The LLUNATIC Data-Cleaning Framework

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

[1] University of Antwerp – Antwerp, Belgium [2] Università della Basilicata – Potenza, Italy
[3] Qatar Computing Research Institute (QCRI) – Doha, Qatar [4] Università Roma Tre – Roma, Italy

ABSTRACT

Data-cleaning (or data-repairing) is considered a crucial problem in many database-related tasks. It consists in making a database consistent with respect to a set of given constraints. In recent years, repairing methods have been proposed for several classes of constraints. However, these methods rely on ad hoc decisions and tend to hard-code the strategy to repair conflicting values. As a consequence, there is currently no general algorithm to solve database repairing problems that involve different kinds of constraints and different strategies to select preferred values. In this paper we develop a uniform framework to solve this problem. We propose a new semantics for repairs, and a chase-based algorithm to compute minimal solutions. We implemented the framework in a DBMS-based prototype, and we report experimental results that confirm its good scalability and superior quality in computing repairs.

1. INTRODUCTION

In the constraint-based approach to data quality, a database is said to be dirty if it contains inconsistencies with respect to some set of constraints. The *data-cleaning* (or *data-repairing*) process consists in removing these inconsistencies in order to clean the database. It represents a crucial activity in many real-life information systems as unclean data often incurs economic loss and erroneous decisions [15].

Data cleaning is a long-standing research issue in the database community. Focusing on recent years, many interesting proposals have been put forward, all with the goal of handling the many facets of the data-cleaning process.

– A plenitude of constraint languages has been devised to capture various aspects of dirty data as inconsistencies of constraints. These constraint languages range from standard database dependency languages such as functional dependencies and inclusion dependencies [1], to conditional functional dependencies [16] and conditional inclusion dependencies [15], to matching dependencies [14] and editing-rules [18], among others. Each of these languages allows to capture different forms of dirtiness in data.

– Various repairing strategies have been proposed for these constraint languages. One of the distinguishing features of these strategies is how they use the constraints to modify the dirty data by changing values into "*preferred*" values. Preferred values can be found from, e.g., master data [24], tuple-certainty and value-accuracy [19], freshness and currency [17], just to name a few.

– Repairing strategies also differ in the kind of repairs that they compute. Since the computation of all possible repairs is infeasible in practice, conditions are imposed on the computed repairs to restrict the search space. These conditions include, e.g., various notions of (cost-based) minimality [7, 8, 10] and certain fixes [18]. Alternatively, sampling techniques are put in place to randomly select repairs [7].

It is thus safe to say that there is already a good arsenal of approaches and techniques for data cleaning at our disposal. In this paper, we want to capitalize on this wealth of knowledge about the subject, and investigate the following foundational problem: *what happens to the data administrator facing a complex data-cleaning problem that requires to bring together several of the techniques discussed above?* This problem is illustrated in the following example.

Example 1: Consider the database shown in Fig. 1 consisting of customer data (CUSTOMERS), with their addresses and credit-card numbers, and medical treatments paid by insurance plans (TREATMENTS). We refer to these two tables as the *target database* to be cleaned. As is common in corporate information systems [24], an additional *master-data table* is available; this table contains highly-curated records whose values have high accuracy and are assumed to be clean. In our approach, master data is referred to as the *source database*, since it is a source of reliable clean data.

CUSTOMERS

	SSN	NAME	PHONE	CONF	STR	CITY	CC#
t_1	111	M. White	408-3334	*0.8*	Red Ave.	NY	112321
t_2	222	L. Lennon	122-1876	*0.9*	NULL	SF	781658
t_3	222	L. Lennon	000-0000	*0.0*	Fry Dr.	SF	784659

TREATMENTS

	SSN	SALARY	INSUR.	TREAT	DATE
t_4	111	10K	Abx	Dental	10/1/2011
t_5	111	25K	Abx	Cholest.	8/12/2012
t_6	222	30K	Med	Eye surg.	6/10/2012

MASTER DATA *(Source Table)*

	SSN	NAME	PHONE	STR	CITY
t_m	222	F. Lennon	122-1876	Sky Dr.	SF

Figure 1: Customers, Treatments and Master Data.

We first illustrate the problem of specifying a set of constraints under which the target database is regarded to be clean, as follows:

$(a1)$ Standard functional dependencies (FD): $d_1 = (\text{SSN}, \text{NAME} \rightarrow \text{PHONE})$ and $d_2 = (\text{SSN}, \text{NAME} \rightarrow \text{CC\#})$ on table CUSTOMERS. The pair of tuples $\{t_2, t_3\}$ in the target database violates both d_1 and d_2; the database is thus dirty.

