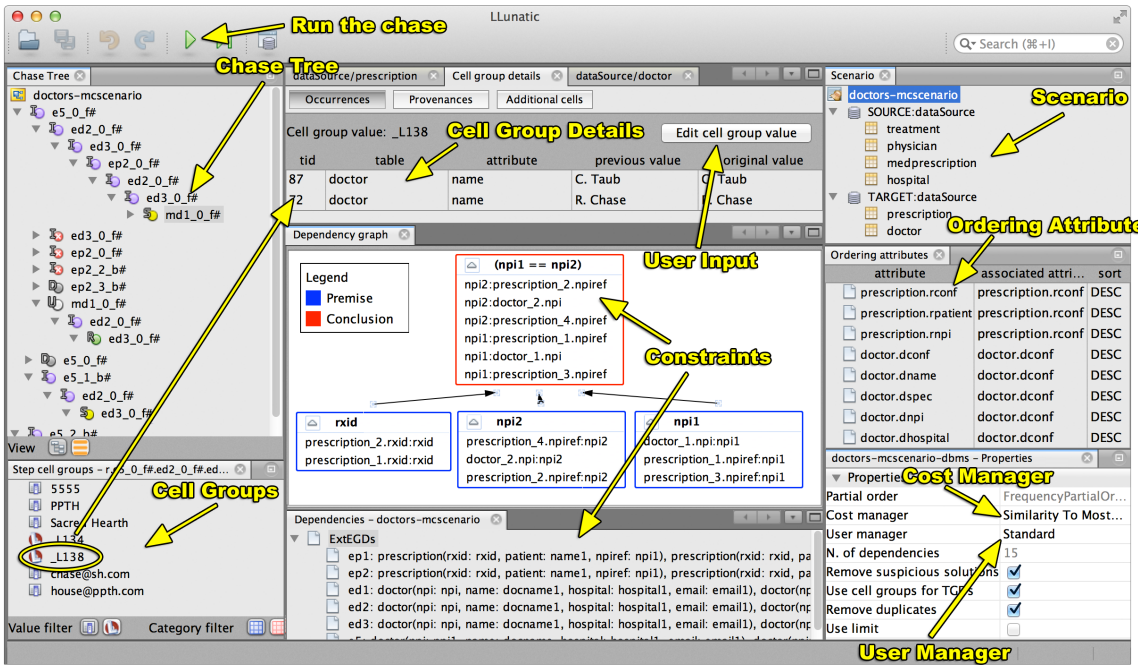


Figure 12.1: LLUNATIC in action

users is handled by the user manager.

A cost manager defines how to prune the chase tree and discard partial solutions along the process to limit the size of the output. Each cost manager is defined by a type and a set of configuration values.

The user interaction is another crucial feature of our system: indeed in those cases in which no preference rule is available, LLUNATIC does not make arbitrary choices, and rather marks conflicts so that users may resolve them later on.

Based on this, LLUNATIC offers powerful features to collect user-inputs. First, it is allows to declaratively specify when the chase should be (temporarily) paused to collect inputs from the user by plugging *user managers* into the chase. A user manager is a declarative condition over the chase tree that stops the chase and asks the user for input. There are several strategies supported:

(*i*) *Interactive* stops the chase and asks for inputs after each new node is added to the tree; (*ii*) *AfterLLUN* only stops for nodes that contain LLUNS; (*iii*) *AfterFork* stops the chase every time one child is created.

When the chase stops and the user is invoked, s/he may pick up a node in the chase tree, consult its history in terms of changes to the original database, inspect the lluns that have been introduced, and analyze the associated cell groups. Based on this, informed decisions are taken in order to remove lluns and replace them with the appropriate constants (see Figure 12.1).

Working with cost managers concretely allows users to explore the trade-offs between the quality of updates, and the cost of their generation. Even more important, cost managers, user-specified preference rules, and user-inputs give a fine-grained control over the solution-generation process, and can be used to learn important lessons, as discussed in the next section.

12.2 Experimental Result

In this section, we consider several cleaning scenarios, of different nature and sizes, and study both the quality of the updates computed by our system, and the scalability of the chase algorithm. We show that our algorithm produces updates of better quality with respect to other systems in the literature, and at the same time scales to large databases. All experiments have been executed on an Intel i7 machine with 2.6Ghz processor and 8GB of RAM under MacOS. The DBMS was PostgreSQL 9.2.

The section is organized as follows. We start by introducing the datasets and how they are used in the three kinds of scenarios we support. We describe the way errors are introduced in the datasets and how solutions are evaluated with several metrics. We then introduce alternative algorithms to obtain solutions and compare them against LLUNATIC.

Datasets and Scenarios

We selected three datasets. The first two are based on real data from the US Department of Health & Human Services (http://www.medicare.gov/hospitalcompare/), the third one is synthetic. More details about datasets and transformations are reported in the Appendix.

(*a*) Hospital-Norm is the normalized version of the hospital data, of which we considered 3 tables with 2 foreign keys, a total of 20 attributes, and approximately 150K tuples.

(*b*) Hospital-Den is a highly denormalized version of the same data, with 100K tuples and 19 attributes, traditionally used in data quality experiments, over which we specified 9 functional dependencies. This second version has traditionally been used in data quality experiments to test algorithms that were restricted to single-table databases. For both Hospital datasets, in our scalability tests we generated instances of size up to 1M tuples by replicating the original data several times.
(*c*) Customers, corresponds to our running example in Figure 1. The source database schemas contain 3 tables, plus 1 master data table and 2 additional tables encoding constants in CFDs. The target database schema contains 2 tables. Dependencies are the ones in Section 1.

We synthetically generated up to 1M tuples for the 4 source tables, with a proportion of 40% in MedTreatments, 40% in Surgeries, and 20% in Patients; the master-data table contains a few hundreds of the tuples present in MedTreatments and in Patients. We consider master-data tuples outside the total, as they cannot be modified. For the target, we generated up to 1M tuples, with a proportion of 40% in the Customers table, and 60% in Treatments;

Based on these datasets, we defined 5 scenarios, one for Type-1, two for Type-2 and two for Type-3. For each scenario we also fixed an expected solution, called DB_{exp}, as follows:
(*i*) a Type-1, data exchange scenario Customers-DE based on a version of the Customers dataset for which there are no conflicts among the sources and an empty target database; we generated a clean and consistent version of the source tables, and based on those the expected instance is the core universal solution for the set of tgds and egds in Section 1;
(*ii*) a Type-2, cleaning scenario Hospital-Den-CL based on the Hospital-Den dataset, with 1 table, 9 functional dependencies, and the standard partial order specification; the expected instance, in this case, is the original table;
(*iii*) a Type-2, cleaning scenario Customers-CL based on the Customers dataset, with the 2 target tables Customers and Treatments, 3 source tables (1 master data table and 2 additional tables encoding constants in CFDs), the 9 extended egds reported in Section 3, and the partial order discussed in Section 4.3; the expected instance, in this case, is the original table;
(*iv*) a Type-3, mapping & cleaning scenario Hospital-Norm-MC based on the Hospital-Norm dataset, with 3 tables, 2 tgds and 12 egds, and the standard partial order specification; the expected instance, in this case, corresponds to the original tables;
(*v*) a Type-3, mapping & cleaning scenario Customers-MC based on the Customers dataset, with the set of tgds and egds in Section 1, (a total of 6 source tables,

2 target tables, 3 tgds and 9 egds), and the partial order in Section 4.3, and a non-empty target database; since we are integrating several sources, fixing the expected instance in this case is less obvious. We consider the clean and consistent versions of the source tables used for scenario Customers-DE, and the core universal solution, C, of the mapping scenario. Then, we introduced random noise and inconsistencies in the sources, and fed them to the mapping and cleaning scenario. We intend to measure to which extent our algorithm is capable of generating a consistent and minimal repair of the target database. To do this, we adopt as an expected solution the core universal solution C above.

It is worth noting that these scenarios somehow represent opposite extremes of the spectrum of data-repairing problems. In fact, the Hospital-Den-CL and Hospital-Norm-MC scenarios contain functional dependencies only, and therefore are quite standard in terms of constraints. However, Hospital-Den-CL can be considered a worst-case in terms of scalability, since all data are stored as a single, non-normalized table, with many attributes and lots of redundancy; over this single table, the 9 dependencies interact in various ways, and there is no partial-order information that can be used to ameliorate the cleaning process. On the contrary, the Customers-CL scenario contains a complex mix of dependencies; this increased complexity of the constraints is compensated by the fact that data are stored as normalized tables, with no redundancy, and preference strategies are given for some of the attributes.

Errors Induction

In order to test our algorithms with different levels of noise, we introduced errors in the datasets. Part of these errors were generated by a random-noise generator. However, in order to be as close as possible to real scenarios, in the HOSPITAL datasets we also used a different source of noise. We asked workers from Mechanical Turk (MT) (https://www.mturk.com/mturk/) to perform data entry for a random sample of tuples from the original database. Workers were shown the original tuple under the form of a jpeg image, and needed to manually copy values into a form. In order to make all the errors detectable by the constraints, we let the workers hand copy only values for attributes involved in constraints. We then extended the noise to the entire datasets by simulating the error patterns of the workers with a program over the clean data. We used different groups of workers with different approval rates; approval rates measure the quality of a worker in terms of the percentage of previous jobs positively evaluated within MT. Approval rates varied between 50% and 99%; for these,

we observed a percentage of wrong values between 5% and 1%. These errors were then complemented with those generated by the random noise generator. Errors have been added to the other datasets with the same procedure.

For all datasets, we generated dirty copies with an increasing number of noisy cells (ranging from 1% to 10% of the total depending on the scenario). Changes to the original values were done only for attributes involved in dependencies, in order to maximize the probability of generating detectable violations. All perturbations are detectable by the constraints (we discarded any generated perturbation over attributes that were not involved in a dependency).

Quality Metrics

For all scenarios, we measure running times and the size of the chase trees. For evaluation of the quality of the solutions for Type-2 and Type-3 scenarios, we have four quality metrics.

The quality of repair algorithms (Type-2) have been traditionally measured by considering a single table with an immutable set of cells, and by reporting precision and recall in terms of dirty cells that have been restored to the original values. More specifically, for each clean database, we generated the set $\mathcal{C}_p$ of perturbated cells. Then, we run each algorithm to generate a set of repaired cells, $\mathcal{C}_r$, and computed precision (P), recall (R), and F-measure ($F = 2 \times (P \times R)/(P + R)$) of $\mathcal{C}_r$ wrt $\mathcal{C}_p$. Since several of the algorithms may introduce variables to repair the database – like our lluns – we calculated two different metrics.

Metric 0.5. The first one is the one adopted in [BIG10], which we call *Metric 0.5*: (*i*) for each cell $c \in \mathcal{C}_r$ repaired to the original value in $\mathcal{C}_p$, the score was 1; (*ii*) for each cell $c \in \mathcal{C}_r$ changed into a value different from the one in $\mathcal{C}_p$, the score was 0; (*iii*) for each cell $c \in \mathcal{C}_r$ repaired to a variable value, if the cell was also in $\mathcal{C}_p$, the score was 0.5. In essence, a llun or a variable is counted as a partially correct change. This gives an estimate of precision and recall when variables are considered as a partial match.

Metric 1.0. Since our scenarios may require a consistent number of variables, due to the need for backward updates, and this metric disfavors variables, we also adopt a different metric, which counts all correctly identified cells. In this metric, called *Metric 1.0*, item (*iii*) above becomes: for each cell $c \in \mathcal{C}_r$ repaired to a variable value, if the cell was also in $\mathcal{C}_p$, the score was 1.

In mapping and cleaning (Type-3), on the contrary, we may have different tables, referential integrity constraints, and the addition of new cells to the target. The presence of new cells makes it impossible to reuse the traditional

metrics. Given a clean target database, we need for each repair a general algorithm to measure the similarity of the whole, multi-table repaired database to the expected target database. A general and efficient algorithm to measure the *similarity* of two complex databases by taking into account foreign keys, different cell ids, and placeholders, like labeled nulls or lluns has been recently developed in [MPRS12], and we adopt it for this metric. Based on this algorithm, we report two different quality measures.

Metric Sim. The first one is the similarity, $sim(\mathsf{Upd}, DB_{exp})$, measured by the algorithm in [MPRS12]. In the comparison, lluns are considered as partial matches, and counted as 0.5 each.

Metric Rep-rate. In the Hospital-Norm-MC this measure can be misleading. There we start with a clean target database, DB_{clean}, and introduce random noise to generate a dirty database, DB_{dirty}. On average, the dirty copy is approximately 90% similar to the clean one, and therefore all repairs will also have high similarity to the clean instance. In this case we report a repair rate defined as:

$$rep\text{-}rate(\mathsf{Upd}, DB_{exp}) = \frac{1 - (1 - sim(\mathsf{Upd}, DB_{exp}))}{(1 - sim(DB_{dirty}, DB_{exp}))}$$

In essence, we measure how much of the original noise a repairing algorithm actually removed. Whenever an algorithm returned more than one repair for a database, we calculated the maximum, minimum, and average quality.

Algorithms

We ran LLUNATIC with several cost managers and several caching strategies, as discussed in Sections 9, 11. In particular we used a new kind of cost manager, called *frequency* cost manager (FR), that adopts the following rules in order to repair an homomorphism class $\mathcal{H}$ for dependency e: it relies on the frequency of values appearing in conclusion cells, and on a similarity measure for values (based on the Levenshtein distance for strings); then: (i) it rejects repair strategies that backward-chase cells with the most frequent conclusion value; (ii) for every other conclusion cell, if its value is similar (distance below a fixed threshold) to the most frequent one, the cell is forward-chased; otherwise, it is backward chased; this is typically used with a frequency rule in the partial order of cell-groups;

We chose variants of the LLUNATIC-FR-sN cost manager – the frequency cost-manager that generates up to N solutions – with $N = 1, 10, 50$, and the LLUNATIC-FR-s1-FO, the forward-only variant of LLUNATIC-FR-s1. We do

not report results obtained by the standard cost manager, as it only can be used with small instances due to its high computing times.

