# Mapping and Cleaning

Floris Geerts [1], Giansalvatore Mecca [2], Paolo Papotti [3], Donatello Santoro [2,4]

[1] *University of Antwerp – Antwerp, Belgium*
floris.geerts@ua.ac.be

[2] *Università della Basilicata – Potenza, Italy*
giansalvatore.mecca@gmail.com

[3] *Qatar Computing Research Institute (QCRI) – Doha, Qatar*
ppapotti@qf.org.qa

[4] *Università Roma Tre – Roma, Italy*
donatello.santoro@gmail.com

***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 paper, 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.**

## I. Introduction

Schema mappings have been proposed in the database literature as an enabling technology for modern data-driven applications. 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 [1], and nice and elegant theoretical foundations [2].

However, it is also well known that data often contain inconsistencies, and that dirty data incurs economic loss and erroneous decisions [3]. 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 cleaning 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, and therefore there is even more need for cleaning. Solving this problem is a challenging task, as discussed in the following example.

**Example 1:** Consider the mapping scenario shown in Figure 1 in which several different hospital-related data sources must be correlated to one another. The first repository has information about *Treatments* and *Physicians*. The second one about *MedPrescriptions*. In turn, the target database organizes data in terms of *Prescriptions* and *Doctors*. Notice how we are leaving ample freedom in the choice of sources. In fact:

$(i)$ we assume that the source databases may contain inconsistencies, and – while this is not mandatory in our approach – possibly come with an associated *confidence*. In our example, we assume that a confidence of 0.5 has been estimated for the first data source, and 0.7 for the second;

$(ii)$ besides possibly unreliable sources, we allow for the presence of *authoritative sources*; in our example, as it is common in large organizations, a *master-data* [4] table, *Hospital MD*, is available, containing a small set of curated tuples that represent a source of highly reliable and synchronized information for the target. Differently from sources 1 and 2, which will be primarily used to move data into the target and can generate conflicts, in our approach authoritative sources are used to repair the target and remove inconsistencies;

$(iii)$ the target can be modified and we do not assume the target to be empty, as it is typical in data translation.

The target also comes with a number of constraints. More specifically: $(a)$ there is an inclusion constraint of the form *Prescriptions.npi* $\subseteq$ *Doctors.npi*; $(b)$ attribute *id* is a key for table *Prescriptions*; $(c)$ table *Doctors* has two keys, namely attributes *npi* (the National Provider Identifier for doctors) and *name*. Notice that the source and target tables may contain inconsistencies. For example, Dr. R. Chase is assigned different *npi*s by the source database (1112 and 1222).

A data architect facing this scenario must therefore 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. We next discuss this in more detail.

**Step 1: Data Translation** The desired transformation can be expressed as a set of *tuple generating dependencies (tgds)* [2]: i.e., two source-to-target tgds and one target tgd to express the given inclusion constraint, as follows (as usual, universal quantifiers in front of the tgds are omitted):

$$\begin{aligned}
m_{st1}.\ & \mathit{Treat}(id, pat, hos, npi), \mathit{Phys}(npi, doc, sp) \\
& \rightarrow \mathit{Presc}(id, pat, npi, 0.5), \mathit{Doc}(npi, doc, sp, hos, 0.5) \\
m_{st2}.\ & \mathit{MedPresc}(id, pat, npi, doc, sp) \\
& \rightarrow \exists Y : \mathit{Presc}(id, pat, npi, 0.7), \mathit{Doc}(npi, doc, sp, Y, 0.7) \\
m_{t1}.\ & \mathit{Presc}(id, pat, npi, cf) \rightarrow \exists Y_1, Y_2, Y_3 : \mathit{Doc}(npi, Y_1, Y_2, Y_3, cf)
\end{aligned}$$

Each tgd states a constraint over the target database. For example, tgd $m_{st2}$ says that for each tuple in the *MedPrescriptions* source table, there must be corresponding tuples in the *Prescriptions* and *Doctors* target tables; $Y$ is an existential variable representing values that are not present in the source database but must be present in the target.

SOURCE

**Source #1** – Confidence 0.5

TREATMENTS

| | ID | Patient | Hospital | NPI |
|---|---|---|---|---|
| $t_1$ | P1 | W. Smith | PPTH | 1112 |

PHYSICIANS

| | NPI | Name | Spec |
|---|---|---|---|
| $t_2$ | 1112 | R. Chase | urol |

**Source #2** – Confidence 0.7

MEDPRESCRIPTIONS

| | ID | Patient | NPI | Doctor | Spec |
|---|---|---|---|---|---|
| $t_3$ | P1 | I. Smith | 1222 | R. Chase | diag |

**Source #3** – Master Data

HOSPITALS MD

| | Doctor | Spec | Hospital |
|---|---|---|---|
| $t_m$ | G.House | diag | PPTH |

TARGET

PRESCRIPTIONS

| ID | Patient | NPI | *Conf* |
|---|---|---|---|
| *(empty)* | | | |

DOCTORS

| | NPI | Name | Spec | Hospital | *Conf* |
|---|---|---|---|---|---|
| $t_5$ | 5556 | G. House | surg | PPTH | 0.5 |
| $t_6$ | 5555 | G. House | urol | *null* | 0.1 |

**(a) Initial instances**

PRESCRIPTIONS

| | ID | Patient | NPI | *Conf* |
|---|---|---|---|---|
| $t_{10}$ | P1 | **W. Smith** | 1112 | 0.5 |
| $t_{11}$ | P1 | **I. Smith** | 1222 | 0.7 |

DOCTORS

| | NPI | Name | Spec | Hospital | *Conf* |
|---|---|---|---|---|---|
| $t_5$ | **5556** | G. House | **surg** | **PPTH** | 0.5 |
| $t_6$ | **5555** | G. House | **urol** | ***null*** | 0.1 |
| $t_{13}$ | **1112** | R. Chase | **urol** | **PPTH** | 0.5 |
| $t_{14}$ | **1222** | R. Chase | **diag** | ***null*** | 0.7 |

**(b) PreSolution for the tgds**

PRESCRIPTIONS

| | ID | Patient | NPI |
|---|---|---|---|
| $t_{11}$ | P1 | I. Smith | 1222 |

DOCTORS

| | NPI | Name | Spec | Hospital |
|---|---|---|---|---|
| $t_5$ | 5556 | G. House | diag | PPTH |
| $t_{14}$ | 1222 | R. Chase | diag | PPTH |

**Minimal Solution a**

PRESCRIPTIONS

| | ID | Patient | NPI |
|---|---|---|---|
| $t_{10}$ | $\mathbf{L_a}$ | W. Smith | 1222 |
| $t_{11}$ | P1 | I. Smith | 1222 |

DOCTORS

| | NPI | Name | Spec | Hospital |
|---|---|---|---|---|
| $t_5$ | 5556 | G. House | diag | PPTH |
| $t_{14}$ | 1222 | R. Chase | diag | PPTH |

**Minimal Solution b**

**(c) Minimal, clean solutions**

Fig. 1: A Hospital Mapping Scenario.

**Step 2: Conflict Resolution** Once the mappings have been executed – for example by translating them into an SQL script – a *pre-solution* like that in Figure 1.(b) is generated. This instance satisfies the s-t tgds and the target inclusion constraint. However, it contains inconsistencies wrt the key constraints (highlighted in bold) due to conflicts in the sources.

As it was recently noted [5], this data repairing problem can be seen as a *cleaning scenario* composed of *cleaning equality generating dependencies (egds)*. These generalize many common data quality constraint formalisms, as follows (confidence attributes are omitted):

$e_1$. $Presc(\boldsymbol{id}, p, npi), Presc(\boldsymbol{id}, p', npi') \rightarrow p = p', npi = npi'$
$e_2$. $Doc(\boldsymbol{npi}, n, sp, h), Doc(\boldsymbol{npi}, n', sp', h') \rightarrow n = n', sp = sp', h = h'$
$e_3$. $Doc(npi, \boldsymbol{n}, sp, h), Doc(npi', \boldsymbol{n}, sp', h') \rightarrow npi = npi', sp = sp', h = h'$
$e_4$. $HospMD(\boldsymbol{n}, sp, \boldsymbol{h}), Doc(npi, \boldsymbol{n}, sp', \boldsymbol{h}) \rightarrow sp = sp'$

Egd $e_4$ is a *cleaning egd* [5], an extended form of egd in which both source (the master data table) and target symbols may appear in the premise; it corresponds to an *editing rule* [6], and states how values in the *Doctors* table can be corrected based on the master-data table *Hospital MD*. Notice that repairing the pre-solution may cause a violation of the tgds and hence the mappings in Step 1 need to be applied again. □

In fact, we will show that simply pipelining existing data exchange and data repairing algorithms often does not provide solutions. This is due to the fact that mappings and quality constraints may interact in a complex way, and therefore require the development of new methods in order to be handled together. Such development is far from trivial, for a number of reasons. On the one side, the standard semantics of schema mappings [2] has been conceived with no conflicts in mind. On the other side, most of the recent data repairing [3] and conflict resolution [7] algorithms concentrate on a single inconsistent table that is dirty and needs to be repaired, and can hardly be extended to handle a complex data transformation task

**Contributions** We build on our recent work [5] and make two important and nontrivial contributions. The first one is the formalization of a new framework for mapping and cleaning that tackles the problems discussed in Example 1. The second consists in the development of a general-purpose chase engine that scales nicely to large databases. More specifically:

$(i)$ 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;

$(ii)$ the framework is a conservative extension of the well-known framework of data exchange, and of most of the 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 [3], that are very important in the management of referential integrity constraints, and for which very little work exists;

$(iii)$ we implement the new semantics for mapping and cleaning scenarios under the form of a chase algorithm. 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 [8];

$(iv)$ 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. 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 [9], [10] that were designed to run in main memory.