In order to compare our system to previous approaches, we tested the following algorithms from the literature, implemented as separate systems:

For Type-1, the DEMO system [PS09], as the state of the art chase engine for mappings.

For Type-2, three repair systems:
(*a*) *Minimum Cost* [BFFR05] (MIN. COST);
(*b*) *Vertex Cover* [KL09] (VERTEX COVER);
(*c*) *Repair Sampling* [BIG10] (SAMPLING), for which, for each experiment, we took 500 samples, as done in the original paper.

For Type-3, we used MIN. COST for repair scenarios with FDs and IDs (in which IDs are repaired only by tuple insertions, and not by deletions or modifications), and an implementation of the PIPELINE algorithm in Section 3 for mapping and cleaning scenarios. The latter is obtained by coupling a standard chase engine for tgds, and the SAMPLING algorithm for FDs in [BIG10]; here, for each experiment, we took 100 samples.

All of these systems support a smaller class of constraints wrt to the ones expressible in our framework. No system can handle Customers-MC and Customers-CL. We therefore limited the comparison to Hospital-Norm-MC and Hospital-Den-CL.

Results

Each experiment was run 5 times, and the results for the best execution are reported, both in terms of quality and execution times. We pick the best result, instead of the average, in order to favor SAMPLING, which is based on a sampling of the possible repairs and has no guarantee that the best repair is computed first.

Whenever an algorithm returned more than one repair for a database, we calculated the quality metrics for each repair; in the graphs, we report the maximum, minimum, and average values. We do not report values for the LLUNATIC-FR-S50 cost manager, since they differ for less than one percentage point from those of LLUNATIC-FR-S10.

Type-1 Experiment: Customers-DE We start by showing the scalability of our chase engine in Figure 12.2.*d*. We compare the performance of LLUNATIC to the data exchange chase engine DEMO on scenario Customers-DE. It can be seen that our implementation is orders of magnitude faster than DEMO.

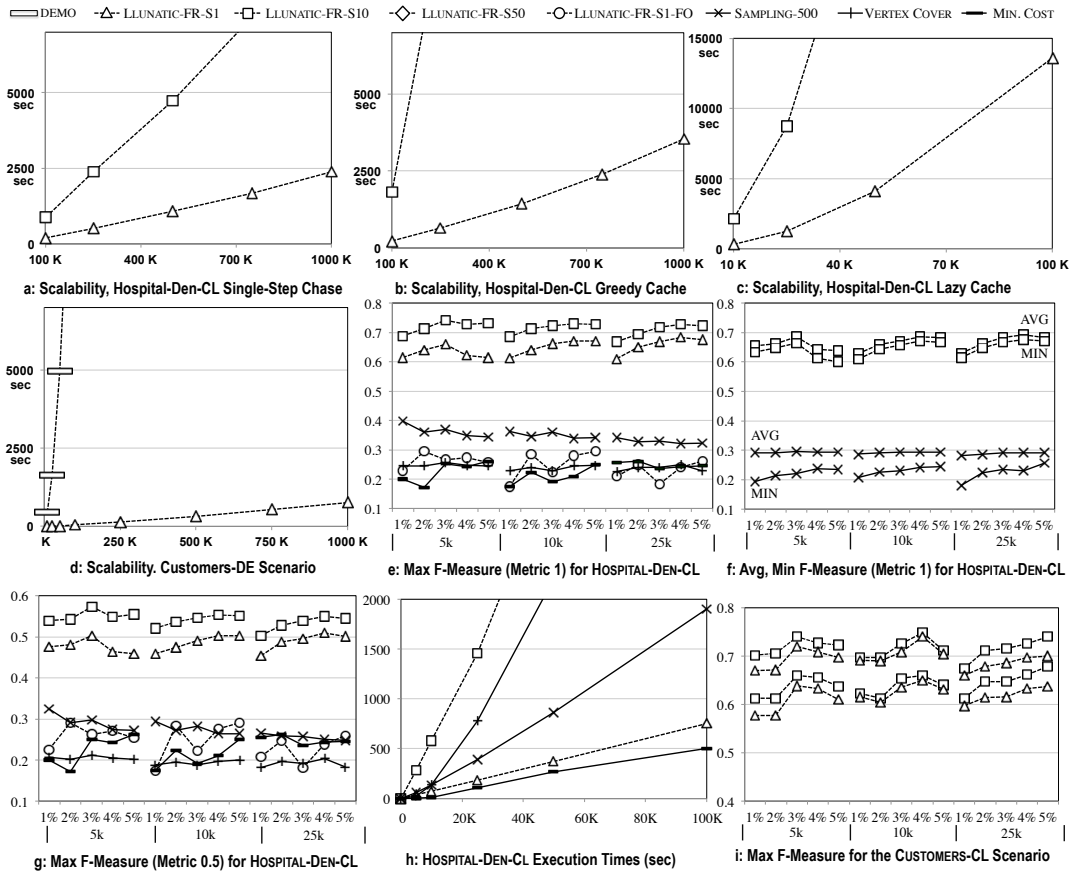


Figure 12.2: Experimental results for HOSPITAL and CUSTOMERS Type-1 and Type-2.

Type-2 Experiment: Hospital-Den-CL We report in Figures 12.2.*a–c* scalability results for some of our cost managers and the caching strategies discussed in Section 11 (single step, greedy, lazy). The charts confirm that, due to the locality of the chase algorithm, the single-step cache represents the best choice in terms of performance. Further experiments were performed with a single-step cache manager.

Figures 12.2.*a*–*d* clearly show the benefits that come with a DBMS implementation wrt main-memory ones, namely the possibility of scaling up to very large databases. While previous works [BFFR05, BIG10] have reported results up to a few thousand tuples, we were able to investigate the performance of the system on databases of millions of tuples. The figures show that LLUNATIC scales in both Type-1 and Type-2 scenarios to large databases. For HospitalDen we replicated the original dataset ten times with 1% errors. In these cases, execution times in the order of an hour for millions of tuples can be considered as a remarkable result, since no system had been able to achieve them before on problems of such exponential complexity.

The trade-offs between quality and scalability are shown in Figures 12.2.*e*–*g*.

We start by showing that LLUNATIC produces repairs of significantly higher quality with respect to those produced by previous algorithms. We ran LLUNATIC with the cost managers listed above, and the three competing algorithms on samples of the HOSPITAL dataset with increasing size (5k to 25k tuples) and increasing percentage of errors (1% to 5%).

The maximum F-measure for Metric 1 is in Figure 12.2.*e*; for the two algorithms that return more than one solution, the minimum and average F-measures are reported in Figure 12.2.*f*. The maximum F-measure for Metric 0.5 is in Figure 12.2.*g*. Quality results for algorithms MIN. COST, VERTEX COVER, and SAMPLING are consistent with those reported in [BIG10], which also conducted a comparison of these three algorithms on scenarios in which forward and backward repairs were necessary.

It is not surprising that the F-measure in these cases is quite low. Consider, in fact, a relation $R(A, B)$ with FD $A \rightarrow B$ and a tuple $R(a, 1)$; suppose the first cell is changed to introduce an error, so that the tuple becomes $R(x, 1)$. There are many cases in which this error is not fixed by repairing algorithms. This happens, in fact, whenever the new tuple, $R(x, 1)$, does not get involved in any conflict, and therefore the error goes undetected. In addition, even if a violation is raised, an algorithm may choose to repair it forward, thus missing the correct repair. Finally, even when a backward repair is correctly identified, algorithms have no clue about the right value for the A attribute, and may do little more than introducing a variable – a llun in our case – to fix the violation. All of these cases contribute to lower precision and recall.

The superior quality achieved by LLUNATIC variants can be explained by first noticing that algorithms capable of repairing both forward and backward obtained better results than those that only perform forward repairs. Besides LLUNATIC, the only other algorithm capable of backward repairs is SAMPLING. However, this algorithm picks up repairs in a random way. On the contrary,

LLUNATIC's chase algorithm explores the space of solutions in a more systematic way, and this explains its improvements in quality. In light of this, the superior quality achieved by the LLUNATIC variants, which clearly outperformed the competitors, is a significant improvement.

Figure 12.2.*h* compares execution times for the various algorithms on Hospital-Den dataset up to 100K tuples, with 1% perturbation. Recall that LLUNATIC is the first DBMS-based implementation of a data repairing algorithm. Therefore, our implementation is somehow disfavored in this comparison. To see this, consider that, when producing repairs, main-memory algorithms may aggressively use hash-based data structures to speed-up the computation of repairs, at the cost of using more memory. Using the DBMS, our algorithm is constrained to use SQL for accessing and repairing data; to see how this changes the cost of a repair, consider that even updating a single cell (a very quick operation when performed in main memory) when using the DBMS requires to perform an UPDATE, and therefore a SELECT to locate the right tuple.

Nevertheless, the LLUNATIC-FR-S1 cost manager scales nicely and has better performances than some of the main memory implementations. We may therefore say that graphs *e*–*h* in Figure 12.2 give us a concrete perception of the trade-offs between complexity and accuracy, and allow us to say that the LLUNATIC-FR-S1 is the best compromise for the HOSPITAL scenario. Other algorithms do not allow to fine tune this trade-off. To see an example, consider the SAMPLING algorithm: we noticed that taking 1000 samples instead of 500 doubles execution times, but it does not produce significant improvements in quality.

Type-2 Experiment: Customers-CL Figures 12.2.*i* reports quality results for the Customers-CL scenario. Recall that LLUNATIC is the first system that is able to handle such kind of scenarios with complex constraints. We notice that quality results are better than those on Hospital-Den-CL; this is a consequence of the clear user-specified preference rules.

It is interesting to report that performances were significantly better on the Customers-CL scenario w.r.t. Hospital-Den-CL . This is not surprising: as we discussed above, this database contains non redundant, normalized tables. This reflects the benefit of a constraint language that allows to express inter-table cleaning constraints.

Type-3 Experiment: Customers-MC The overall scalability of the chase is confirmed on scenario Customers-MC in Figure 12.3.*a*. In fact, the normalized nature of the data guarantees performance results that are significantly bet-

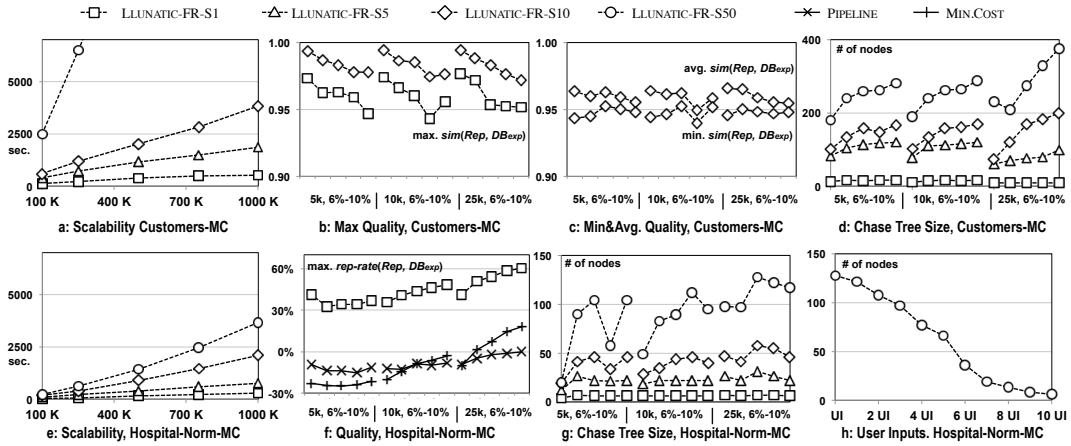


Figure 12.3: Experimental results for HOSPITAL and CUSTOMERS Type-3.

ter than those reported for the denormalized scenario in Hospital-Den-CL, even though in this case we are chasing tgds and egds together.

The execution times achieved by the algorithm can be considered as a remarkable result for problems of this complexity. They are even more surprising if we consider the size of the chase trees that our algorithm computes, which may reach several hundreds of nodes as reported in Figure 12.3.*d*. Consider also that each node in the tree is a copy of the entire database. It is also worth noting that storing chase trees as delta databases is crucial in order to achieve such a level of scalability. Without such a representation system times would be orders of magnitude higher.

Figures 12.3.*b*–*c* report the quality achieved by the various cost managers, in terms of the similarity to the core instance, $sim(\mathsf{Upd}, DB_{exp})$. LLUNATIC is the only system capable of handling scenarios of this complexity, and therefore no baseline is available. Notice that achieving 100% quality is in some cases impossible, since the sources have been made dirty in a random way, and some conflicts are not even detected by the dependencies. However, quality of the solutions is very high. This is a consequence of the rich preference rules that come with this scenario.

Type-3 Experiment: Hospital-Norm-MC Figure 12.3.*e* confirms the excellent scalability of chasing tgds and egds on normalized databases, even with chase

trees of large size (Figure 12.3.*g*). We do not report computation times for the PIPELINE and MIN.COST algorithms since they were designed to run in main memory and do not scale to large databases.