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. In fact, we believe that this proposal may bring new maturity to both schema mappings and data repairing.

## II. Overview

**A Quick Data Repairing Tutorial** We start by providing a quick summary of the LLUNATIC data repairing framework [5]. Let us first ignore tgds and concentrate on tuples that are already present in the target. Assume that we are given the constraints expressed by egds $e_1$–$e_4$, and need to repair the target. Consider for example egd $e_3$ that states that *name* is a key for *Doctors*. Tuples $t_5$ : *(5556, G. House, surg, PPTH)* and $t_6$ : *(5555, G. House, urol, null)* contain various violations. In data repairing terminology, these correspond to *cells* – i.e., tuple attributes – with conflicting values.

**Conflicts and Partial Orders** Our approach is to *chase* the database using the dependencies to remove the conflicts. There are various ways to chase tuples $t_5$ and $t_6$ with egd $e_3$. One obvious way is to equate the values of conflicting cells. This corresponds to chasing the egd in a *forward way*, and requires selecting a *preferred value* among the conflicting values.

Discarding the null value is an obvious choice for the *hospital* attribute. In other cases, the value to prefer is less obvious, and may differ from case to case. In fact, a flexible data repairing algorithm should allow users to easily specify their preference strategies. Consider for example attribute *Spec*. Since tuples come with a confidence value, it is reasonable in this example to choose the value for specialty with the higher confidence, *surg* in this case. An elegant way to model such preferences is to specify a *partial order* $\preceq_p$, i.e., an order of preference among values of cells in the database. Assume the partial order specified by the user gives preference to values with higher confidence, i.e., *urol* (conf. 0.1) $\preceq_p$ *surg* (conf. 0.5). Then, we may forward chase tuples $t_5, t_6$ for attribute *spec* by changing cell $t_6$.*spec* to *surg*.

**Cell Groups and Upgrades** We model this partial repair by a *cell group*. Its primary function is to express a relationship among cells that need to be repaired together. In essence, since $t_5$.*spec* and $t_6$.*spec* have been equated to satisfy egd $e_3$, we want to keep track of this during the chase, so that we do not disrupt this equality in the following. We write this new cell group $g$ that has been generated by the chase by the syntax:

$$g = \langle \mathit{surg} \rightarrow \{t_5.\mathit{spec}, t_6.\mathit{spec}\}\rangle$$

and call $\{t_5.\mathit{spec}, t_6.\mathit{spec}\}$ the *occurrences* of $g$.

Cell groups are at the core of the chase algorithm. In fact, we see databases as collections of cell groups, with the starting database corresponding to the trivial set of cell groups in which each cell is assigned its own value. Each chase step removes one or more conflicting cell groups, and generates a new cell group to remove the violation. Therefore, after our first chase step, a repair $\mathsf{Rep} = \{g\}$ of $J$ has been generated.

A crucial feature of our approach is that $\mathsf{Rep}(J)$ is an improvement over $J$, since it was obtained by changing the value of a conflicting cell to a "better" value, according to the partial order specified by the user. We therefore say that $\mathsf{Rep}(J)$ is an *upgrade* of the original dirty database $J$.

Cell groups are even more powerful than this, since they can also carry *lineage* information for changes to the database, in terms of values from authoritative sources. Consider for example egd $e_4$, that specifies how to correct the target database based on master data tuples. Recall that our current repair contains cell group $g$, that has been generated at previous chase steps. Forward chasing tuples $t_5, t_6$ with master data tuple $t_m$ and egd $e_4$ further corrects the *spec* cells to value *diag*. Since the value comes from an authoritative source, we repair the violation by a new cell group $g'$, in which cell $t_m$.*spec* is explicitly stored as a *justification*, in symbols: $g' = \langle \mathit{diag} \rightarrow \{t_5.\mathit{spec}, t_6.\mathit{spec}\}\ \mathit{by}\ \{t_m.\mathit{spec}\}\rangle$. Conditional functional dependencies [3] are handled in a similar way. Again, the new repair $\mathsf{Rep}' = \{g'\}$ is an upgrade of $\mathsf{Rep}$. In fact, its cell groups carry more justifications, and are therefore better supported by authoritative sources.

**Lluns** So far we have used various rules to solve conflicts, in part standard (constants are better than nulls, authoritative values are better than target values), in part scenario-dependent and specified through $\preceq_p$. There are, however, cases in which these rules do not suggest any clear strategy to remove a violation. Consider the *npi* attribute. Suppose that no specification has been provided as to which of the values of *npi* should be preferred. In this case, to forward chase dependency $e_3$, we need to change cells $t_{11}$.*npi* and $t_{12}$.*npi* into some unknown value that may be either 5555 or 5556, or some other preferred value that we currently do not know.

To mark this fact, we introduce a new class of symbols, called *lluns*. In essence, lluns are placeholders used to mark conflicts for which a solution will need to be provided later on, for example by asking for user inputs. In this respect, lluns can thus be regarded as the opposite of nulls since lluns carry "more information" than constants.

We may summarize by saying that forward chasing $t_5$ and $t_6$ with egds $e_3, e_4$ generates the following repair to upgrade the original target database (we omit cell group justifications when they are empty; $L_0$ is a llun):

$$\begin{aligned} \mathsf{Rep} = \{\quad & g_1 = \langle \mathit{PPTH} \rightarrow \{t_5.\mathit{hospital}, t_6.\mathit{hospital}\}\rangle \\ & g_2 = \langle \mathit{diag} \rightarrow \{t_5.\mathit{spec}, t_6.\mathit{spec}\}\ \mathit{by}\ \{t_m.\mathit{spec}\}\rangle \\ & g_3 = \langle L_0 \rightarrow \{t_5.\mathit{npi}, t_6.\mathit{npi}\}\rangle \ldots\} \end{aligned}$$

**Backward Chasing and Chase Trees** There are, however, different ways to enforce an egd. Besides equating values to satisfy the conclusion, one may think of falsifying the premise. For example, consider egd $e_3$ that states that *name* is a key, i.e., two tuples that agree on *name* cannot have different values. As an alternative to forward chasing tuples $t_5, t_6$, we may also chase them *backward*. To do this, it suffices to change the *name* attribute of either tuple to a new llun, $L_i$. This means that there are three different ways to chase $t_5, t_6$ with $e_3$: $(i)$ the forward repair discussed above; $(ii)$ two backward repairs, the first one made of cell group $g_{b1} = \langle L_{b1} \rightarrow \{t_5.\mathit{name}\}\rangle$, the second of $g_{b2} = \langle L_{b2} \rightarrow \{t_6.\mathit{name}\}\rangle$.

To be more precise, in both cases we shall change into a llun the *entire cell group* of $t_5$.*name* and $t_6$.*name*. This is in fact a key feature of the chase algorithm: it preserves cell groups in order to guarantee that each chase step actually upgrades the target wrt the previous one.

It should be clear that the chase is in fact a *parallel chase*, and generates a tree of repairs. This is due not only to the presence of forward and backward repairs, but also to the fact that dependencies may be repaired in different orders, and this may yield different results.

**Cleaning Scenarios** The ideas discussed above have been formalized under the notion of a *cleaning scenario* [5]. Intuitively, in a cleaning scenario, clean source instances of a source schema $\mathcal{S}$ (e.g., master data) are linked to dirty target instances of a target schema $\mathcal{T}$ by means of cleaning egds $\Sigma_e$; solutions to cleaning scenarios can be regarded as repairs of the target instances, and are generated by means of the chase procedure discussed above, that, in contrast to its counterpart in data exchange, never fails.