In terms of quality, we notice that finding the right repairs for Hospital-Norm-MC is quite hard, since here we have no preference relations, and there is very little redundancy in the tables. In Figure 12.3.*f* we report metric *rep-rate*(Upd, DB_{exp}) for the three algorithms that we ran on this scenario. Two things are apparent: LLUNATIC was able to partially repair the dirty database, but the overall quality was lower than the maximum one achieved in scenario Customers-MC.

On the contrary, both the MIN.COST, and the PIPELINE somehow lowered the quality. In fact, on the one side, the MIN.COST algorithm cannot backward repair cells. The PIPELINE algorithm samples repairs in a random fashion and cannot properly handle interactions among tgds and egds. As a consequence, both algorithms manage to generate a consistent repair, but at the cost of adding many unnecessary tuples to the target to repair foreign keys, and this lowers their score.

We finish by mentioning Figure 12.3.*h*, in which we study the impact of user inputs on the chase process. We run the experiment for 25K tuples interactively, and provided random user inputs by alternating the change of a llun value with the rejection of a leaf. It can be seen that small quantities of inputs from the user may significantly prune the size of the chase tree, and therefore speed-up the computation of solutions.

Chapter 13

Related Works

There has been a host of work on both data exchange and data quality management (see [ABLM10] and [FG12] for recent surveys, respectively). From the data exchange perspective, the traditional framework allows to define and enforce target constraints useful for data quality, such as functional dependencies, but a general semantics to handle conflicting values has never been proposed before our work. From the data cleaning perspective, the existing algorithms can be applied on the materialized instance of exchanges from multiple sources, but, as we have discussed above, they fail short in modeling important information coming from the sources and the transformation itself, thus obtaining repairs that are not valid instances.

Our mapping & cleaning approach is applicable to general classes of constraints and provides an elegant notion of solution. We now discuss related proposals in more details, grouped according to topic

13.1 Data Repair

Several classes of constraints have been proposed to characterize and improve the quality of data. Most relevant to our work are the (semi-)automated repairing algorithms for these constraints [BIG10, BFFR05, CFG$^+$07, FLM$^+$10, FLM$^+$11, KL09]. These methods differ in the constraints that they admit, e.g., FDs [BIG10, BFFR05], CFDs [CFG$^+$07, KL09], inclusion dependencies [BFFR05], and editing rules [FLM$^+$10], and the underlying techniques used to improve their effectiveness and efficiency, e.g., statistical inference [CFG$^+$07], measures of the reliability of the data [BFFR05, FLM$^+$10], and user interaction [CFG$^+$07, YEN$^+$11].

	Dependency Language				Rep. Strat.		Value Preference			Solution Selection			
System	FD	CFD	ER	TGD	RHS	LHS	Conf.	Curr.	MD	Cost	Cert.	Cardin.	Sampl.
[BFFR05]	√				√		√	√		√			
[CFG+07]	√	√			√	√	√	√		√			
[KL09]	√	√			√	√				√			
[FLM+10]			√		√				√		√		
[BIG10]	√				√	√				√		√	√
[DEE+13]	√	√	√		√	√	√		√			√	
[CIP13]	√	√	√		√	√	√		√			√	
Llunatic	√	√	√	√	√	√	√	√	√	√	√	√	√
	ext. dependencies				chase proced.		partial order			cost manager			

Table 13.1: Feature Comparison for Type-2 Scenarios.

All of these methods work for a specific class of constraints only, with the exception of [CIP13, FLM+11]. A flexible data quality system was recently proposed [DEE+13] which allows user-defined procedural code for detection and cleaning. These works explore the interaction among different kinds of dependencies, but they do not have a unified formal semantics with a definition of solution, neither the generality of our partial order to model preferences. Table 13.1 summarizes the features of Llunatic with respect to some of these approaches for Type-2 scenarios.

More importantly, none of the above algorithms allow tgds between schemas, i.e., the mappings. However, some of the ingredients of our scenarios are inspired by, but different from, features of other repairing approaches: repairing based on both premise and conclusion of constraints [CFG+07, BIG10, KL09], cells [BIG10, KL09, BFFR05], groups of cells [BFFR05], partial orders and its incorporation in the chase [BKL11]. We discuss these aspects in detail next.

We do allow for forward and backward chasing. Similarly, [CFG+07, KL09, BIG10] resolve violations by changing values for attributes in both the premise and conclusion of constraints. They do, however, only support a limited class of constraints. Previous works [KL09, BIG10] have used variables in order to repair the left-hand side of dependencies. With respect to variables, our lluns are a more sophisticated tool. In our approach, the full power of lluns is achieved in conjunction with cell-groups: for each llun, the corresponding cell group provides complete provenance data for the llun, both in terms of target and source cells. Therefore, it represents an ideal support for user intervention, when the value of the llun must be resolved to some constant. In

fact, lluns and cell-groups can be seen as a novel representation system [IL84] for solutions, that stands in between of the naive tables of data exchange, and of the more expressive c-tables, trying to strike a balance between complexity and expressibility.

An approach similar to ours has been proposed in [BKL11], with respect to a different cleaning problem. The authors concentrate on scenarios with matching dependencies and matching functions, where the main goal is to merge together values based on attribute similarities, and develop a chase-based algorithm. They show that, under proper assumptions, matching functions provide a partial order over database values, and that the partial order can be lifted to database instances and repairs. A key component of their approach is the availability of matching functions that are essentially total, i.e., they are able to merge any two comparable values. In fact, the problem they deal with can be seen as an instance of the entity-resolution problem. In this thesis, we deal with the different problem of data-repairing under a large class of data-cleaning constraints, and have a more ambitious goal, i.e., to embed different forms of value preference into a general semantics for the cleaning process. Our main intuition is that the notion of a partial order is an effective way to let users specify value preferences, and to incorporate them into the semantics in a principled way. In order to do this, we have shown that reasoning on the ordering of values – as in [BKL11] – or on the ordering of single cells is not enough. On the contrary, it is necessary to devise a more sophisticated notion of a partial order for cell-groups, i.e., groups of cells that need to be repaired together and for which lineage information is maintained. Also, we do not make strong assumptions about the possibility of resolving all conflicts among values in the database, and therefore introduce lluns as a third category of values besides nulls and constants.

Finally, a major contribution of this work is the development of a DBMS-based implementation of the repairing algorithm. We devoted special care in designing our chase algorithm over equivalence-classes, a notion originally proposed in [BFFR05] for FDs, and in developing the representation system of delta databases; delta database are similar to U-relations [AJKO08], but have been adapted to better suit the needs of the data repairing process.

13.2 Inference of Accuracy, Currency and Truth Discovery

A crucial and far from trivial contribution of this thesis consists in developing a new semantics for a cleaning scenario that allows to incorporate many different

strategies to pick-up preferred values and solve conflicts, as reported in Table 13.1. We showed that our partial order allows users to model master-data, value confidence, and even currency rules. The partial order can also be used to take into account tuple provenances and tuple certainty.

Algorithms for data repairing with preference relations were introduced in [SCM12]. Their changes are based on tuple deletions, not on cell changes; and preferences are among tuples, not cell values. Also, they do not consider tgds, the main challenge dealt with in our framework.

This work is also related to prior work on truth discovery from data sources [DS13]. In fact, these methods first discover dependencies on sources, such as copy relationships, to identify reliable sources; then, they employ these statistics in a probabilistic vote counting to estimate the accuracy of tuples with inconsistent values. In contrast, we do not focus on heuristics to compute the most probable value for a lluns, but expose the llun for user consumption in order to have a supervised repair.

As discussed in Section 10, a chase procedure to infer accuracy information represented by partial orders was devised in [CFY13].

13.3 Data Exchange

Our framework can be seen as an extension of the data exchange setting [FKMP05]. Furthermore, we development a DBMS-based implementation of the chase that outperforms existing chase engines [PS09] (DEMO) and can handle a much larger class of scenarios with its support to cleaning egds. We are not aware of any prior studies on optimizations for the chase. Studies to guarantee scalability for data exchange scenarios were undertaken in [MPRB09, MMP10, MMP$^+$11], but they were based on the technique of rewriting dependencies to remove the need to chase egds.

In industrial settings, most data quality related tasks are executed with ETL tools (e.g, Talend, and Informatica PowerCenter). These systems are employed for data transformations and have low-level modules for specific data quality tasks, such as verification of addresses and phone numbers. However, these tools do not offer a declarative interface and lack a formal underpinning.

Chapter 14

Conclusions and Future Work

In this thesis we faced with the problem of translating data from different repositories using schema mappings and improving the quality of the resulting database using declarative constraints. We discussed the state-of-the-art in both these fields and we argued that schema mapping and data cleaning have been considered so far in isolation, whereas in real world scenarios they are strongly related problems.

In light of these considerations, we presented the LLUNATIC mapping and cleaning system, the first comprehensive proposal to handle schema mappings and data repairing in a uniform way. The main contributions in our work are (a) a new declarative semantics for mapping a cleaning scenarios that generalizes most of the existing approaches (b) an in-depth comparison of our semantics to previous ones (c) an optimized chase algorithm to compute solutions that incorporates user interaction (d) a working system based on a scalable DBMS chase engine.

We believe that these contributions make a significant advancement with respect to the state-of-the-art, and may bring new maturity to both schema mappings and data repairing. In particular we want to encourage other researchers to use and extend LLUNATIC in order to define their existing or future mapping and cleaning semantics.

As future work, we plan to extend the language of dependencies in order to capture denial constraints. This will require to rework the notion of cell group and the chase engine. Furthermore we will try to generalize semantics which use a black-box approach in order to detect and repair violations, such as [DEE+13].

Appendices

Proofs of the Theorems

PROPOSITION 1 *There exist sets Σ_t of non-recursive tgds, Σ_e of cleaning egds, and instances $\langle I, J\rangle$ such that procedure $\mathsf{pipeline}_{\Sigma_t \cup \Sigma_e}(\langle I, J\rangle)$ does not return solutions.*

Proof: Consider the s-t tgd $S(x, y) \rightarrow T_1(x, y)$, target tgd $T_1(x, y) \rightarrow T_2(x, y)$ and egd $T_2(x, y), T_2(x, y') \rightarrow y = y'$. Given source instance $I = \{S(1, 2), S(1, 3)\}$, enforcing the tgds gives a pre-solution $J_0 = \{T_1(1, 2), T_1(1, 3), T_2(1, 2), T_2(1, 3)\}$. This instance satisfies the tgds (in the standard sense), but not the egd. Assume the repair algorithm changes both tuples in T_2 to 3. Then, when we enforce the egd, we end up with a new instance $J_1 = \{T_1(1, 2), T_1(1, 3), T_2(1, 3)\}$ that satisfies the egd, but does not longer satisfy the target tgd. The pipelining approach thus runs indefinitely without generating a solution. □

PROPOSITION 2 *The binary relation $\preceq_\Pi$ as specified in Definition 9 is a partial order.*

*Proof:*Given $\langle I, J\rangle$, we consider the set of cells $\mathcal{C} = \mathit{cells}(I) \cup \mathit{cells}(J) \cup \mathit{new\text{-}cells}(J)$. We need to show that $\langle \mathcal{C}, \preceq_\Pi\rangle$ is a partially-ordered set. This amounts to show that $\preceq_\Pi$ is reflexive, antisymmetric, and transitive.

We denote by $\mathit{const\text{-}cells}(J), \mathit{null\text{-}cells}(J)$ the set of cells of J that has a value in CONSTS, NULLS, respectively. We regard the binary relation $\preceq_\Pi$ as a directed graph G_o with cells in $\mathcal{C}$ as vertices and a directed edge between them as specified by $\preceq_\Pi$. This graph can be represented in a compact way by the following adjacency matrix, which we call A, in which cells are grouped in blocks, numbered 1–5 (recall that cells in $\mathit{new\text{-}cells}(J)$ have different null

values, and therefore are not ordered):

A	$1.auth\text{-}cells(I)$	$2.cells(I) \setminus auth\text{-}cells(I)$	$3.const\text{-}cells(J)$	$4.null\text{-}cells(J)$	$5.new\text{-}cells(J)$
$1.auth\text{-}cells(I)$	1(=)	0	0	0	0
$2.cells(I) \setminus auth\text{-}cells(I)$	1	1(Π)	1(Π)	0	0
$3.const\text{-}cells(J)$	1	1(Π)	1(Π)	0	0
$4.null\text{-}cells(J)$	1	1	1	1(=)	0
$5.new\text{-}cells(J)$	1	1	1	0	1(=)

where: (i) 1 denotes an edge; 0 no edges; (ii) 1(=) denotes that an edge is present provided that the cells are equal; (iii) 1(Π) denotes that an edge is present if the cells are ordered according to the partial order specification, Π. We notice that the adjacency matrix above is made of the following blocks:

A	$cols$ 1.	$cols$ 2, 3.	$cols$ 4.	$cols$ 5.
$rows$ 1.	I	0	0	0
$rows$ 2, 3.	1	P	0	0
$rows$ 4.	1	1	I	0
$rows$ 5.	1	1	0	I

where: (i) I is the identity matrix; (ii) 1 is a matrix made of ones only; (iii) 0 is a block made of zeros only; (iv) P is the adjacency matrix of the partial order induced by the partial-order specification, Π, over the cells of $\langle I, J\rangle$.

It can be seen that $\preceq_\Pi$ is reflexive ($\forall c : c \preceq_\Pi c$, and so all elements on the diagonal of the matrix are equal to 1). Since the matrix is lower triangular, $\preceq_\Pi$ is also anti-symmetric.

To verify transitivity, we need to show that $\forall a, b, c : a \preceq_\Pi b, b \preceq_\Pi c$ implies that $a \preceq_\Pi c$. To show this we will show that whenever there is a path of length 2 between a and c, then there is also an edge between a to c in G_o. To find out paths of length 2, we can multiply the matrix by itself. Given the structure of matrix A, it is easy to verify that A^2 is also lower triangular, and has the following structure:

A^2	$cols$ 1.	$cols$ 2, 3.	$cols$ 4.	$cols$ 5.
$rows$ 1.	I	0	0	0
$rows$ 2, 3.	B_1	P^2	0	0
$rows$ 4.	B_2	B_4	I	0
$rows$ 5.	B_3	B_5	0	I

where $B_1 - B_5$ are blocks of non-null elements. Therefore, different elements for which there is a path of length 2 may only be found in $B_1 - B_5$ and P^2. But we know that elements in $B_1 - B_5$ are such that the corresponding elements in A are equal to 1, and therefore there is also a corresponding edge in between these nodes in G_o. This means that the transitivity property is satisfied.

Let us now turn our attention to elements in P^2. Recall that these elements encode ordering relationships associated with the partial order specification, Π. These correspond to a partial order by definition, and therefore they are also transitive.

This concludes the proof. □

PROPOSITION 3 *Relation $\preceq_\Pi$ among valid cell-groups over $\langle I, J\rangle$ as specified in Definition 12 is a partial order.*

*Proof:*Given $\langle I, J\rangle$, we consider the set of valid cell groups $\mathcal{G}$ over $\langle I, J\rangle$. We need to show that $\langle \mathcal{G}, \preceq_\Pi\rangle$ is a partially-ordered set. This amounts to show that $\preceq_\Pi$ is reflexive, antisymmetric, and transitive.

By looking at the definition, we notice that $\preceq_\Pi$ is the composition of two different relations among cell groups. The first one is the containment relationship $\subseteq$ among occurrences, justifications and backward flags. This requires that, whenever $g \preceq_\Pi g'$, then it is the case that $\mathit{occ}(g) \subseteq \mathit{occ}(g'), \mathit{just}(g) \cup \mathit{just}(g')$, and that if $\mathit{isBckw}(g)$ is true, then $\mathit{isBckw}(g')$ is also true. The latter condition ensures that $\mathit{cells}(g) \subseteq \mathit{cells}(g')$, and this is known to be a partial order.

The second one is the following relation, denoted by $\trianglelefteq$, such that $g \trianglelefteq g'$ if one of the following holds:

(a) $\mathit{val}(g) \in$ NULLS (in this case, $\mathit{val}(g')$ may be either a null, or a constant, or a llun);

(b) $\mathit{val}(g') \in$ LLUNS (in this case, $\mathit{val}(g)$ may be either a null, or a constant, or a llun);

(c) $\mathit{val}(g) \in$ CONSTS, $\mathit{val}(g') \in$ CONSTS, the two cell groups are non-strict, and $\mathit{val}(g) = \mathit{val}(g')$;

(d) $\mathit{val}(g) \in$ CONSTS, $\mathit{val}(g') \in$ CONSTS and both cell groups are strict.

Our goal is to show that the composition of the two is still a partial order. We find it useful to divide valid cell groups in several subsets:

1. *lluns*: cell groups with a llun value (either strict or non-strict);

2. *strict, const*: strict cell groups with a constant value;

3. *non-strict, const*: non-strict cell groups with a constant value;

4. *nulls*: cell groups with a null value (these may only be strict).

Similarly to Proposition 2, we regard the binary relation $\preceq_\Pi$ as a directed graph G_o with cell groups in $\mathcal{G}$ as vertices and a directed edge between them as specified by $\preceq_\Pi$. This graph can be represented in a compact way by the following adjacency matrix, which we call A:

A	1.*lluns*	2.*strict, const*	3.*non-strict, const*	4.*nulls*
1.*lluns*	$1(\subseteq)$, *rule (b)*	0	0	0
2.*strict, const*	$1(\subseteq)$, *rule (b)*	$1(\subseteq)$, *rule (d)*	0	0
3.*non-strict, const*	$1(\subseteq)$, *rule (b)*	0	$1(\subseteq,=)$, *rule (c)*	0
4.*nulls*	$1(\subseteq)$, *rules (a&b)*	$1(\subseteq)$, *rule (a)*	$1(\subseteq)$, *rule (a)*	$1(\subseteq)$, *rule (a)*

where: (i) 1 denotes an edge; 0 no edges; (ii) $1(\subseteq)$ denotes that an edge is present provided that the containment property is verified; (iii) $1(\subseteq,=)$ denotes that an edge is present if both the containment property and the value of the two cell groups are identical.

We need to verify that $\preceq_\Pi$ is reflexive, antisymmetric and transitive. It is reflexive by definition ($\forall g, g' : g = g' \rightarrow g \preceq_\Pi g'$). To prove that it is antisymmetric, we notice that whenever $g \preceq_\Pi g'$ and $g' \preceq_\Pi g$, then g and g' have exactly the same occurrences, justifications and meta-cell. In addition, we can show that they need to have exactly the same value, and therefore are identical and antisymmetry is guaranteed. In fact:

- if both g and g' are strict, they have the set of occurrences, justifications and meta-cells $\mathcal{C}$ and therefore the same value *lub-val*($\mathcal{C}$);
- if g is strict, and g' is not, we end up in a contradiction: while it may be the case that $g \preceq_\Pi g'$, the opposite cannot be true (the value of g' is a generalization of *lub-val*(g') = *lub-val*(g) and the opposite cannot be true); a similar contradiction arises in the case in which g' is strict, and g is not;
- if both are not strict, either: (i) their values are both lluns; in this case the two cell groups are identical up to renaming of lluns; or (ii) they are both equal constants; (iii) or we have a similar contradiction (the value of g' is a llun, and the value of g is a constant, and therefore it is not possible that $g' \preceq_\Pi g$);

To prove transitivity, we notice that the adjacency matrix above is made of the

following blocks:

A	*cols* 1. *lluns*	*cols* 2. *strict, const*	*cols* 3. *non-strict, const*	*cols* 4. *nulls*
rows 1. *lluns*	A_1	0	0	0
rows 2. *strict, const*	A_2	A_3	0	0
rows 3. *non-strict, const*	A_4	0	A_5	0
rows 4. *nulls*	A_6	A_7	A_8	A_9

where each A_i is the adjacency matrix that encodes the relationship in the corresponding block of A above. For example, A_1 is matrix such that for any cells groups g_i, g_j with a llun value A_1 has a 1 in element (i, j) if g_1 is contained in g_2.

We have already noticed that the containment relationship $\subseteq$ among cell groups is itself a partial order. In addition, it is easily verified that also the conjunction of the containment relation, $\subseteq$, and of relation $=$ (true if two cell groups have the same value) for cell groups with constant values is a partial order. Therefore, each A_i is the adjacency matrix of a partial order.

Consider the product of A with itself:

A^2	*cols* 1. *lluns*	*cols* 2. *strict, const*	*cols* 3. *non-strict, const*	*cols* 4. *nulls*
rows 1. *lluns*	A_1^2	0	0	0
rows 2. *strict, const*	B_1	A_3^2	0	0
rows 3. *non-strict, const*	B_2	0	A_5^2	0
rows 4. *nulls*	B_3	B_4	B_5	A_9^2

where each B_i is the sum of the products of some of the A_i:

$$
\begin{aligned}
B_1 &= A_2A_1 + A_3A_2 \\
B_2 &= A_4A_1 + A_5A_4 \\
B_3 &= A_6A_1 + A_7A_2 + A_8A_4 + A_9A_6 \\
B_4 &= A_7A_3 + A_9A_7 \\
B_5 &= A_8A_5 + A_9A_8
\end{aligned}
$$

Since A_1, A_3, A_5, A_9 are adjacency matrices of partial orders, we know they are transitive, i.e., for any element that is non null in their square product (denoting a path of length 2 in the graph), we know that the corresponding element is also non null in the original matrix respectively.

It remains to show that transitivity is also satisfied for blocks $B_1 - B_5$. We analyze these in the following. First, we consider blocks B_1, B_2, B_3 in the first columns of A^2. These blocks encode paths of length 2 between any cell group

g_1 and a cell group g_2 such that $val(g_2) \in$ LLUNS. We know that containment is transitive, and therefore it is satisfied among g_1 and g_2, and since the value of g_2 is a llun, we also know that $g_1 \preceq_{\sqcap} g_2$, so the blocks are transitive.

It remains to discuss blocks B_4, B_5, We do this in a compact way by the following tables, in which we discuss the paths they encode in terms of the different kinds of cell groups involved and their values, an draw the conclusion that these are also transitive:

B_4	A_7A_3			*or*	A_9A_7		
cell groups	g_1	g'	g_2		g_1	g'	g_2
types	*null*	*strict, const*	*strict, const*		*null*	*null*	*strict, const*
values	N_i	c_j	c_k		N_i	N_i	c_j
hence:	$g_1 \preceq_{\sqcap} g_2$				$g_1 \preceq_{\sqcap} g_2$		

B_5	A_8A_5			*or*	A_9A_8		
cell groups	g_1	g'	g_2		g_1	g'	g_2
types	*null*	*non-strict, const*	*non-strict, const*		*null*	*null*	*non-strict, const*
values	N_i	c_j	c_j		N_i	N_i	c_j
hence:	$g_1 \preceq_{\sqcap} g_2$				$g_1 \preceq_{\sqcap} g_2$		

This concludes the proof. □

PROPOSITION 4 *Relation $\preceq_{\sqcap,\mathsf{User}}$ among valid cell-groups is a partial order.*

Proof: The proof builds on the one of Proposition 3. It can be seen that relation $\preceq_{\sqcap,\mathsf{User}}$ can be represented as the following adjacency matrix A_g, where A is the matrix in the proof of Proposition 3:

A_g	$1.user\text{-}nonstr_{\mathcal{M},\langle I,J\rangle}$	$2.user\text{-}strict_{\mathcal{M},\langle I,J\rangle}$	$3.auth\text{-}nonstr_{\mathcal{M},\langle I,J\rangle}$	$4.auth\text{-}strict_{\mathcal{M},\langle I,J\rangle}$	$5.std_{\mathcal{M},\langle I,J\rangle}$
$1.user\text{-}nonstr_{\mathcal{M},\langle I,J\rangle}$	$1(\subseteq)$	0	0	0	0
$2.user\text{-}strict_{\mathcal{M},\langle I,J\rangle}$	$1(\subseteq)$	$1(\subseteq)$	0	0	0
$3.auth\text{-}nonstr_{\mathcal{M},\langle I,J\rangle}$	$1(\subseteq)$	$1(\subseteq)$	$1(\subseteq)$	0	0
$4.auth\text{-}strict_{\mathcal{M},\langle I,J\rangle}$	$1(\subseteq)$	$1(\subseteq)$	$1(\subseteq)$	$1(\subseteq)$	0
$4.std_{\mathcal{M},\langle I,J\rangle}$	$1(\subseteq)$	$1(\subseteq)$	$1(\subseteq)$	$1(\subseteq)$	A

As usual, we know that $\preceq_{\sqcap,\mathsf{User}}$ is reflexive by definition. To show that it is antisymmetric, we notice that $\preceq_{\sqcap,\mathsf{User}}$ restricted to standard cell groups – i.e., block A – is antisymmetric. Let us now concentrate on user and authoritative cell groups. Consider g, g' s.t. $g \preceq_{\sqcap,\mathsf{User}} g', g' \preceq_{\sqcap,\mathsf{User}} g$. This means that cell groups have exactly the same occurrences, justifications, and backward flag. In addition, since the matrix is lower triangular, they belong to the same subset among the ones listed above. Based on this, we can show that they have exactly the same value (up to the renaming of lluns). In fact:

- if they are both non-strict, their values are both lluns;
- if they are both strict with user inputs, consider the set of cells $\mathcal{C} = occ(g) \cup just(g) = occ(g') \cup just(g')$; if $\mathsf{User}(\mathcal{C})$ is defined, they have the same user-provided value; otherwise, their values are both lluns;
- if they are both strict with authoritative cells, the upper-bound value is again the same, since the set of cells are the same.

It remains to show that $\preceq_{\Pi,\mathsf{User}}$ is transitive. We notice that each block of this matrix is a partial order, and therefore it is fairly easy to show along the lines of the proofs of Proposition 3 that matrix A_g also encodes a transitive graph. □

THEOREM 5 *Every (core) solution of a data exchange scenario corresponds to a (minimal) solution of its associated mapping scenario, and vice versa.*

Proof: We show how mapping & cleaning scenario's relate to the standard mapping scenario's as studied in data exchange. More specifically, a mapping scenario can be regarded as a special case of mapping & cleaning scenario, as follows. Consider a mapping & cleaning scenarios $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \preceq_p\}$. Assume the following:

- $\mathcal{S}_a = \emptyset$;
- Σ_t is the set of standard s-t tgds in Σ_{st}^{de} and the set of standard target tgds in Σ_t^{de};
- Σ_e is the set of standard egds in Σ_t^{de};
- Π and User are empty;
- the set of lluns, LLUNS, is also empty, i.e., we only allow for constants and labeled nulls in instances.

Let $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{T}\rangle$ be such that $J = \emptyset$. Under these assumption, we define the associated mapping scenario $\mathcal{M} = \{\mathcal{S}, \mathcal{T}, \Sigma_t, \Sigma_e\}$.

Solutions We claim that a solution J_m of $\mathcal{M}$ for I corresponds to a solution J_{mc} of $\mathcal{MC}$ for $\langle I, \emptyset\rangle$, and vice versa.