**Mapping and Cleaning** A significant limitation of cleaning scenarios is that they do not support tgds. We next provide the main intuitions behind our extension of cleaning scenarios with tgds, referred to as *mapping and cleaning scenarios*.

The first crucial intuition is that, while the chase of tgds has traditionally been seen as the insertion of tuples into the target, with some effort it is possible to model it in terms of cell groups and repairs. Consider tgd $m_{t1}$ in our example, stating the inclusion dependency *Prescr.npi* $\subseteq$ *Doc.npi* (similar arguments hold for the s-t tgds).

Assume that a new tuple $t_x$ with npi *123* has been added to the prescriptions by chasing the s-t tgds, and this violates the constraint. Then, we chase tuple $t_x$ with $m_{t1}$ as follows: $(i)$ we add a new tuple $t_y$ : *(123,* $N_1$*,* $N_2$*,* $N_3$*)* to table *Doctors*; $(ii)$ in addition, we update the current repair by adding a new cell group, to keep track that the foreign key $t_x$*.npi* and the primary key $t_y$*.npi* are related to each other: $g_x = \langle \mathit{123} \rightarrow \{t_x.\mathit{npi}, t_y.\mathit{npi}\}\rangle$. This is crucial in our semantics: in this way, we guarantee that any following step upgrades the database by maintaining this cell group, and no unnecessary regressions are introduced in the repair.

Notice also that we do not backward-chase tgds, since the only way to backward-chase a tgd like $m_{t1}$ consists in deleting tuple $t_x$, and there is general consensus that deleting tuples causes unnecessary loss of information.

**Chasing with User Inputs** User inputs are modeled in mapping & cleaning scenarios by means of an oracle function, User, that works on repairs, or, better, on the cell groups they are made of. Function User is a partial function, to model the fact that users are usually only requested to provide fragments of inputs. It may be invoked on any node of the chase tree, and may either $(i)$ change the value of a cell group, $(ii)$ refuse a cell group – and therefore the entire repair of a chase node – because it is considered as a wrong way to clean the target. In the chase, we therefore have a third kind of step, aside from those that chase egds and tgds, and these correspond to invoking function User over a node of the tree.

These extensions come with a cost. In Section V, we show that a complete revision of the notions of satisfaction of a dependency, upgrade, and ultimately of the notion of a solution is required.

## III. Schema-Mappings Background

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 and NULLS be a countably infinite set of labeled nulls, distinct from CONSTS. 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 $i \in [1, k]$. We denote by $dom(I)$ the set of constants and nulls in $I$. We assume the presence of *unique tuple identifiers* in an instance; by $t_{\mathit{tid}}$ we denote the tuple with id "*tid*" in $I$. A *cell* is a location in $I$ specified by a tuple id/attribute pair $t_{\mathit{tid}}.A_i$.

**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 $\psi(\overline{x}, \overline{y})$ 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 $\overline{x}$.

Notice that, besides these standard definitions, in this paper we will make use of *extended tgds* and *cleaning egds* over schemas $\mathcal{S}, \mathcal{T}$, where $\phi(\overline{x})$ is a conjunction of relational atoms over $\mathcal{S} \cup \mathcal{T}$, i.e., we mix source and target atoms in the premise.

We assume the standard definition [2] of a *mapping scenario*, *solution*, *universal solution*, and *core solution* as the smallest of the universal solutions for a mapping scenario $\mathcal{M}$ over instances $\langle I, J \rangle$ of $\langle \mathcal{S}, \mathcal{T} \rangle$. We also assume the standard definition of the chase of a mapping scenario $\mathcal{M}$ over $\langle I, J \rangle$.

## IV. Mapping & Cleaning Scenarios

Building on the LLUNATIC framework, we next provide an extension that supports both tgds and egds.

**Definition 1** [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 $\mathcal{MC} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \preceq_p, \text{User}\}$, where: $(i)$ $\mathcal{S} \cup \mathcal{S}_a$ is the source schema and $\mathcal{T}$ is the target schema; $\mathcal{S}_a$ denotes the set of *authoritative source tables*; $(ii)$ $\Sigma_t$ is a set of *extended tgds*, as defined in Section III; $(iii)$ $\Sigma_e$ is a set of *cleaning egds*, as defined in Section III; $(iv)$ $\preceq_p$ is a specification of a partial-order for the values of cells in the target database; $(v)$ User is a partial function over repairs used to model user inputs.

LLUNS, $\preceq_p$ and User are described in more detail next.

**LLUNS and partial orders** Let $\langle I, J \rangle$ be an instance of $\langle \mathcal{S}, \mathcal{S}_a, \mathcal{T} \rangle$. Apart from taking values from CONSTS and NULLS, target instances $J$ may take values from a third countably infinite set of values, $\text{LLUNS} = \{L_1, L_2, \ldots\}$, distinct from CONSTS and NULLS.

A preference relation $\lhd$ among values formalizes the relationship between NULLS, CONSTS and LLUNS. 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 $v_1$, in symbols $v_1 \lhd v_2$, if $v_1$ and $v_2$ are of different types, and either $(i)$ $v_1 \in \text{NULLS}$, i.e., $v_1$ is a null value; or $(ii)$ $v_2 \in \text{LLUNS}$, i.e., $v_2$ is a llun.

We also assume that values of cells from authoritative tables in $I$ are *always* preferred over those coming from $J$.

In addition, users can *declaratively* specify preferred values in CONSTS by means of a partial order $\preceq$ among the values of cells in $\langle I, J \rangle$. One easy and effective way to do this was introduced in [5]. In essence, users may specify a preference rule for each attribute of the target schema under the form of a partially-ordered set. Consider Example 1; as we discussed, it would be natural to state that, whenever two different specialties are present for the same doctor, the one with higher confidence should be preferred. To specify this, users may associate with attribute *spec* the poset $(\mathcal{D}_{\text{CONF}}, \leq)$, where $\mathcal{D}_{\text{CONF}}$ is the domain of attribute *conf*, and $\leq$ is the standard order relation over real values. In essence, we are saying that, whenever specialties $v_1$ and $v_2$ of *spec*-cells need to be ordered: $(i)$ we look up the corresponding *conf*-cells, i.e., the values of the *conf* attribute of the respective tuples; $(ii)$ we order them according to $\leq$; $(iii)$ we induce an order for the original specialty values from that.

Associated with a mapping & cleaning scenario we therefore have a partial order $\preceq_p$, that subsumes both $\lhd$, and $\preceq$, i.e., $\lhd \cup \preceq \subseteq \preceq_p$. We use lluns to ensure the completeness of $\preceq_p$. In fact, given a set of cells from a database, $\mathcal{C}$, we define the *upper bound* $val_{\mathsf{lub}}(\mathcal{C})$, as the least upper bound of their values according to $\preceq_p$, if this exists, or a new llun value.

**User Inputs** We abstract user inputs by seeing the user as an oracle. More formally, we call a *user-input function* a partial function User that takes as input a pair of instances, $\langle I, J \rangle$, and a set of cells $\mathcal{C}$ over $\langle I, J \rangle$ and returns one of the following values, denoted by $\mathsf{User}(\mathcal{C})$: $(i)$ $v$, to denote that the target cells in $\mathcal{C}$ should be repaired to value $v$; $(ii)$ $\perp$, to denote that repairing the cells in $\mathcal{C}$ would represent an incorrect modification to the database, and therefore the repair should not be be performed.

It is readily verified that Example 1 can be regarded as an instance of a mapping & cleaning scenario.

## V. 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 [2], and on known data repairing algorithms, like those in [5], [9] or [10]? As an example, let $\Sigma_t$ be a set of tgds and $\Sigma_e$ be a set of egds, and $I$ and $J$ instances of $\mathcal{S} \cup \mathcal{S}_a$ and $\mathcal{T}$, respectively. Assume that we simply pipeline the chase of tgds, $\mathsf{chase}^{de}_{\Sigma_t}$, [2], and a repair algorithm for egds, $\mathsf{repair}_{\Sigma_e}$, as reported in Figure 2.

```
pipeline$_{\Sigma_t \cup \Sigma_e}(\langle I, J \rangle)$
    $\langle I, J_{tmp} \rangle := \langle I, J \rangle$;
    while (true)
        $\langle I, J_{tmp} \rangle :=$ chase$^{de}_{\Sigma_t}(\langle I, J_{tmp} \rangle)$;
        $\langle I, J_{tmp} \rangle :=$ 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
```

Fig. 2: 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 $\Sigma_t$ of non-recursive tgds, $\Sigma_e$ 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.*

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.

In this section, we formalize the semantics of solutions of a mapping & cleaning scenario. 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 $\Sigma_t$ and $\Sigma_e$ with a revised notion of satisfaction. We first give the formal definition of a solution. Then, we introduce the main ingredients of the definition.

**Definition 2** [SOLUTION] Given a *mapping&cleaning* scenario $\mathcal{M} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \preceq_p, \mathsf{User}\}$, and an input instance $\langle I, J \rangle$, a *solution* for $\mathcal{M}$ over $\langle I, J \rangle$ is a *repair* Rep s.t.:

$(i)$ Rep *upgrades* the initial target instance $J$, in symbols $J \preceq_{p,\mathsf{User}} \mathsf{Rep}$;

$(ii)$ $\langle I, \mathsf{Rep}(J) \rangle$ *satisfies after repairs* $\Sigma_t \cup \Sigma_e$ under $\preceq_{p,\mathsf{User}}$.

We first revise the definition of a cell group to allow for tgds. Let $\langle I, J \rangle$ be an instance of $\langle \mathcal{S}, S_a, \mathcal{T} \rangle$. We denote by *auth-cells*$(I)$ and *target-cells*$(J)$ the set of cells in the authoritative tables in $I$ and in the target instance $J$, respectively. Let *new-cells*$(J)$ denote the (infinite) set of cells corresponding to tuples over $\mathcal{T}$ that are not in $J$. Intuitively, *new-cells*$(J)$ represent possible insertions in $J$.

**Definition 3** [CELL GROUP] A *cell group* $g$ over $\langle I, J \rangle$ is a triple $g = \langle v, occ(g), just(g) \rangle$, where:

$(i)$ $v = val(g)$ is a value in CONSTS $\cup$ NULLS $\cup$ LLUNS;

$(ii)$ *occ*$(g)$ is a finite set of cells in *target-cells*$(J)$ $\cup$ *new-cells*$(J)$, called the *occurrences* of $g$;

$(iii)$ *just*$(g)$ is a finite set of cells in *auth-cells*$(I)$, called the *justifications* of $g$. We denote by *cells*$(g)$ the set *occ*$(g)$ $\cup$ *just*$(g)$.

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.

In a mapping & cleaning scenario, we are only interested in cell-groups that are consistent with the partial order $\preceq_p$ and user oracle User. We say that a cell group $g = \langle v \rightarrow occ(g), by\ just(g) \rangle$ has a *valid value* if one of the following conditions holds.

$(a)$ $val(g) = \mathsf{User}(cells(g))$, for some constant $v \in$ CONSTS, i.e., the value of $g$ comes from some user input;

$(b)$ $\mathsf{User}(cells(g))$ is undefined, and $val(g) = val_{lub}(cells(g))$, i.e., the cell group takes the value of the least upper bound as defined in Section IV; in this case we say that $g$ is *strict*.

$(c)$ $\mathsf{User}(cells(g))$ is undefined, and $val(g) \in$ LLUNS, i.e., the value of $g$ is a llun value.

In the following, we assume that all cell groups are valid.

**Example 2:** Consider relation $R(A, B)$, with three dependencies: $(i)$ an FD $A \rightarrow B$, and two CFDs [3]: $(ii)$ $A[a] \rightarrow B[v_1], A[a] \rightarrow B[v_2]$; the first one states that whenever $R.A$ is equal to "$a$", $R.B$ should be equal to "$v_1$" (similarly for the second); in our approach, these are encoded by egds

$e_1 : R(x,y), c1(x,z) \rightarrow y = z$, and $e_2 : R(x,y), c2(x,z) \rightarrow y = z$; in fact, we see constants as values coming from the source database; to this end, we introduce two source tables, $c1, c2$, with a single tuple $t_{c1}$, $t_{c2}$ each, encoding the patterns in the CFDs: $t_{c1} : (A : a, B : v_1), t_{c_2} : (A : a, B : v_2)$ [5]. Notice that the two CFDs clearly contradict each other. Nevertheless, differently from previous approaches, later on we will provide a semantics for this well known example [3] by leveraging lluns first, and then user inputs. Assume $R$ contains two tuples: $t_1 : R(a,1), t_2 : R(a,2)$, and that $\preceq_p$ states that greater values of the $A$ attribute are to be preferred over smaller ones (i.e., the poset associated with *B* is its own domain with the standard $\leq$ on integers). Following is a set of cell groups:

$$\begin{array}{rc}
g_1 = \langle 1 \rightarrow \{t_1.\mathrm{B}\}, \textit{by } \emptyset\rangle & \textit{valid, strict} \\
g_1' = \langle L_1 \rightarrow \{t_1.\mathrm{B}\}, \textit{by } \emptyset\rangle & \textit{valid, non strict} \\
g_2 = \langle 2 \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, \textit{by } \emptyset\rangle & \textit{valid, strict} \\
g_2' = \langle 3 \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, \textit{by } \emptyset\rangle & \textit{non valid} \\
g_3 = \langle v_1 \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, \textit{by } \{t_{c1}.\mathrm{B}\}\rangle & \textit{valid, strict} \\
g_4 = \langle L \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, \textit{by } \{t_{c1}.\mathrm{B}, t_{c2}.\mathrm{B}\}\rangle & \textit{valid, strict} \\
g_5 = \langle k \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, \textit{by } \{t_{c1}.\mathrm{B}, t_{c2}.\mathrm{B}\}\rangle & \textit{valid, strict if} \\
 & \mathsf{User}(\textit{cells}(g_4)) = k
\end{array}$$

Cell groups are used to specify *repairs*, i.e., modifications to the target database. In our approach, this can be done in two ways: $(i)$ by changing cell values, as specified in cell groups; $(ii)$ by adding new tuples as an effect of tgds.

**Definition 4** [REPAIR] A *repair* Rep of $J$ is a set of cell groups over $\langle I, J\rangle$ such that there exists a set of tuples $\Delta J$, distinct from $J$, for which:

$(i)$ each cell occurring in Rep corresponds to a cell in $J \cup \Delta J$;

$(ii)$ each cell in $J \cup \Delta J$ occurs at most once in Rep;

$(iii)$ each cell in $\Delta J$ takes the value specified by Rep.

We denote by Rep$(J)$ the target instance obtained by changing the values in $J$ as specified by Rep and by inserting $\Delta J$.

**Example 3:** Consider an authoritative source table $R(A, B)$, and a target table $S(A, B)$, with a tgd $R(x, y) \rightarrow \exists z : S(x, z)$. Suppose $I = \{t : R(1, 2)\}$, following is a set of repairs:

$$\begin{array}{llll}
\Delta J_1 = \{t_1 : S(1, N_1)\} & \mathsf{Rep}_1 = & \{g_{11} = \langle 1 \rightarrow \{t_1.\mathrm{B}\}, \textit{by } \{t.\mathrm{A}\}\rangle \\
 & & \;\; g_{12} = \langle N_1 \rightarrow \{t_1.\mathrm{B}\}, \textit{by } \emptyset\rangle\} \\
\Delta J_2 = \{t_2 : S(1, 3)\} & \mathsf{Rep}_2 = & \{g_{21} = \langle 1 \rightarrow \{t_2.\mathrm{B}\}, \textit{by } \{t.\mathrm{A}\}\rangle \\
 & & \;\; g_{22} = \langle 3 \rightarrow \{t_2.\mathrm{B}\}, \textit{by } \emptyset\rangle\}
\end{array}$$

A repair is simply one possible way of changing the target database. Different $\Delta J$ yield completely different repairs. On the contrary, in a mapping & cleaning scenario, we are only interested in repairs that are solutions for the given scenario, i.e., that represent actual upgrades of the target and satisfy the dependencies. For example, $\{g_1\}$ and $\{g_2\}$ are repairs for Example 2, but are not solutions. Among these, of particular interest are *minimal solutions*. In Example 3 $\mathsf{Rep}_1$ is minimal, while $\mathsf{Rep}_2$ is not. We introduce these notions next.

**Partial Order over Cell Groups** In order to define when a repair Rep can be regarded as an *upgrade* to the initial target instance, we lift the partial order $\preceq_p$ from values to repairs. To achieve this, we revise the partial order on cell groups (as used in cleaning scenarios) such that it properly takes into account user input. Furthermore, since the presence of tgds possibly results in the creation of tuples that are not in the initial target instance, we then show how to compare cell groups and repairs with different tuple ids.