$\boxed{\Rightarrow}$ Let J_m be a solution of $\mathcal{M}$ for I. We define Upd_m as the update consisting of the following cell groups: for each $v \in dom(J_m)$, Upd_m contains

the cell group $g_v = \langle v \to occ_v, by\ \emptyset \rangle$, where occ_v consists of all cells in J_m that have value v. Since $J = \emptyset$, its trivial modification Upd_0 is empty. Clearly, $\mathsf{Upd}_0 \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_m$. It remains to show that $\langle I, \mathsf{Upd}_m(J) \rangle$ satisfies Σ_{st} and Σ_t after upgrades. For this, it suffices to observe that the definition of satisfaction after upgrades incorporates the standard notion of satisfaction. Since, J_m is a solution and thus satisfies Σ_t and Σ_e (standard semantics), we may conclude that $\langle I, \mathsf{Upd}_m(J) \rangle$ also satisfies Σ_t and Σ_e in the standard sense, and thus also satisfies these after upgrades.

$\boxed{\Leftarrow}$ Let $J_{mc} = \mathsf{Upd}(\emptyset)$. That is, J_{mc} is the set of (new) tuples specified by Upd. Suppose that $\langle I, J_{mc} \rangle$ is a solution of $\mathcal{MC}$. We need to show that we can obtain a solution J_m of the associated mapping scenario $\mathcal{M}$. Recall that $\langle I, J_{mc} \rangle$ is a solution if $\emptyset \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$, which is trivially satisfied, and $\langle I, J_{mc} \rangle$ satisfies Σ_t and Σ_e after upgrades. It remains to show that $\langle I, J_{mc} \rangle$ satisfies Σ_t and Σ_e under the standard semantics of first-order logic.

Consider an egd $e = \forall \overline{x}(\phi(\overline{x}) \to x_i = x_j)$ in Σ_e. Since $\langle I, J_{mc} \rangle$ satisfies e after upgrades, we know that for any homomorphism h of ϕ into J_{mc}, either (i) $h(x_i) = h(x_j)$; or (ii) $g_h(x_i) \preceq_{\Pi,\mathsf{User}} g_h(x_j)$, or vice versa. Clearly, if case (i) holds then $J_{mc} \models e$ and there is nothing left to show. Suppose that case (ii) holds. In this case, it is easily verified that $g_h(x_i)$ must have empty occurrences and thus is of the form $\langle h(x_i) \to \emptyset, by\ just, \mathsf{bckw} \rangle$. Indeed, if a target cell would be part of $g_h(x_i)$'s occurrences then it is also part of $g_h(x_j)$'s occurrences and h must have assigned x_i and x_j the same value. However, since egds are defined on the target, $cells_h(x_i)$ consists of target cells only. As a consequence, $g_h(x_i)$ must contain target cells which implies that case (ii) cannot hold.

Next, consider a tgd $m : \forall \overline{x}, \overline{z}\big(\phi(\overline{x}, \overline{z}) \to \exists \overline{y} \psi(\overline{x}, \overline{y})\big)$ in Σ_t. Clearly, if $\langle I, J_{mc} \rangle$ satisfies m under the standard semantics then nothing needs to be shown. Assume that there exists a homomorphism h of ϕ into $\langle I, J_{mc} \rangle$ that cannot be extended to a homomorphism h' of ψ into $\langle I, J_{mc} \rangle$, but $\mathsf{Upd}_h^{can} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$ holds. Recall that $\mathsf{Upd}_h^{can} \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$ implies that we can map each tuple $t \in \mathsf{Upd}_h^{can}(\emptyset)$ to a tuple $t_{mc} \in J_{mc}$ by means a mapping h_{id} such that for each cell group $g \in \mathsf{Upd}_h^{can}$ there exists a cell group $g' \in \mathsf{Upd}$ such that $h_{id}(g) \preceq_{\Pi,\mathsf{User}} g'$. In view of the absence of authoritative sources, user function and LLUNS, $h_{id}(g) \preceq_{\Pi,\mathsf{User}} g'$ is equivalent, by Definition 13, to saying that the cell-containment property is satisfied, and either (a) $h_{id}(g)$ and g' have the same value (up to renaming of NULLS); (b) $h_{id}(g)$ and g' have strict values in CONSTS; or (c) $val(g')$ is more informative than $val(h_{id}(g))$. The latter implies that $val(h_{id}(g)) \in$ NULLS and $val(g') \in$ CONSTS since Π is absent and no LLUNS are available.

Consider a relational atom $T(\bar{x}_0, \bar{y}_0)$ in ψ. We will show that, de-

spite our assumption, we can extend h into a homomorphism h'' such that $h''(T(\bar{x}_0, \bar{y}_0)) \in \mathsf{Upd}(J)$, and thus $\langle I, J_{mc}\rangle$ satisfies m under the standard semantics. Let $t = (h(x_1), \ldots, h(x_k), \perp_{y_1}, \ldots, \perp_{y_\ell})$ be a tuple in $\mathsf{Upd}_h^{can}(\emptyset)$, where $\bar{x}_0 = (x_1, \ldots, x_k)$, $\bar{y}_0 = (y_1, \ldots, y_\ell)$, $h(x_i) \in$ CONSTS and $\perp_{y_i} \in$ NULLS. We know that h_{id} maps $t.tid$ to a tuple id $t_{mc}.tid$ of a tuple t_{mc} in J_{mc}. Let g_{x_i} be the cell group $\langle h(x_i) \rightarrow occ_i,\ by\ just_i, \mathsf{bckw}\rangle$ in Upd_h^{can} that contains the cell corresponding to $h(x_i)$. Consider $h_{id}(g_{x_i}) = \langle h(x_i) \rightarrow h_{id}(occ_i),\ by\ h_{id}(just_i), \mathsf{bckw}\rangle$. We know that there exists a cell group $g' = \langle w \rightarrow occ',\ by\ just', \mathsf{bckw}'\rangle$ in Upd such that $h_{id}(occ_i) \subseteq occ'$, $h_{id}(just_i) \subseteq just'$ and bckw implies bckw'. We next argue that $w = h(x_i)$. Indeed, clearly w cannot be more informative than $h(x_i)$ since the latter is a constant. This rules out case (c) given previously. Furthermore, suppose that we are in case (b). Then $h(x_i)$ and w are strict constant values and thus $h(x_i) = \textit{lub-val}(occ_i \cup just_i)$ and $w = \textit{lub-val}(occ' \cup just')$. Since the cell-containment property is satisfied, $h(x_i) = w$ since otherwise $occ' \cup just'$ contains a cell with a value distinct from $h(x_i)$ which would make g' invalid (as we have no LLUNS). Clearly, case (a) also implies that $h(x_i) = w$.

Let h^{can} be an extension of h such that $h^{can}(x) = h(x)$, $h^{can}(z) = h(z)$ and $h^{can}(y) = \perp_y$, where $\perp_y$ denotes a labeled null. Let g_{y_i} be the cell group $\langle h^{can}(y_i) \rightarrow occ_i,\ by\ \emptyset, \mathsf{bckw}\rangle$ in Upd_h^{can} that contains the cell corresponding to $h^{can}(y_i)$. We again have that there exists a cell group $g'' \in \mathsf{Upd}$ such that $g'' = \langle w'' \rightarrow occ'',\ by\ just'', \mathsf{bckw}''\rangle$ in Upd such the cell-containment property is satisfied. Since $h_{id}(occ_i) \subseteq occ''$ and $h(y_i)$ is null, w'' is either null or a constant. This implies that there exists a tuple in $\mathsf{Upd}(J)$ of the form $(h(x_1), \ldots, h(x_k), c_1, \ldots, c_\ell)$ with c_i either a labeled null or a constant. We now define $h''(x_i) = h(x_i)$ and $h''(y_i) = c_i$. Clearly, h'' is an extension of h and maps $T(\bar{x}_0, \bar{y}_0)$ into $\mathsf{Upd}(J)$. In other words, h can be extended into a homomorphism h' of ψ into $\langle I, J_{mc}\rangle$. Hence, $J_{mc} \models m$ as desired.

Minimal vs. core solutions We show that core solutions (for mappings) correspond to minimal solutions (for mapping & cleaning), and vice versa.

$\boxed{\Rightarrow}$ We first show that core solutions correspond to minimal solutions. Let J_{core} be a core solution of a mapping scenario $\mathcal{M}$ and instance I. Let Upd be a minimal update, relative to $\preceq_{p,\mathsf{User}}$, such that J_{core} corresponds to $\mathsf{Upd}(\emptyset)$ and such that $\langle I, \mathsf{Upd}(\emptyset)\rangle$ is a solution of the corresponding mapping and cleaning scenario $\mathcal{MC}$ and instance I. Note that in the previous section we showed the existence of such an update, hence we can always find such a minimal update. We show that $\langle I, \mathsf{Upd}(\emptyset)\rangle$ is a minimal solution of $\mathcal{MC}$.

Suppose that $\langle I, \mathsf{Upd}'(\emptyset)\rangle$ is a solution of $\mathcal{MC}$ and I such that $\mathsf{Upd}' \prec_{p,\mathsf{User}}$

Upd. Observe that this implies that $\mathsf{Upd}'(\emptyset)$ is a solution of $\mathcal{M}$ and I that is different from J_{core}. Clearly, if $|\mathsf{Upd}'(\emptyset)| = |J_{core}|$ then they must be isomorphic and thus only differ in a renaming of labeled nulls. This implies that we can modify Upd', by performing this relabeling, such that $\mathsf{Upd}'(\emptyset) = J_{core}$ while $\mathsf{Upd}' \prec_{p,\mathsf{User}} \mathsf{Upd}$. This contradicts, the assumption that Upd is the minimal update with this property. Hence, $|\mathsf{Upd}'(\emptyset)| > |J_{core}|$. From $\mathsf{Upd}' \prec_{p,\mathsf{User}} \mathsf{Upd}$ this implies that $\mathsf{Upd}' \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$ but not the other way around. It now readily follows that there cannot exist a homomorphism from J_{core} into $\mathsf{Upd}'(\emptyset)$, contradicting the assumption that J_{core} is a core universal solution.

$\boxed{\Leftarrow}$ For the converse, let $\langle I, \mathsf{Upd}(\emptyset)\rangle$ be a minimal solution for $\mathcal{MC}$ and $\langle I, \emptyset\rangle$. We claim that $\mathsf{Upd}(\emptyset)$ is a core solution for $\mathcal{M}$ and I. Suppose that there exists a solution J_1 of $\mathcal{M}$ and I such that $J_1 \subset \mathsf{Upd}(\emptyset)$. Clearly, $J_1 = \mathsf{Upd}'(\emptyset)$ where Upd' is the restriction of Upd to cells occurring in J_1. It is readily verified that $\langle I, \mathsf{Upd}'(\emptyset)\rangle$ is also a solution for $\mathcal{MC}$ and $\langle I, \emptyset\rangle$, and $\mathsf{Upd}' \prec_{p,\mathsf{User}} \mathsf{Upd}$. This contradicts the minimality of $\langle I, \mathsf{Upd}(\emptyset)\rangle$. Hence, minimal solutions correspond to core solutions. □

THEOREM 6 *Given a cleaning scenario $\mathcal{CS} = \{\langle \mathcal{S}, \mathcal{T}\rangle, \Sigma_e, \Pi\}$ and an input instance $\langle I, J\rangle$, there always exists a solution for $\mathcal{CS}$ and $\langle I, J\rangle$.*

Proof: Indeed, there is always a solution corresponding to the update that changes all cells of J to a single llun L, and justifies it by all cells in I, i.e., $\mathsf{Upd}_{trivial} = \langle L \rightarrow \mathit{cells}(J), \mathit{by}\ \mathit{cells}(I)\rangle$. We make the straightforward assumption that this solution is never refused by the user function, User. □

THEOREM 7 *Given two solutions $\mathsf{Upd}, \mathsf{Upd}'$ for a scenario $\mathcal{CS}$ over instance $\langle I, J\rangle$, one can check $\mathsf{Upd} \preceq_\Pi \mathsf{Upd}'$ in $O(n + kmlog(m))$ time, where n is the number of cells in J, k is the maximum number of cell groups in Upd, Upd', and m is the maximum size of a cell group in Upd, Upd'.*

Proof: Notice that in a cleaning scenario the set of target cells is fixed (there are no insertions due to tgds), and therefore we only consider identity id mappings. The crux of the proof is that every cell in J may belong to a single cell group. We may therefore use hashing to map the cell group for a cell c according to Upd, to the corresponding cell group according to Upd'. Then, a sort-scan algorithm can be used to check containment of occurrences and cardinalities. □

THEOREM 8 *Given a mapping & cleaning* $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$*, instances* I *of* $\mathcal{S} \cup \mathcal{S}_a$ *and* J *of* $\mathcal{T}$*, and oracle* User*, the chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *may not terminate after a finite number of steps. If it terminates, it generates a finite set of results, each of which is a solution for* $\mathcal{MC}$ *over* $\langle I, J\rangle$*. Even if the chase terminates, not every minimal solution is generated.*

Proof: The existence of non-terminating mapping & cleaning scenarios is an immediate consequence of the fact these are a conservative extension of mapping scenarios for which such non-terminating examples are known [FKMP05]. If the chase terminates, then by the definition of chase this means that no chase rule is applicable, which in turns means that all tgds and egds are either satisfied (in the standard sense) or are satisfied after upgrades. Mapping scenarios exists for which the chase does not compute a universal solution, although there exists one. These scenarios carry over to our setting.