We denote by $\preceq_{p,\mathsf{User}}$ the partial order over cell groups. Intuitively, when comparing cell groups with $\preceq_{p,\mathsf{User}}$, we should give priority to values specified by user-inputs, then to values from authoritative sources, and then again to values taken according to the preference strategy specified by $\preceq_p$. More formally, given cell groups $g$ and $g'$ with valid values, we say that $g \preceq_{p,\mathsf{User}} g'$ iff:

$(i)$ $\textit{occ}(g) \subseteq \textit{occ}(g')$ and $\textit{just}(g) \subseteq \textit{just}(g')$, i.e., we say that a cell group $g'$ can only be preferred over a cell group $g$ according to $\preceq_p$ if a *containment property* is satisfied; if the containment property is not satisfied then these cell groups represent incomparable ways to modify a target instance. In addition, we require that one of the following holds:

$(ii.a)$ $\textit{val}(g') \in \text{CONSTS}$ and $\textit{val}(g') = \mathsf{User}(\textit{cells}(g'))$, i.e., the value of $g'$ comes from a user input, or:

$(ii.b)$ $\mathsf{User}(\textit{cells}(g))$ and $\mathsf{User}(\textit{cells}(g'))$ are undefined, and either $g$ and $g'$ are strict, or $\textit{val}(g) \lhd \textit{val}(g')$. In fact, if $g$ and $g'$ are strict, the containment property above guarantees that $g'$ carries more occurrences and/or more justifications, and therefore it is to be preferred. As an alternative, it might be that $g'$ is not strict, i.e., it is "overgeneralizing" the value of its occurrences.

Consider Example 2. Cell groups are ordered as follows:

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

It is also true that $g_1 \preceq_{p,\mathsf{User}} g_1'$, since $g_1'$ is not strict, and $L_1$ is more informative than 1 (item $ii.b$).

**Id Mappings and Upgrades** We next lift $\preceq_{p,\mathsf{User}}$ from cell groups to repairs. To accommodate for the introduction of new tuples, as possibly required by the tgds, we need to be able to compare cell groups and repairs with different tuple ids. Consider Example 3: to discover that $\mathsf{Rep}_1$ is minimal, we need to map tuple $t_1 \in \Delta J_1$ into tuple $t_2 \in \Delta J_2$. We do this using *id mappings*.

Let Rep and Rep$'$ be two repairs over $\langle I, J\rangle$. An *id mapping* $h_{id}$ from Rep to Rep$'$ maps tuple ids appearing in Rep, denoted by $\mathsf{tids}(\mathsf{Rep})$, into those appearing in Rep$'$, $\mathsf{tids}(\mathsf{Rep}')$. We denote by $h_{id}(t_i)$ the image of $t_i$. An id mapping $h_{id}$ can be extended to cells and then to cell groups. Given cell $t_i.A_j$, its *image* according to $h_{id}$ is the cell $h_{id}(t_i).A_j$. Given $g$ and $h_{id}$, the *image* of $g$ according to $h_{id}$ is the cell group: $h_{id}(g) = \langle v' \rightarrow h_{id}(\textit{occ}(g)) \textit{ by } h_{id}(\textit{just}(g))\rangle$ with $v'$ defined as follows:

$(i)$ if $g$ is strict, then $v' = \textit{val}_{lub}(h_{id}(\textit{cells}(g)))$, and we say that $h_{id}(g)$ is strict;

$(ii)$ if $\textit{val}(g) = L \in \text{LLUNS}$, then $v' = L$;

$(iii)$ if $\textit{val}(g) = \mathsf{User}(\textit{cells}(g))$, then $v' = \mathsf{User}(h_{id}(\textit{cells}(g)))$, if it is defined; otherwise, we say that $h_{id}(g)$ is *undefined*.

We rely on id mappings to revise our notion of an upgrade.

**Definition 5** [UPGRADE] Given two repairs Rep and Rep$'$ over $\langle I, J\rangle$, a partial order specification $\preceq_p$ and an oracle User of cell groups over $\langle I, J\rangle$, we say that Rep$'$ *upgrades* Rep, denoted by $\mathsf{Rep} \preceq_{p,\mathsf{User}} \mathsf{Rep}'$, if there exist an id mapping $h_{id} : \mathsf{tids}(\mathsf{Rep}) \rightarrow \mathsf{tids}(\mathsf{Rep}')$ such that for each cell group $g \in \mathsf{Rep}$ there exists a cell group $g' \in \mathsf{Rep}'$ such that $h_{id}(g)$ is defined, and $h_{id}(g) \preceq_{p,\mathsf{User}} g'$.

**Satisfaction after Repairs** It is crucial that our semantics properly handle the interaction of tgds and egds. We have had several hints that enforcing egds may disrupt the logical satisfaction of tgds, or even of other egds. To see this, consider first Example 2. Notice that, after we upgrade the database with cell group $g_5$ we write a user input, $k$ into cells $t_1.B, t_2.B$, to 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.

To give another example with tgds, consider our motivating Example 1. The s-t tgd $m_{st1}$ uses source tuples $t_1, t_2$ from source #1 to generate tuple $t_{13}$ : *(1112, R. Chase, urol, PPTH)* into the target; we call $t_{13}$ the *canonical repair* for $m_{st1}$ and $t_1, t_2$. After egds have been enforced, along the lines discussed in Section II, however, the tuple is upgraded in several ways, and becomes *(1222, R. Chase, diag, PPTH)*. Again, after the changes the target instance does not satisfy tgd $m_{st1}$ in the standard sense.

In both these cases, however, we still want to consider these repairs as solutions, since they are the result of an "improvement" of values that originally satisfied the dependencies, but were dirty. Let $\mathsf{Rep}$ be a repair over $\langle I, J\rangle$. Clearly, if $\langle I, \mathsf{Rep}(J)\rangle$ satisfies an edg or tgd in the standard semantics, nothing needs to be done. Otherwise, we revise the semantics for edgs and tgds. In order to do this, given a dependency (egd or tgd), variables $x, x'$, and an homomorphism $h$ of the premise into $\langle I, \mathsf{Rep}(J)\rangle$, we want 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 $\langle I, \mathsf{Rep}(J)\rangle$ 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 $\mathit{cells}_h(x)$.

More formally: $g_h(x) = \langle v \rightarrow \mathit{occ}, \mathit{by\ just}\rangle$, where: $(i)$ $v = h(x)$; $(ii)$ *occ* (resp. *just*) is the union of all occurrences (resp. justifications) of the cell groups in $\mathsf{Rep}$ for cells in $\mathit{cells}_h(x)$; $(iii)$ in addition, *just* contains all cells in $\mathit{cells}_h(x)$ that belong to the authoritative tables in $I$.

We can now define the notion of satisfaction after repairs.

**Definition 6** [SATISFACTION AFTER REPAIRS (EGDS)] We say that $\langle I, \mathsf{Rep}(J)\rangle$ *satisfies after repairs* $e$ wrt the partial order $\preceq_{p,\mathsf{User}}$ if, whenever there is an homomorphism $h$ of $\phi(\overline{x})$ into $\langle I, \mathsf{Rep}(J)\rangle$, then $(i)$ either the value of $h(x_i)$ and $h(x_j)$ are equal (standard semantics), or $(ii)$ it is the case that $g_h(x_i) \preceq_{p,\mathsf{User}} g_h(x_j)$ or $g_h(x_j) \preceq_{p,\mathsf{User}} g_h(x_i)$.

Consider Example 2. Our solution is $\mathsf{Rep} = \{g_5\}$ (or $g_4$, if no user inputs are provided); egd $e_1 : R(x,y), c1(x,z) \rightarrow y = z$ is indeed satisfied after repairs by $\mathsf{Rep}$; consider, in fact, homomorphism $h$ mapping $z$ to $v_1$; the cell group associated by $h$ with $z$ is $g = \langle v_1, \{\}, \{t_{c1}.\mathrm{B}\}\rangle$ (it has no occurrences since $z$ is mapped to a single source cell), and the cell group of $y$ in $\mathsf{Rep}$, $g_5$ upgrades $g$.

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

Intuitively, the canonical repair $\mathsf{Rep}_h^{can}$ represents the "standard way" to repair the tgds, defined as follows. 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 $h_{can}(\psi(\overline{x}, \overline{y}))$, each with a fresh tuple id. Then, $\mathsf{Rep}_h^{can}$ is such that:

$(i)$ $J_{can} = \mathsf{Rep}_h^{can}(J)$;

$(ii)$ $\mathsf{Rep}_h^{can}$ coincides with $\mathsf{Rep}$ when restricted to $\mathit{cells}(J)$;

$(iii)$ it contains a cell group $g_h(z)$ over $\langle I, J_{can}\rangle$ for each variable $z \in \bar{x} \cup \bar{y}$.