For soundness recall that, according to the Definition 16, a solution for $\mathcal{MC}$ is an update Upd s.t. (i) $J \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}$ and (ii) $\langle I, \mathsf{Upd}(J)\rangle$ satisfies after upgrades $\Sigma_t \cup \Sigma_e$ under $\preceq_{\Pi,\mathsf{User}}$. Regarding the point (ii) we know, by definition, that every leaf is a valid update Upd_ℓ such that there is no dependency or user inputs applicable to $\langle I, \mathsf{Upd}_\ell(J)\rangle$. For (i) it suffices to observe that for each path $J, \mathsf{Upd}_0, \ldots, \mathsf{Upd}_\ell$ in the chase tree from the root, J to a leaf, Upd_ℓ, the following condition holds: $J \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_0(J) \preceq_{\Pi,\mathsf{User}} \ldots \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_k(J)$. This is due to the fact that going from Upd_i to Upd_{i+1} cell groups are changed in four possible ways:
(a) Upd_{i+1} is the result of a chase step for user inputs, where User applies to a cell group $g \in \mathsf{Upd}_i$, and Upd_{i+1} contains a cell group g_{User} with the same occurrences, justifications and backward flag of g, and it has a user-provided value. Clearly $g \preceq_{\Pi,\mathsf{User}} g_{\mathsf{User}}$, and since Upd_{i+1} differs only on g, $\mathsf{Upd}_i \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_{i+1}$;
(b) Upd_{i+1} is the result of a chase step for a tgd, where an extended tgd m applies to Upd_i, and Upd_{i+1} is the canonical update for m. Upd_{i+1} contains new cell groups for existential variables in m, and it changes cell groups for universal variables adding new cell occurrences only, and keeping the same value, justification and backward flag. So $\mathsf{Upd}_i \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_{i+1}$;
(c) it is the case that an Upd_{i+1} is the result of a forward chase step of an extended egd e on Upd_i. Upd_{i+1} is obtained from Upd_i merging the cell groups for variables x and x' as described in Definition 24. Since the merged cell group satisfies the cell-containment properties with both $g(x)$ and $g(x')$, and its value is strict, $\mathsf{Upd}_i \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_{i+1}$;

(d) finally it is the case that an Upd_{i+1} is the result of a backward chase step of an extended egd e on Upd_i. It is obtained by changing a cell group g_{ij} in Upd_i to another cell group g'_{ij} that has same occurrences and justifications, backward flag set, and it has a strict value. So $\mathsf{Upd}_i \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_{i+1}$; □

THEOREM 9 *Given a mapping & cleaning* $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \Pi, \mathsf{User}\}$, *instances* I *of* $\mathcal{S} \cup \mathcal{S}_a$ *and* J *of* $\mathcal{T}$, *and oracle* User, *if* Σ_t *is a set of weakly-acyclic tgds, then the chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *terminates after a finite number of steps, and each leaf in the chase tree is a solution for* $\mathcal{MC}$.

Proof: The crux of the proof stands in the conservative nature of our chase procedure wrt cell groups (a cell group created during the chase is never "broken" at subsequent steps), and in the notion of satisfaction after upgrades. We give the proof for non recursive tgds. The generalization to weakly acyclic tgds is rather straightforward.

Consider a tgd $m : \phi(\bar{x}) \rightarrow \exists \bar{y} : \psi(\bar{x}, \bar{y})$ in Σ_t. Given $\langle I, J\rangle$, we define the *premise tuples*, PREM-TUPLES$(m, \langle I, J\rangle)$ for m over $\langle I, J\rangle$ as the set of all tuple ids in $\langle I, J\rangle$ for atoms that appear in $\phi(\bar{x})$. Let us call n the size of PREM-TUPLES$(m, \langle I, J\rangle)$. It is easy to see that, when chasing $\langle I, J\rangle$ with m, m can be fired for a number of times that is bounded by a function of n. In fact, any homomorphism h for which m can be fired needs to map $\phi(\bar{x})$ into a distinct combination of tuples from PREM-TUPLES$(m, \langle I, J\rangle)$.

In addition, we notice that, whenever m fires at step k for homomorphism h using some of the tuples in PREM-TUPLES$(m, \langle I, K\rangle)$, it generates a new instance of the target, K', by a canonical update, in which (a) new tuples $h(\psi(\bar{x}, \bar{y}))$ are added to the target; (b) new cell groups relating the cells of tuples in $h(\phi(\bar{x}))$ and those of $h(\psi(\bar{x}, \bar{y}))$ are generated. It is important to note that, at subsequent chase steps – either of the tgds or of the egds – the canonical update is preserved. Therefore, m will remain satisfied after upgrades for homomorphism h, and no new tuples will be added to the target.

Since Σ_t is non-recursive, we can stratify the tgds in such a way that for each tgd m in stratum i, atoms in PREM-TUPLES$(m, \langle I, K\rangle)$ at any chase step K may come from $\langle I, J\rangle$, or come from firing tgds at strata below i.

We may therefore show that every tgd $m \in \Sigma_t$ stops firing after a finite number of steps. This is easily proven by induction on the number of strata.

Base case: tgds in the first stratum can only fire once for any homomorphism h of $\phi(\bar{x})$ into PREM-TUPLES$(m, \langle I, K\rangle)$; in fact, since the tgds are non-recursive, during the chase no other tgd adds tuples to the relations in $\phi(\bar{x})$.

Similarly for egds, that may change the cell groups, but do not add new tuples, neither break homomorphisms for which m has been already fired. Therefore, the tgd stops firing after a finite number of steps.

Recursive case: consider now a tgd m in stratum i; suppose that we are at step K of the chase, and that all tgds at strata below i have stopped firing; then, m may only fire once for each homomorphisms of $\phi(\bar{x})$ into PREM-TUPLES$(m, \langle I, K\rangle)$; however any tuple in PREM-TUPLES$(m, \langle I, K\rangle)$ was either already in PREM-TUPLES$(m, \langle I, J\rangle)$, or it was generated by a sequence of chase steps of tgds at strata that precedes i. Therefore, also m will stop firing after a finite number of steps.

For soundness, see the proof of Therorem 8. □

THEOREM 10 *Given a cleaning scenario* $\mathcal{CS} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \emptyset, \Sigma_e, \Pi, \mathsf{User}\}$ *and an instance* $\langle I, J\rangle$*, the chase of* $\langle I, J\rangle$ *with* Σ_e *(i) terminates; (ii) it generates a finite set of results, each of which is a solution for* $\mathcal{CS}$ *over* $\langle I, J\rangle$*.*

Proof: For termination, it suffices to observe that for each path $J, \mathsf{Upd}_0, \ldots, \mathsf{Upd}_k$ in the chase tree from the root, J to a leaf, Upd_k, the following condition holds $J \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_0(J) \preceq_{\Pi,\mathsf{User}} \ldots \preceq_{\Pi,\mathsf{User}} \mathsf{Upd}_k(J)$. This is due to the fact that, going from Upd_i to Upd_{i+1} by a chase step, cell groups grow monotonically (either the set of occurrences/justifications grows due to a forward step, or the value is upgraded due to a backward step). Due to our semantics, there is a topmost element (the trivial update $\mathsf{Upd}_{trivial} = \langle L \rightarrow \mathit{cells}(J), \mathit{by\ cells}(I)\rangle$). Hence, every path in the chase tree is bounded in length as, in the worst case, it reaches $\mathsf{Upd}_{trivial}$. For soundness, see the proof of Theorem 9.

□

THEOREM 11 *Given a cleaning scenario* $\mathcal{CS} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \emptyset, \Sigma_e, \Pi, \mathsf{User}\}$ *and an instance* $\langle I, J\rangle$*,* $\mathcal{CS}$ *may have at most an exponential number of solutions over* $\langle I, J\rangle$*, and each solution is computed in a number of steps that is polynomial in the size of* $\langle I, J\rangle$*.*

Proof: In general, it is readily verified that a cleaning scenario can have at most an exponential number of solutions. When considering the disjunctive chase procedure, as outlined above, one can verify that each solution is computed in a number of steps that is polynomial in the size of the data. For this, it suffices to observe that one can associate an integer-valued function f on updates such

that $f(\mathsf{Upd}) < f(\mathsf{Upd}')$ whenever $\mathsf{Upd} \rightarrow_{e,\mathcal{H}} \mathsf{Upd}'$ during the chase. Intuitively, f depends on the number of llun values and sizes of cell groups in the updates. Since both the number of lluns and size of cell groups is bounded by the input instance, we may infer that f cannot be increased further after polynomially many steps, i.e., when a solution is obtained.

In contrast, computing all solutions by means of the chase takes exponential time in the size of instance. Indeed, given the polynomial size of each branch in the chase tree, as argued above, and the fact that the branching factor is polynomially bounded by the input, the overall chase tree is exponential in size. □

THEOREM 12 *Consider the chase tree* $\mathsf{chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$*, generated by the chase of* $\mathcal{MC}$ *over* $\langle I, J\rangle$ *as defined in Section 8. If the chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *terminates, then the revised chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *also terminates. In this case, the revised chase procedure generates a chase tree* $\mathsf{revised\text{-}chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$ *such that for any node in* $\mathsf{chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$*, there is an identical node in* $\mathsf{revised\text{-}chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$*.*

Proof: The proof of the first part is very similar to the proof of Theorem 8.

We will prove by induction on the level of nodes in the chase tree that for any node Upd in $\mathsf{chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$, there is an identical node Upd^{rev} in $\mathsf{revised\text{-}chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$.

Base case: the root node in both chase trees is the empty repair $\langle I, J\rangle$;

Recursive case: suppose that the theorem is true for any node from the root to level $n-1$. Consider now a node Upd_n in $\mathsf{chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$ at level n. We can identify its father Upd_{n-1}, and since this is a node at level $n-1$, there is an identical node Upd^{rev}_{n-1} in $\mathsf{revised\text{-}chase}_{\Sigma_t,\Sigma_e,\mathsf{User}}(\langle I, J\rangle)$.

If Upd_n is the result of a chase step for a tgd or a user input we have nothing to prove, because the revised chase differs from the standard one only on the definition of a chase step for egds.

We now consider the case that Upd_n is the result of the forward or backward chasing of an egd e on $\langle I, \mathsf{Upd}_{n-1}(J)\rangle$ with homomorphism h. This means that h violates the condition for $\langle I, \mathsf{Upd}_{n-1}(J)\rangle$ to satisfy after upgrades e : $\forall \overline{x}(\phi(\overline{x}) \rightarrow x = x')$, i.e. $h(x) \neq h(x')$ and neither $g_h(x) \preceq_{\Pi,\mathsf{User}} g_h(x')$ nor $g_h(x') \preceq_{\Pi,\mathsf{User}} g_h(x)$.

Since Upd^{rev}_{n-1} is equivalent to Upd_{n-1}, the same homomorphism h of e is applicable to Upd^{rev}_{n-1}. Let's call $\mathcal{H}$ the homomorphism class for Upd^{rev}_{n-1} and e that contains h. We know that $\mathcal{H}$ generates a violation for $\mathsf{Upd}^{rev}_{n-1}(J)$ and e,

because it contains in the set of conclusion groups $\mathsf{c\text{-}groups}_{\mathcal{H}}$, the cell groups $g_h(x)$ and $g_h(x')$, that we know to be different. Recall that, by definition 31, the revised chase step over for Upd_{n-1}^{rev} and e will generate a new update for each valid repair strategy $rs_{\mathcal{H}}$ of $\mathcal{H}$. We need to prove that there exists a repair strategy that applied to Upd_{n-1}^{rev} generates an update Upd_n^{rev} identical to Upd_n. We need to distinguish two cases:

- Upd_n is the result of a forward chase, then by definition is

$$\mathsf{Upd}_n = \mathsf{Upd}_{n-1} - \{g_h(x), g_h(x')\} \cup merge_{\preceq_{\Pi,\mathsf{User}}}(g_h(x) \cup g_h(x')).$$

  In order to generate the same update, consider a repair strategy $rs_{\mathcal{H}}^f$ for $\mathcal{H}$ that maps any cell group in $\mathsf{c\text{-}groups}_{\mathcal{H}}$ to "unaffected", except $g_h(x)$ and $g_h(x')$ that are marked as "forward". The chase of the chase step strategy $css_f : \{e, \mathcal{H}, rs_{\mathcal{H}}^f\}$ over Upd_{n-1}^{rev} generates the following update:

$$\mathsf{Upd}_n^{rev} = \mathsf{Upd}_{n-1}^{rev} - \mathsf{forw\text{-}g}_{rs_{\mathcal{H}}^f} \cup merge_{\preceq_{\Pi,\mathsf{User}}}(\mathsf{forw\text{-}g}_{rs_{\mathcal{H}}^f})).$$

  Since Upd_{n-1} and Upd_{n-1}^{rev} are identical, and $\mathsf{forw\text{-}g}_{rs_{\mathcal{H}}^f} = \{g_h(x), g_h(x')\}$, the resulting update Upd_n^{rev} is identical to Upd_n.

- Upd_n is the result of a backward chase, then it is defined as

$$\mathsf{Upd}_n = \mathsf{Upd}_{n-1} - \{g_{ij}\} \cup \{g'_{ij}\}$$

  where g_{ij} is the cell group of the cell $c_j \in \mathit{cells}_h(x_i)$ in Upd_{n-1}, and x_i is a witness variable in e. We know, by definition 25, that $\mathit{val}(g_{ij}) \in \textsc{consts}$ and $\mathit{auth\text{-}cells}(g_{ij}) = \emptyset$.

  Consider now the repair strategy $rs_{\mathcal{H}}^b$ for $\mathcal{H}$ that maps any cell group in $\mathsf{c\text{-}groups}_{\mathcal{H}}$ to "unaffected", except $g_h(x)$ that is marked as "backward". Assume also that this repair strategy choose, for each target cell $c_i \in g_h(x)$, to backward repair the cell c_j. Since both c_i and c_j are covered by the same homomorphism h, and the cell group of c_j according to Upd_{n-1}^{rev} has a constant value and empty justification (recall that Upd_n^{rev} and Upd_n are identical), this is a valid repair strategy for Upd_{n-1}^{rev}. Now it is easy to verify that the chase of the chase step strategy $css_b : \{e, \mathcal{H}, rs_{\mathcal{H}}^b\}$ over Upd_{n-1}^{rev} generates a repair Upd_n^{rev} identical to Upd_n.

□

Details on Examples and Experiments

Updates for Solutions in Figure 1.2

Consider our Example 1. Following is two updates $\mathsf{Upd}_1, \mathsf{Upd}_2$, that represent the solutions shown in Figure 1.2.

$$
\begin{aligned}
&\mathsf{Upd}_1\{g_1 : \langle \mathit{SF} \rightarrow \{t_4.\mathsf{City}_{[\mathit{NY}]}\},\ \mathit{by}\ \{t_{e_8}.\mathsf{City}^{\mathit{auth}}_{[\mathit{SF}]}\}\rangle,\\
&\ g_2 : \langle \mathit{F.\ Lennon} \rightarrow \{t_5.\mathsf{Name}_{[\mathit{L.\ Lennon}]}\},\ \mathit{by}\ \{t_m.\mathsf{Name}^{\mathit{auth}}_{[\mathit{F.\ Lennon}]}\}\rangle,\\
&\ g_3 : \langle \mathit{122\text{-}1876} \rightarrow \{t_5.\mathsf{Ph}_{[\mathit{122\text{-}1876}]}, t_6.\mathsf{Ph}_{[\mathit{000\text{-}0000}]}\},\ \mathit{by}\ \emptyset\}\rangle,\\
&\ g_4 : \langle \mathit{Sky\ Dr.} \rightarrow \{t_5.\mathsf{Str}_{[\mathit{null}]}, t_6.\mathsf{Str}_{[\mathit{Fry\ Dr.}]}\},\ \mathit{by}\ \{\{t_m.\mathsf{Str}^{\mathit{auth}}_{[\mathit{Sky\ Dr.}]}\}\rangle,\\
&\ g_5 : \langle L_0 \rightarrow \{t_5.\mathsf{CC\#}_{[\mathit{781658}]}, t_6.\mathsf{CC\#}_{[\mathit{784659}]}\},\ \mathit{by}\ \emptyset\rangle,\\
&\ g_6 : \langle \mathit{F.\ Lennon} \rightarrow \{t_6.\mathsf{Name}_{[\mathit{L.\ Lennon}]}\},\ \mathit{by}\ \{t_m.\mathsf{Name}^{\mathit{auth}}_{[\mathit{F.\ Lennon}]}\}\rangle,\\
&\ g_7 : \langle L_1 \rightarrow \{t_{10}.\mathsf{SSN}^{\mathit{new}}, t_{11}.\mathsf{SSN}^{\mathit{new}}, t_{12}.\mathsf{SSN}^{\mathit{new}}, t_{13}.\mathsf{SSN}^{\mathit{new}}\},,\\
&\qquad \mathit{by}\ \{t_1.\mathsf{SSN}_{[\mathit{123}]}, t_2.\mathsf{SSN}_{[\mathit{123}]}, t_3.\mathsf{SSN}_{[\mathit{124}]}\}\rangle,\\
&\ g_8 : \langle \mathit{W.\ Smith} \rightarrow \{t_{10}.\mathsf{Name}^{\mathit{new}}\},\ \mathit{by}\ \{t_1.\mathsf{Name}_{[\mathit{W.\ Smith}]}\}\rangle,\\
&\ g_9 : \langle \mathit{3456} \rightarrow \{t_{10}.\mathsf{Ph}^{\mathit{new}}, t_{11}.\mathsf{Ph}^{\mathit{new}}\},\ \mathit{by}\ \{t_1.\mathsf{Ph}_{[\mathit{0000}]}, t_3.\mathsf{Ph}_{[\mathit{3456}]}\}\rangle,\\
&\ g_{10} : \langle \mathit{Pico\ Blvd} \rightarrow \{t_{10}.\mathsf{Str}^{\mathit{new}}\},\ \mathit{by}\ \{t_1.\mathsf{Str}_{[\mathit{Pico\ Blvd}]}\}\rangle,\\
&\ g_{11} : \langle \mathit{LA} \rightarrow \{t_{10}.\mathsf{City}^{\mathit{new}}\},\ \mathit{by}\ \{t_1.\mathsf{City}_{[\mathit{LA}]}\}\rangle,\\
&\ g_{12} : \langle \mathit{null} \rightarrow \{t_{10}.\mathsf{CC\#}^{\mathit{new}}\},\ \mathit{by}\ \emptyset\rangle,\\
&\ g_{13} : \langle \mathit{W.\ Smith} \rightarrow \{t_{11}.\mathsf{Name}^{\mathit{new}}\},\ \mathit{by}\ \{t_3.\mathsf{Name}_{[\mathit{W.\ Smith}]}\}\rangle,\\
&\ g_{14} : \langle \mathit{Pico\ Blvd} \rightarrow \{t_{11}.\mathsf{Str}^{\mathit{new}}\},\ \mathit{by}\ \{t_3.\mathsf{Str}_{[\mathit{Pico\ Blvd}]}\}\rangle,\\
&\ g_{15} : \langle \mathit{LA} \rightarrow \{t_{11}.\mathsf{City}^{\mathit{new}}\},\ \mathit{by}\ \{t_3.\mathsf{City}_{[\mathit{LA}]}\}\rangle,\\
&\ g_{16} : \langle \mathit{null} \rightarrow \{t_{11}.\mathsf{CC\#}^{\mathit{new}}\},\ \mathit{by}\ \emptyset\rangle,\\
&\ g_{17} : \langle \mathit{25K} \rightarrow \{t_7.\mathsf{Salary}_{[\mathit{10K}]}, t_8.\mathsf{Salary}_{[\mathit{25K}]}\},\ \mathit{by}\ \emptyset\rangle,\\
&\ g_{18} : \langle \mathit{Dental} \rightarrow \{t_8.\mathsf{Treat}_{[\mathit{Cholest}]}\},\ \mathit{by}\ \{t_{e_4}.\mathsf{Treat}^{\mathit{auth}}_{[\mathit{Dental}]}\}\rangle,\\
&\ g_{19} : \langle \mathit{null} \rightarrow \{t_{12}.\mathsf{Salary}^{\mathit{new}}, t_{13}.\mathsf{Salary}^{\mathit{new}}\},\ \mathit{by}\ \emptyset\rangle,
\end{aligned}
$$

$g_{20} : \langle Med \rightarrow \{t_{12}.\mathsf{Insur}^{new}\}, by\ \{t_2.\mathsf{Insur}_{[Med]}\}\rangle,$
$g_{21} : \langle Eye\ surg. \rightarrow \{t_{12}.\mathsf{Treat}^{new}\}, by\ \{t_2.\mathsf{Treat}_{[Eye\ surg]}\}\rangle,$
$g_{22} : \langle 12/01/2013 \rightarrow \{t_{12}.\mathsf{Date}^{new}\}, by\ \{t_2.\mathsf{Date}_{[12/01/2013]}\}\rangle,$
$g_{23} : \langle Lapar \rightarrow \{t_{13}.\mathsf{Treat}^{new}\}, by\ \{t_3.\mathsf{Treat}_{[Lapar]}\}\rangle,$
$g_{24} : \langle 03/11/2013 \rightarrow \{t_{13}.\mathsf{Date}^{new}\}, by\ \{t_3.\mathsf{Date}_{[03/11/2013]}\}\rangle\}$

$\mathsf{Upd}_2\{g_{10}, g_{11}, g_{12}, g_{13}, g_{14}, g_{15}, g_{16}, g_{17}, g_{20}, g_{21}, g_{22}, g_{23}, g_{24},$
$g_{26} : \langle L_2 \rightarrow \{t_5.\mathsf{SSN}_{[222]}\}, by\ \emptyset, \mathsf{bckw}\rangle,$
$g_{27} : \langle 123 \rightarrow \{t_{10}.\mathsf{SSN}^{new}, t_{12}.\mathsf{SSN}^{new}\}, by\ \{t_1.\mathsf{SSN}_{[123]}, t_2.\mathsf{SSN}_{[123]}\}\rangle,$
$g_{28} : \langle L_5 \rightarrow \{t_{10}.\mathsf{Name}^{new}\}, by\ \{t_1.\mathsf{Name}_{[W.\ Smith]}\}, \mathsf{bckw}\rangle,$
$g_{29} : \langle 0000 \rightarrow \{t_{10}.\mathsf{Ph}^{new}\}, by\ \{t_1.\mathsf{Ph}_{[0000]}\}\rangle,$
$g_{30} : \langle 3456 \rightarrow \{t_{11}.\mathsf{Ph}^{new}\}, by\ \{t_3.\mathsf{Ph}_{[3456]}\}\rangle,$
$g_{31} : \langle 124 \rightarrow \{t_{11}.\mathsf{SSN}^{new}, t_{13}.\mathsf{SSN}^{new}\}, by\ \{t_3.\mathsf{SSN}_{[123]}\}\rangle,$
$g_{32} : \langle L_3 \rightarrow \{t_7.\mathsf{Insurance}_{[Abx]}\}, by\ \emptyset, \mathsf{bckw}\rangle,$
$g_{33} : \langle L_4 \rightarrow \{t_8.\mathsf{Insurance}_{[Abx]}\}, by\ \emptyset, \mathsf{bckw}\rangle,$
$g_{34} : \langle null \rightarrow \{t_{12}.\mathsf{Salary}^{new}\}, by\ \emptyset\rangle,$
$g_{35} : \langle null \rightarrow \{t_{13}.\mathsf{Salary}^{new}\}, by\ \emptyset\rangle$
$\}$

Experimental Settings

We consider three scenarios, of different nature and sizes. The first two are based on real data from the US Department of Health & Human Services (http://www.medicare.gov/hospitalcompare/), and the third one is synthetic. For each of them we report the schemas, the s-t tgds, the target tgds, and the target cleaning egds.

Hospital-Norm The first dataset is Hospital-Norm, the normalized version of the hospital data, of which we considered 3 tables with 2 foreign keys, a total of 20 attributes, and approximately 150K tuples.

Schema:

Hosp(ProviderNumber, HospitalName, Addr1, Add2, Addr3, ZipCode, CountyName, PhoneNumber)
Zip(ZipCode, City, State)
Meas(ProviderNumber, Diagnosis, Cases, Footnote, Mid)

Target tgds:

m_{t1}. Hosp(pn, hn, a1, a2, a3, zc, cn, ph) $\rightarrow$ Zip(zc,Y_0,Y_1)
m_{t2}. Hosp(pn, hn, a1, a2, a3, zc, cn, ph) $\rightarrow$ Meas(pn,Y_0,Y_1,Y_2,Y_3)

Target cleaning egds:

e_1. $\mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{zc}, \mathsf{cn}, \mathsf{ph}), \mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{zc}', \mathsf{cn}', \mathsf{ph}') \rightarrow \mathsf{hn} = \mathsf{hn}'$
e_2. $\mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{zc}, \mathsf{cn}, \mathsf{ph}), \mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{zc}', \mathsf{cn}', \mathsf{ph}') \rightarrow \mathsf{a1} = \mathsf{a1}'$
e_3. $\mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{zc}, \mathsf{cn}, \mathsf{ph}), \mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{zc}', \mathsf{cn}', \mathsf{ph}') \rightarrow \mathsf{a2} = \mathsf{a2}'$
e_4. $\mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{zc}, \mathsf{cn}, \mathsf{ph}), \mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{zc}', \mathsf{cn}', \mathsf{ph}') \rightarrow \mathsf{a3} = \mathsf{a3}'$
e_5. $\mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{zc}, \mathsf{cn}, \mathsf{ph}), \mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{zc}', \mathsf{cn}', \mathsf{ph}') \rightarrow \mathsf{zc} = \mathsf{zc}'$
e_6. $\mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{zc}, \mathsf{cn}, \mathsf{ph}), \mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{zc}', \mathsf{cn}', \mathsf{ph}') \rightarrow \mathsf{cn} = \mathsf{cn}'$
e_7. $\mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{zc}, \mathsf{cn}, \mathsf{ph}), \mathsf{Hosp}(\mathbf{pn}, \mathsf{hn}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{zc}', \mathsf{cn}', \mathsf{ph}') \rightarrow \mathsf{ph} = \mathsf{ph}'$
e_8. $\mathsf{Zip}(\mathbf{zc}, \mathsf{ci}, \mathsf{st}), \mathsf{Zip}(\mathbf{zc}, \mathsf{ci}', \mathsf{st}') \rightarrow \mathsf{st} = \mathsf{st}'$
e_9. $\mathsf{Meas}(\mathsf{pn}, \mathsf{di}, \mathsf{ca}, \mathsf{fn}, \mathbf{mi}), \mathsf{Meas}(\mathsf{pn}', \mathsf{di}', \mathsf{ca}', \mathsf{fn}', \mathbf{mi}) \rightarrow \mathsf{pn} = \mathsf{pn}'$
e_{10}. $\mathsf{Meas}(\mathsf{pn}, \mathsf{di}, \mathsf{ca}, \mathsf{fn}, \mathbf{mi}), \mathsf{Meas}(\mathsf{pn}', \mathsf{di}', \mathsf{ca}', \mathsf{fn}', \mathbf{mi})\mathsf{di} = \mathsf{di}'$
e_{11}. $\mathsf{Meas}(\mathsf{pn}, \mathsf{di}, \mathsf{ca}, \mathsf{fn}, \mathbf{mi}), \mathsf{Meas}(\mathsf{pn}', \mathsf{di}', \mathsf{ca}', \mathsf{fn}', \mathbf{mi}) \rightarrow \mathsf{ca} = \mathsf{ca}'$
e_{12}. $\mathsf{Meas}(\mathsf{pn}, \mathsf{di}, \mathsf{ca}, \mathsf{fn}, \mathbf{mi}), \mathsf{Meas}(\mathsf{pn}', \mathsf{di}', \mathsf{ca}', \mathsf{fn}', \mathbf{mi}) \rightarrow \mathsf{fn} = \mathsf{fn}'$

Hospital-Den Hospital-Den is a highly denormalized version of the same data, with 100K tuples and 19 attributes. This second version has been used in data quality experiments to test algorithms that were restricted to single-table databases. For both Hospital datasets, in our scalability tests we generated instances of size up to 1M tuples by replicating the original data several times.