**Definition 7** [SATISFACTION AFTER REPAIRS (TGDS)] We say that $\langle I, \mathsf{Rep}(J)\rangle$ *satisfies after repairs* $m$ under partial order $\preceq_{p,\mathsf{User}}$ if, whenever there is an homomorphism $h$ of $\phi(\overline{x}, \overline{z})$ into $\langle I, \mathsf{Rep}(J)\rangle$, then $(i)$ either $m$ is satisfied by $\langle I, \mathsf{Rep}(J)\rangle$ in the standard sense, or $(ii)$ $\mathsf{Rep}_h^{can} \preceq_{p,\mathsf{User}} \mathsf{Rep}$.

Consider Example 1 and solution (a) in Figure 1. While tgd $m_{st1}$ is not satisfied in the standard sense – solution (a) does not contain its canonical repair – it is satisfied after repairs following the previous definition.

This concludes the formalization of the notion of a solution as given in Definition 2. We are interested in solutions that are *minimal*, i.e., they do not contain unneeded tuples into the target and upgrade the initial target instance as little as possible. To quantify minimality we leverage $\preceq_p$ to decide when one repair $\mathsf{Rep}'$ *strictly upgrades* another repair $\mathsf{Rep}$, denoted by $\mathsf{Rep} \prec_{p,\mathsf{User}} \mathsf{Rep}'$. More specifically, $\mathsf{Rep} \prec_{p,\mathsf{User}} \mathsf{Rep}'$ if:

$(i)$ $\mathsf{Rep} \preceq_{p,\mathsf{User}} \mathsf{Rep}'$, but not the other way around; or

$(ii)$ $\mathsf{Rep} \preceq_{p,\mathsf{User}} \mathsf{Rep}'$, according to id mapping $h_{id}$, $\mathsf{Rep}' \preceq_{p,\mathsf{User}} \mathsf{Rep}$, according to id mapping $h'_{id}$, and $h'_{id}$ is surjective while $h_{id}$ is not surjective.

**Definition 8** [MINIMAL SOLUTIONS] A *minimal solution* for a mapping and cleaning scenario is any solution that is minimal wrt the partial order $\prec_{p,\mathsf{User}}$.

Two minimal solutions for our motivating example are shown in Figure 1. These can be made non-minimal by adding unneeded tuples, or unnecessary changes (like, for example, changing the name of Dr. House to a llun $L_b$).

Our semantics is a conservative extension of both the one of mapping scenarios and of cleaning scenarios. In fact, a cleaning scenario $\mathcal{C}$ corresponds to a mapping & cleaning scenario in which $\mathcal{S}$, $\Sigma_t$ and $\mathsf{User}$ are absent. A mapping scenario $\mathcal{M}$ is a mapping & cleaning scenario in which $\Sigma_t$ and $\Sigma_e$ are standard dependencies, LLUNS, $\mathcal{S}_a$ and $\mathsf{User}$ are absent and $\preceq_p$ simply states that nulls are less informative than constants. In addition:

**Theorem 2:** *Every (core) solution of a mapping scenario corresponds to a (minimal) solution of its associated mapping & cleaning scenario, and vice versa. Similarly for (minimal) solutions of cleaning scenarios.*

## VI. The Chase

We compute solutions for mapping & cleaning scenarios by means of a generalized chase procedure. In essence, our chase generates a tree of repairs by three main kind of steps: $(i)$ chasing egds (forward and backward) with cell groups; $(ii)$ chasing tgds (forward only) with cell groups; $(iii)$ correcting cell groups or refuting repairs by means of $\mathsf{User}$.

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

**Chase Step for Tgds** Given an extended tgd $m : \forall \bar{x}, \bar{z}$ $(\phi(\bar{x}, \bar{z}) \rightarrow \exists \bar{y} : (\psi(\bar{x}, \bar{y})))$ in $\Sigma_t$ applicable to $\langle I, \mathsf{Rep}(J)\rangle$ with homomorphism $h$, by *chasing* $m$ on $\langle I, \mathsf{Rep}(J)\rangle$ with $h$ we generate a new repair $\mathsf{Rep}'$ obtained from $\mathsf{Rep}$, in symbols $\mathsf{Rep} \rightarrow_{m,h} \mathsf{Rep}'$, by:

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

$(ii)$ adding the new cell groups in the canonical repair $\mathsf{Rep}_h^{can}$, i.e.: $\mathsf{Rep}' = \mathsf{Rep} - \{g_h(x) | x \in \bar{x}\} \cup \mathsf{Rep}_h^{can}$.

**Chase Step for Egds** We first introduce the notions of *witness* and *witness variable* for an egd. Let $e : \forall \bar{x}\ (\phi(\bar{x}) \rightarrow x = x')$ be a cleaning egd. A *witness variable* for $e$ is a variable $x \in \bar{x}$ that has multiple occurrences in $\phi(\bar{x})$. For an homomorphism $h$ of $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J)\rangle$, 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 $e$. Given the tuples in Example 2, $\{R(1, a), R(1, b)\}$, and egd $R(x, y), R(x, y') \rightarrow y = y'$, $x$ is the witness variable, and $h(x) = 1$ is the witness.

Given a cleaning egd $e : \forall \bar{x}\ (\phi(\bar{x}) \rightarrow x = x')$ in $\Sigma_e$ applicable to $\langle I, \mathsf{Rep}(J)\rangle$ with homomorphism $h$, by *forward chasing* $e$ on $\langle I, \mathsf{Rep}(J)\rangle$ with $h$ we generate a new repair $\mathsf{Rep}_f$ obtained from $\mathsf{Rep}$ by:

$(i)$ removing $g_h(x)$ and $g_h(x')$;

$(ii)$ adding the new cell group $lub_{\preceq_{p,\mathsf{User}}}(g_h(x), g_h(x'))$, i.e.: $\mathsf{Rep}_f = \mathsf{Rep} - \{g_h(x), g_h(x')\} \cup lub_{\preceq_{p,\mathsf{User}}}(g_h(x), g_h(x'))$.

By *backward chasing* $e$ on $\langle I, \mathsf{Rep}(J)\rangle$ with $h$ we try to falsify the premise in all possible ways. To do this, we generate a number of new repairs as follows: for each witness variable $x_i \in \bar{x}_w$ of $e$, and each cell $c_j \in cells_h(x_i)$, consider the corresponding cell group according to $\mathsf{Rep}$, $g_{ij} = g_{\mathsf{Rep}}(c_j)$. If $val(g_{ij}) \in$ CONSTS and $just(g_{ij}) = \emptyset$, generate a new repair $\mathsf{Rep}_{bij}$ obtained from $\mathsf{Rep}$ by changing all cells in $g_{ij}$ to another cell group $g'_{ij}$ that is an immediate successor of $g_{ij}$ in the partial order: $\mathsf{Rep}_{b_{ij}} = \mathsf{Rep} - \{g_{ij}\} \cup \{g'_{ij}\}$

**Chase Step for User Inputs** We say that $\mathsf{User}$ applies to a group $g \in \mathsf{Rep}$ if $\mathsf{User}(cells(g))$ is defined, and returns a value that is different from $val(g)$. We say that $\mathsf{User}$ *refuses* $\mathsf{Rep}$ if $\mathsf{User}(cells(g)) = \bot$. If $\mathsf{User}$ refuses $\mathsf{Rep}$, we mark $\mathsf{Rep}$ as *invalid*. Otherwise, we denote by $\mathsf{User}(\mathsf{Rep})$ the repair obtained from $\mathsf{Rep}$ by changing any cell group $g \in \mathsf{Rep}$ such that $\mathsf{User}$ applies to $g$, to a new cell group $g_{\mathsf{User}}$ obtained from $g$ by changing $val(g)$ to $\mathsf{User}(cells(g))$.

Chase steps generate a *chase tree* whose root is $\langle I, J\rangle$, and for each valid node $\mathsf{Rep}$, the children of $\mathsf{Rep}$ are the repairs $\mathsf{Rep}_0, \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n$ generated by steps above. Any valid leaf in the tree is called a *result*.

If the chase procedure terminates, it generates solutions, although not necessarily minimal ones.

**Theorem 3:** *Given a mapping & cleaning* $\mathcal{M} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \preceq_p, \mathsf{User}\}$*, instances* $I$ *of* $\mathcal{S} \cup \mathcal{S}_a$ *and* $J$ *of* $\mathcal{T}$*, and oracle* $\mathsf{User}$*, 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* $\langle I, J\rangle$.

We can prove that, as soon as the tgds are non recursive, than 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 [11].

**Theorem 4:** *Given a mapping & cleaning* $\mathcal{M} = \{\mathcal{S}, \mathcal{S}_a, \mathcal{T}, \Sigma_t, \Sigma_e, \preceq_p, \mathsf{User}\}$*, instances* $I$ *of* $\mathcal{S} \cup \mathcal{S}_a$ *and* $J$ *of* $\mathcal{T}$*, and oracle* $\mathsf{User}$*, if* $\Sigma_t$ *is a set of weakly-acyclic tgds [2] then the chase of* $\langle I, J\rangle$ *with* $\Sigma_t, \Sigma_e, \mathsf{User}$ *terminates.*

**Cost Managers and User Managers** In terms of complexity, it is well-known [3] that a database can have an exponential number of repairs, even for a cleaning scenario with a single FD and when no backward chase steps are allowed. Given such a high level of complexity, as it is common in data repairing, we need to devise ways to prune the chase tree and discard some of the solutions in favor of others. In [5], we incorporate these pruning methods into the chase process in a principled and user-customizable way by introducing a component, called the *cost manager*. Intuitively, given a chase tree $t$, a *cost manager* is a predicate CM over the nodes for $t$. For each node $n$ in the chase tree, i.e., for each repair, it may either select it (CM$(n)$ = *true*), or discard it (CM$(n)$ = *false*).

During the chase, we shall systematically make use of the cost manager. Whenever a chase step is discarded by the cost manager, the corresponding branch is no longer pursued. The trivial cost manager is the one that keeps all solutions, and may be used for very small scenarios. Our implementation offers a rich library of cost managers. Among these, we mention the following (further details are available in [5]): $(i)$ a *forward-only* cost manager (FO): it accepts only forward repairs and discards all nodes that contain backward ones; $(ii)$ a *maximum size* cost manager (SN): it accepts new branches in the chase tree as long as the number of leaves is less than $N$; $(iii)$ a *frequency* cost manager (FR) that intuitively uses the frequency of values for an attribute to decide whether to forward or backward chase: values that are similar to the most frequent one are forward chased, the others are bacwkard chased. Notice that combinations of these strategies are possible, to obtain, e.g., a FR-S5 cost manager.

In a mapping & cleaning setting, we also plug into the chase a second component, called the *user manager*. This is a second predicate over nodes in the tree, that it is used to decide when the chase should be stopped to request for user inputs. There are several simple strategies to do this: $(i)$ a *step-by-step* user manager stops the chase and asks for inputs after each new node is added to the tree; $(ii)$ an *after-llun* user manager only stops for nodes that contain lluns; $(iii)$ a *level n* cost manager stops the chase after $n$ new levels have been added to the tree, and so on.

## VII. Scalability and Optimizations

A key goal of this paper is to develop a scalable implementation of the chase algorithm. To do this, we build on optimizations for egds introduced in [5], and in particular the notion of *delta databases*, a representation system conceived to efficiently store and process chase trees. Our goal in this paper 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 [12], 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* [13] 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. More specifically, we assign a unique id to each node in the chase tree, i.e., to each chase step. We represent each cell in the database as a triple: *(table, tupleId, attributeName)*. In addition, each cell group has a unique id. Then table *occurrences* has schema (*stepId, cellGroupId, cellGroupValue, table, tupleId, attributeName*); table *justifications* has schema (*stepId, cellGroupId, table, tupleId, attributeName, sourceValue*).

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:

$$\text{insert into } R_3 \text{ select } x, \text{sk}(x) \text{ from } R_1, R_2 \text{ where } R_1.x = R_2.x \text{ and } x \text{ not in ( select } x \text{ from } R_3).$$

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:

$$\text{select } R_1.x, R_2.x, R_3.x, R_3.y \text{ from } R_1, R_2, R_3 \text{ where } R_1.x = R_2.x \text{ and } R_2.x = R_3.x \text{ and } x \text{ in (select } x \text{ from } \textit{violations}).$$

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:

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

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.

## VIII. EXPERIMENTS

Experiments have been executed running the Java prototype of the LLUNATIC system on a Intel i7 machine with 2.6Ghz processor and 8GB of RAM under MacOS. The DBMS was PostgreSQL 9.2.

**Datasets.** 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/), and the third one is synthetic.

$(a)$ 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.

$(c)$ Doctors, corresponds to our running example in Figure 1.

**Errors.** In order to test our algorithms with different levels of noise, we introduced errors in the datasets using a random noise generator. For each datasets, we generated copies with a number of noisy cells ranging from 5% to 10% of the total.

**Scenarios.** Based on these datasets, we generated 4 different scenarios. For each scenario we also fixed an expected solution, called $DB_{exp}$, as follows:

$(i)$ a 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;

$(ii)$ a mapping & cleaning scenario Doctors-MC based on the Doctors dataset, with the dependencies in Section I (a total of 4 source tables, 2 target tables, 3 tgds and 11 egds), and the partial order specification discussed in the paper;

$(iii)$ a 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;

$(iv)$ a data exchange scenario Doctors-DE based on the Doctors dataset, with the same dependencies as the mapping and cleaning one, but no conflicts among the sources; we generated a clean and consistent version of the source tables, and based on those the expected instance as the core universal solution for the given set of tgds and egds.

It remains to discuss how we fixed the expected instance for the Doctors-MC scenario. We considered the clean and consistent versions of the source tables used for scenario Doctors-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 adopt as an expected solution the core universal solution $C$ discussed above.

**Algorithms.** We run LLUNATIC with several cost managers and several caching strategies, as discussed in Sections VI, VII. In addition, we compared our system to several other algorithms in the literature, as follows:

$(a)$ an implementation of the *Mimimum Cost* algorithm proposed in [9] (MIN.COST) to repair FDs and IDs, in which IDs are repaired only by tuple insertions, and not by deletions or modifications;

$(b)$ an implementation of the PIPELINE algorithm in Section V, obtained by coupling a standard chase engine for tgds, and the repair algorithm for FDs in [10]; for the latter, for each experiment, we took 100 samples.

$(c)$ the DEMO system [14] chase engine for mappings.

**Quality Metrics.** 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 [15]. Based on this algorithm, we report two different quality measures. The first one is the actual similarity, *sim*(Rep, $DB_{exp}$), measured by the algorithm in [15]. In the comparison, lluns are considered as partial matches, and counted as 0.5 each.

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

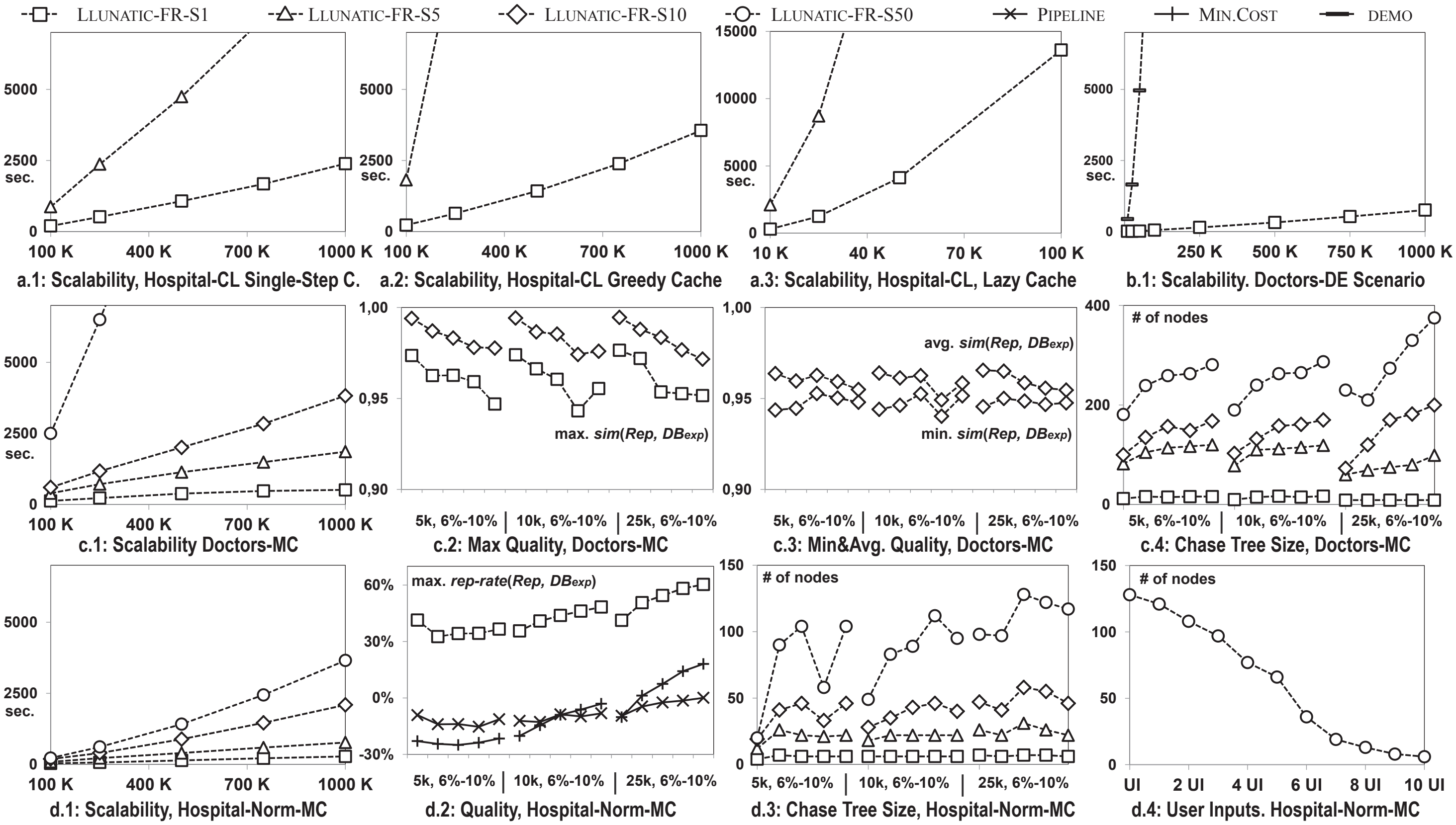

Fig. 3: Experimental results.