Schema:

$$\begin{aligned}\mathsf{Hosp}(&\mathsf{ProviderNumber}, \mathsf{HospitalName}, \mathsf{Addr1}, \mathsf{Addr2}, \mathsf{Addr3}, \mathsf{Zip},\\ &\mathsf{County}, \mathsf{Phone}, \mathsf{City}, \mathsf{State}, \mathsf{Type}, \mathsf{Owner}, \mathsf{Emergency},\\ &\mathsf{Condition}, \mathsf{MsCode}, \mathsf{MsName}, \mathsf{Score}, \mathsf{Sample}, \mathsf{StAvg})\end{aligned}$$

Target cleaning egds:

e_1. $\mathsf{Hosp}(\mathsf{p}, \mathsf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathbf{z}, \mathsf{c}, \mathsf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathsf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathsf{n}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathbf{z}, \mathsf{c}', \mathsf{h}', \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathsf{mc}', \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{i} = \mathsf{i}'$
e_2. $\mathsf{Hosp}(\mathsf{p}, \mathsf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathbf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathsf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathsf{n}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathbf{h}, \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathsf{mc}', \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{z} = \mathsf{z}'$
e_3. $\mathsf{Hosp}(\mathsf{p}, \mathsf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathbf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathsf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathsf{n}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathbf{h}, \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathsf{mc}', \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{i} = \mathsf{i}'$

e_4. $\mathsf{Hosp}(\mathsf{p}, \mathsf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathsf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathbf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathsf{n}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathsf{h}', \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathbf{mc}, \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{mn} = \mathsf{mn}'$
e_5. $\mathsf{Hosp}(\mathsf{p}, \mathsf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathsf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathbf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathsf{n}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathsf{h}', \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathbf{mc}, \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{d} = \mathsf{d}'$
e_6. $\mathsf{Hosp}(\mathbf{p}, \mathsf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathsf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathbf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathbf{p}, \mathsf{n}', \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathsf{h}', \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathbf{mc}, \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{v} = \mathsf{v}'$
e_7. $\mathsf{Hosp}(\mathsf{p}, \mathbf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathsf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathsf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathbf{n}, \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathsf{h}', \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathsf{mc}', \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{a1} = \mathsf{a1}'$
e_8. $\mathsf{Hosp}(\mathsf{p}, \mathbf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathsf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathsf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathbf{n}, \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathsf{h}', \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathsf{mc}', \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{a2} = \mathsf{a2}'$
e_9. $\mathsf{Hosp}(\mathsf{p}, \mathbf{n}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{z}, \mathsf{c}, \mathsf{h}, \mathsf{i}, \mathsf{s}, \mathsf{t}, \mathsf{o}, \mathsf{e}, \mathsf{d}, \mathsf{mc}, \mathsf{mn}, \mathsf{x}, \mathsf{l}, \mathsf{v}),$
$\mathsf{Hosp}(\mathsf{p}', \mathbf{n}, \mathsf{a1}', \mathsf{a2}', \mathsf{a3}', \mathsf{z}', \mathsf{c}', \mathsf{h}', \mathsf{i}', \mathsf{s}', \mathsf{t}', \mathsf{o}', \mathsf{e}', \mathsf{d}', \mathsf{mc}', \mathsf{mn}', \mathsf{x}', \mathsf{l}', \mathsf{v}') \rightarrow \mathsf{t} = \mathsf{t}'$

Customers Customers is the third dataset and corresponds to our running example in Figure 1.1. The source database schemas contain 3 tables, plus 1 master data table and 2 additional tables encoding constants in CFDs. The target database schema contains 2 tables. Dependencies are the ones in Section 1.

Schema:

Patients(ssn, name, phone, street, city, conf)
Surgeries(ssn, insurance, treatments, date)
MedTreatments(ssn, name, phone, street, city, insurance, treatment, date, conf)
Hospitals(ssn, name, phone, street, city)
Cste4(insurance, treatment)
Cste8(insurance, city)

Source to target tgds:

m_1. $\mathsf{Pat}(\mathsf{ssn}, \mathsf{name}, \mathsf{phn}, \mathsf{str}, \mathsf{city}, \mathsf{conf}), \mathsf{Surg}(\mathsf{ssn}, \mathsf{ins}, \mathsf{treat}, \mathsf{date})$
$\rightarrow \exists Y_1, Y_2 : \mathsf{Cust}(\mathsf{ssn}, \mathsf{name}, \mathsf{phn}, \mathsf{conf}, \mathsf{str}, \mathsf{city}, Y_1), \mathsf{Treat}(\mathsf{ssn}, Y_2, \mathsf{ins}, \mathsf{treat}, \mathsf{date})$
m_2. $\mathsf{MedTreat}(\mathsf{ssn}, \mathsf{name}, \mathsf{phn}, \mathsf{str}, \mathsf{city}, \mathsf{ins}, \mathsf{treat}, \mathsf{date}, \mathsf{conf})$
$\rightarrow \exists Y_3, Y_4 : \mathsf{Cust}(\mathsf{ssn}, \mathsf{name}, \mathsf{phn}, \mathsf{conf}, \mathsf{str}, \mathsf{city}, Y_3), \mathsf{Treat}(\mathsf{ssn}, Y_4, \mathsf{ins}, \mathsf{treat}, \mathsf{date})$

Target tgd:

m_3. $\mathsf{Treat}(\mathsf{ssn}, \mathsf{sal}, \mathsf{ins}, \mathsf{treat}, \mathsf{date}) \rightarrow \exists Y_5, Y_6, Y_7, Y_8, Y_9, Y_{10} :$
$\mathsf{Cust}(\mathsf{ssn}, Y_5, Y_6, Y_7, Y_8, Y_9, Y_{10})$

Target cleaning egds:

e_1. $\mathsf{Cust}(\mathbf{ss}, \mathbf{n}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{Cust}(\mathbf{ss}, \mathbf{n}, \mathsf{p'}, \mathsf{s'}, \mathsf{c'}, \mathsf{cc'}) \rightarrow \mathsf{p} = \mathsf{p'}$
e_2. $\mathsf{Cust}(\mathbf{ss}, \mathbf{n}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{Cust}(\mathbf{ss}, \mathbf{n}, \mathsf{p'}, \mathsf{s'}, \mathsf{c'}, \mathsf{cc'}) \rightarrow \mathsf{cc} = \mathsf{cc'}$
e_3. $\mathsf{Cust}(\mathsf{ss}, \mathbf{n}, \mathsf{p}, \mathbf{s}, \mathbf{c}, \mathsf{cc}), \mathsf{Cust}(\mathsf{ss'}, \mathbf{n}, \mathsf{p'}, \mathbf{s}, \mathbf{c}, \mathsf{cc'}) \rightarrow \mathsf{ss} = \mathsf{ss'}$
e_4. $\mathsf{Treat}(\mathsf{ssn}, \mathsf{s}, \mathbf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{Cst}_{e_4}(\mathbf{ins}, \mathsf{tr'}) \rightarrow \mathsf{tr} = \mathsf{tr}'$
e_5. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathbf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{MD}(\mathbf{ssn}, \mathsf{n'}, \mathbf{p}, \mathsf{s'}, \mathsf{c'}) \rightarrow \mathsf{n} = \mathsf{n'}$
e_6. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathbf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{MD}(\mathbf{ssn}, \mathsf{n'}, \mathbf{p}, \mathsf{s'}, \mathsf{c'}) \rightarrow s = s'$
e_7. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathbf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{MD}(\mathbf{ssn}, \mathsf{n'}, \mathbf{p}, \mathsf{s'}, \mathsf{c'}) \rightarrow c = c'$
e_8. $\mathsf{Cust}(\mathbf{ssn}, \mathsf{n}, \mathsf{p}, \mathsf{s}, \mathsf{c}, \mathsf{cc}), \mathsf{Treat}(\mathbf{ssn}, \mathsf{sal}, \mathbf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{Cst}_{e_8}(\mathbf{ins}, \mathsf{c'}) \rightarrow \mathsf{c} = \mathsf{c'}$
e_9. $\mathsf{Treat}(\mathbf{ssn}, \mathsf{s}, \mathsf{ins}, \mathsf{tr}, \mathsf{d}), \mathsf{Treat}(\mathbf{ssn}, \mathsf{s'}, \mathsf{ins'}, \mathsf{tr'}, \mathsf{d'}) \rightarrow \mathsf{s} = \mathsf{s'}$

Bibliography

[ABC99] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, pages 68–79, 1999.

[ABLM10] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Relational and XML Data Exchange*. Morgan & Claypool, 2010.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AJKO08] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and Simple Relational Processing of Uncertain Data. In *ICDE*, pages 983–992, 2008.

[Ber11] L. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool, 2011.

[BFFR05] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.

[BFM07] L. Bravo, W. Fan, and S. Ma. Extending Dependencies with Conditions. In *VLDB*, pages 243–254, 2007.

[BGMM+09] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a Generic Approach to Entity Resolution. *VLDB J.*, 18(1):255–276, 2009.

[BIG10] George Beskales, Ihab F. Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3:197–207, 2010.

[BKL11] L. Bertossi, S. Kolahi, and L. Lakshmanan. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. In *ICDT*, pages 268–279, 2011.

[BV84] C. Beeri and M.Y. Vardi. A Proof Procedure for Data Dependencies. *J. of the ACM*, 31(4):718–741, 1984.

[CFG+07] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

[CFY13] Y. Cao, W. Fan, and W. Yu. Determining the relative accuracy of attributes. In *SIGMOD*, pages 565–576, 2013.

[CIP13] X Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations into Context. In *ICDE*, 2013.

[CT06] L. Chiticariu and W. C. Tan. Debugging Schema Mappings with Routes. In *VLDB*, pages 79–90, 2006.

[DEE+13] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, pages 541–552, 2013.

[DS13] Xin Luna Dong and Divesh Srivastava. Big data integration. In *ICDE*, 2013.

[FFP10] S. Flesca, F. Furfaro, and F. Parisi. Querying and Repairing Inconsistent Numerical Databases. *TODS*, pages 1–77, 2010.

[FG12] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool, 2012.

[FGJK08] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional Functional Dependencies for Capturing Data Inconsistencies. *ACM TODS*, 33, 2008.

[FKMP05] R. Fagin, P.G. Kolaitis, R.J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[FKP05] R. Fagin, P.G. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *ACM TODS*, 30(1):174–210, 2005.

[FLM+10] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *PVLDB*, 3(1):173–184, 2010.

[FLM+11] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction Between Record Matching and Data Repairing. In *SIGMOD*, pages 469–480, 2011.

[GMPS13] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-Cleaning Framework. *PVLDB*, 6(9):625–636, 2013.

[GMPS14] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, 2014. http://db.unibas.it/projects/llunatic.

[GST11] S. Greco, F. Spezzano, and I. Trubitsyna. Stratification criteria and rewriting techniques for checking chase termination. *PVLDB*, 4(11):1158–1168, 2011.

[IL84] T. Imieliński and W. Lipski. Incomplete Information in Relational Databases. *J. of the ACM*, 31(4):761–791, 1984.

[KL09] Solmaz Kolahi and Laks V. S. Lakshmanan. On Approximating Optimum Repairs for Functional Dependency Violations. In *ICDT*, 2009.

[Los09] D. Loshin. *Master Data Management.* Knowl. Integrity, Inc., 2009.

[MMP10] B. Marnette, G. Mecca, and P. Papotti. Scalable data exchange with functional dependencies. *PVLDB*, 3(1):105–116, 2010.

[MMP+11] B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. *PVLDB*, 4(12):1438–1441, 2011.

[MPR12] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings: Scalable Core Computations in Data Exchange. *Inf. Systems*, 37(7):677–711, 2012.

[MPRB09] G. Mecca, P. Papotti, S. Raunich, and M. Buoncristiano. Concise and Expressive Mappings with +SPICY. *PVLDB*, 2(2):1582–1585, 2009.

[MPRS12] G. Mecca, P. Papotti, S. Raunich, and D. Santoro. What is the IQ of your Data Transformation System? In *CIKM*, pages 872–881, 2012.

[PS09] R. Pichler and V. Savenkov. DEMo: Data Exchange Modeling Tool. *PVLDB*, 2(2):1606–1609, 2009.

[PVM+02] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.

[SCM12] S. Staworko, J. Chomicki, and J. Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012.

[tCCKT09] B. ten Cate, L. Chiticariu, P. Kolaitis, and W. C. Tan. Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries. *PVLDB*, 2(1):1006–1017, 2009.

[YEN+11] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.