high similarity to the clean instance. In this case we report a repair rate defined as: $\textit{rep-rate}(\mathsf{Rep}, DB_{exp}) = 1 - (1 - \textit{sim}(\mathsf{Rep}, DB_{exp}))/(1 - \textit{sim}(DB_{dirty}, DB_{exp}))$. 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.

**Experiment a: Hospital-Den-CL** We report in Figures 3.$a_1$–$a_3$ scalability results for some of our cost managers and the different caching strategies discussed in Section VII (lazy, greedy, and single step). 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.

The optimizations introduced in this paper bring a dramatic improvement in terms of performance: the chase engine is up to four times faster than the original version reported in [5]. This is a significant achievement, since Hospital-Den is a sort of a worst-case scenario, with a large denormalized table with many FDs on it, and this aggravates query and update times.

**Experiment b: Doctors-DE** The scalability of our chase engine is confirmed in Figure 3.$b_1$. We compare the performance of LLUNATIC to the data exchange chase engine DEMO on scenario Doctors-DE. It can be seen that our implementation is orders of magnitude faster than DEMO.

**Experiment c: Doctors-MC** The overall scalability of the chase is confirmed on scenario Doctors-MC in Figure 3.$c_1$. In fact, the normalized nature of the data guarantees performance results that are significantly better than those reported for the denormalized scenario in Experiment a, 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 3.$c_4$. Consider also that each node in the tree is a copy of the entire database.

Figures 3.$c_2$–$c_3$ report the quality achieved by the various cost managers, in terms of the similarity to the core instance, $\textit{sim}(\mathsf{Rep}, DB_{exp})$. 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.

**Experiment d: Hospital-Norm-MC** Figure 3.$d_1$ confirms the excellent scalability of chasing tgds and egds on normalized databases, even with cost managers that produce multiple solutions and generate chase trees with hundreds of nodes (Figure 3.$d_3$). 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. Notice that experiment Hospital-Norm-MC is faster than Doctors-MC, even though the overall number of dependencies is similar. This is not surprising, since scenario Doctors-MC comprises the full range of dependencies that can be handled in our framework, including s-t tgds, while Hospital-Norm-MC relies on tgds only to express target referential integrity constraints.

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 3.$d_2$ we report metric *rep-rate*$(\mathsf{Rep}, DB_{exp})$ 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 Doctors-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 3.$d_4$, 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.

## IX. RELATED WORK

There has been a host of work on both data exchange and data quality management (see [16] and [3] for recent surveys, respectively). Although unifying approaches have been proposed, e.g., [9], [3], [17], these deal with specific classes of constraints only: [9] considers inclusion and functional dependencies; [3] extends this to their conditional counterparts; and [17] treats entity resolution together with data integration. Our mapping & cleaning approach is applicable to general classes of constraints and provides an elegant notion of solution.

This work is an extension of our earlier work on cleaning scenarios [5] by accommodating for mappings, and work on data exchange [2]. As discussed in [5], cleaning scenarios incorporate many data cleaning approaches including [9], [18], [6], [19], [10] and [20]. The same holds for mapping & cleaning scenarios. Furthermore, some of the ingredients of our scenarios are inspired by, but different from, features of other repairing approaches (e.g., repairing based on both premise and conclusion of constraints [18], [10], cells [10], [9], groups of cells [9], partial orders and its incorporation in the chase [21]). As previously observed, these approaches support limited classes of constraints. A flexible data quality system was recently proposed [22] which allows user-defined constraints but does not allow tgds.

Uncertainty in schema mappings has been investigated in [23], with reference to a different, data-integration setting.

We are not aware of any prior studies on optimizations for the chase. The only available chase engine for data exchange is DEMO [14], which hardly scales to large databases.

Algorithms for data repairing with preference relations were introduced in [24]. They only consider denial constraints, and are based on tuple deletions, not on cell changes; preferences are among tuples, not cell values. Also, they do not consider tgds, the main challenge dealt with in our framework.

Recently, a chase procedure to infer accuracy information represented by partial orders was devised in [25]. An integration of these ideas into our framework is left as future work.

## REFERENCES

[1] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin, "Translating Web Data," in *VLDB*, 2002, pp. 598–609.

[2] R. Fagin, P. Kolaitis, R. Miller, and L. Popa, "Data Exchange: Semantics and Query Answering," *TCS*, vol. 336, no. 1, pp. 89–124, 2005.

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

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

[5] F. Geerts, G. Mecca, P. Papotti, and D. Santoro, "The LLUNATIC Data-Cleaning Framework," *PVLDB*, vol. 6, no. 9, pp. 625–636, 2013.

[6] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu, "Towards certain fixes with editing rules and master data," *PVLDB*, vol. 3, no. 1, pp. 173–184, 2010.

[7] J. Bleiholder and F. Naumann, "Data fusion," *ACM Comp. Surv.*, vol. 41, no. 1, pp. 1–41, 2008.

[8] L. Chiticariu and W. C. Tan, "Debugging Schema Mappings with Routes," in *VLDB*, 2006, pp. 79–90.

[9] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi, "A cost-based model and effective heuristic for repairing constraints by value modification," in *SIGMOD*, 2005, pp. 143–154.

[10] G. Beskales, I. F. Ilyas, and L. Golab, "Sampling the repairs of functional dependency violations under hard constraints," *PVLDB*, vol. 3, pp. 197–207, 2010.

[11] S. Greco, F. Spezzano, and I. Trubitsyna, "Stratification criteria and rewriting techniques for checking chase termination," *PVLDB*, vol. 4, no. 11, pp. 1158–1168, 2011.

[12] B. ten Cate, L. Chiticariu, P. Kolaitis, and W. C. Tan, "Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries," *PVLDB*, vol. 2, no. 1, pp. 1006–1017, 2009.

[13] G. Mecca, P. Papotti, and S. Raunich, "Core Schema Mappings: Scalable Core Computations in Data Exchange," *Inf. Systems*, vol. 37, no. 7, pp. 677–711, 2012.

[14] R. Pichler and V. Savenkov, "DEMo: Data Exchange Modeling Tool," *PVLDB*, vol. 2, no. 2, pp. 1606–1609, 2009.

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

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

[17] M. Hernández, G. Koutrika, R. Krishnamurthy, L. Popa, and R. Wisnesky, "Hil: a high-level scripting language for entity integration," in *EDBT*, 2013, pp. 549–560.

[18] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in *VLDB*, 2007, pp. 315–326.

[19] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu, "Interaction Between Record Matching and Data Repairing," in *SIGMOD*, 2011, pp. 469–480.

[20] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas, "Guided data repair," *PVLDB*, vol. 4, no. 5, pp. 279–289, 2011.

[21] L. Bertossi, S. Kolahi, and L. Lakshmanan, "Data Cleaning and Query Answering with Matching Dependencies and Matching Functions," in *ICDT*, 2011, pp. 268–279.

[22] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. Ilyas, M. Ouzzani, and N. Tang, "Nadeef: a commodity data cleaning system," in *SIGMOD*, 2013, pp. 541–552.

[23] X. L. Dong, A. Y. Halevy, and C. Yu, "Data integration with uncertainty," *VLDB J.*, vol. 18, no. 2, pp. 469–500, 2007.

[24] S. Staworko, J. Chomicki, and J. Marcinkowski, "Prioritized repairing and consistent query answering in relational databases," *Ann. Math. Artif. Intell.*, vol. 64, no. 2-3, pp. 209–246, 2012.

[25] Y. Cao, W. Fan, and W. Yu, "Determining the relative accuracy of attributes," in *SIGMOD*, 2013, pp. 565–576.