$(a2)$ A conditional FD (CFD): $d_3 = (\text{INSUR}[\text{Abx}] \rightarrow \text{TREAT}[\text{Dental}])$ on table TREATMENTS, expressing that insurance company 'Abx' only offers dental treatments ('Dental'). Tuple t_5 violates d_3, adding more dirtiness to the target database.

$(a3)$ A master-data based editing rule (eR), d_4, stating that whenever a tuple t in CUSTOMERS agrees on the SSN and PHONE attributes with some master-data tuple t_m, then the tuple t must take its NAME, STR, CITY attribute values from t_m. Tuple t_2 does not adhere to this rule.

$(a4)$ 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 $\{t_1, t_4\}$ and tuples $\{t_1, t_5\}$ violate this constraint.

With the dirty target database at hand, we are faced with the problem of repairing it. The main problem is to identify and select "preferred values" as modifications to repair the data.

$(b1)$ Consider FD d_1. To repair the target database one may want to equate $t_2[\text{PHONE}]$ and $t_3[\text{PHONE}]$. 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. If we assume that one prefers values with higher confidence, we can repair $t_3[\text{PHONE}]$ by changing it to '122-1876'.

$(b2)$ Similarly, 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 $d_6 = (\text{SSN} \rightarrow \text{SALARY})$, this may lead us to repair the obsolete salary value '10K' in t_4 with the more recent (and preferred) value '25K' in t_5.

$(b3)$ Notice that we don't always have a clear policy to choose preferred values. For example, when repairing $t_2[\text{CC\#}]$ and $t_3[\text{CC\#}]$ 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.

Another crucial aspect that complicates matters is the interaction between dependencies: repairing them in different orders may generate different repairs.

$(c1)$ Consider dependencies d_1 and d_4. As discussed above, we can use d_1 to repair tuples t_2, t_3 such that both have phone-number '122-1876'; then, since t_2 and t_3 agree with the master-data tuple t_m, we can use d_4 to fix names, streets and cities, to obtain: (222, F. Lennon, 122-1876, Sky Dr., SF, 781658), for t_2, and (222, F. Lennon, 122-1876, Sky Dr., SF, 784659), for t_3. However, if, on the contrary, we apply d_4 first, only t_2 can be repaired as before; then, since t_2 and t_3 do not share the same name anymore, d_1 has no violations. We thus get a different result, of inferior quality.

A first, striking observation about our example is that, despite many studies on the subject, there is currently no way to handle this kind of scenarios. This is due to several strong limitations of the known techniques.

Problem 1: Missing Semantics First, although repairing strategies exist for each of the individual classes of constraints discussed at items $(a1)$, $(a2)$ and $(a3)$, there is currently no formal semantics for their combination. In fact, the interactions shown in $(c1)$ require a uniform treatment of the repairing process and a clear definition of what is a repair. 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 this case.

Problem 2: Missing Repair Algorithms Since there is no semantics, we have no algorithms at our disposal to compute repairs. Notice that combining the repairing algorithms available for each of the constraints in isolation does not really help, since repairing a constraint of one type may break one of a different type. Also, current 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 illustrated in $(b1)$ and $(b2)$ into existing repairing algorithms in a principled way.

Problem 3: Main-Memory Implementations and Scalability Third, even if we were able to devise a reasonable semantics for this kind of scenarios, we would still face a paramount problem, i.e., computing solutions in a scalable way despite the high complexity of the problem. Computing repairs requires to explore a space of solutions of exponential size wrt the size of the database. In fact, previous proposals have mainly adopted main-memory implementations to speed-up the computation, with a rather limited scalability (in the order of the tens of thousands of tuples).

Contributions The main contribution of this paper consists in developing a uniform framework for data-cleaning problems that solves the issues discussed above. More specifically:

(i) We introduce a language to specify constraints based on *equality generating dependencies (egds)* [4] that generalizes many of the constraints used in the literature. This standardizes the way to express dependencies, and extends them to express inter-table constraints, with several benefits in terms of scalability, as discussed in our experiments.

(ii) The core contribution of the paper consists in the definition of a novel semantics for the data-cleaning problem. The definition of such a semantics is far from trivial, since our goal is to formalize the process of cleaning an instance as the process of *upgrading* its quality, regardless of the specific notions of value preference adopted in a given scenario. Our semantics 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 notion of an upgrade 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 paper, 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.

(iii) We introduce the notion of a *minimal solution* and develop algorithms to compute minimal solutions, based on a parallel-chase procedure. The definition of the chase is far from trivial, since our goal is to guarantee both generality and proper scalability. To start, we chase violations not at tuple level, but at *equivalence-class level* [8]. This allows us to 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 as

	DEPENDENCY LANGUAGE				REPAIR STRATEGY		VALUE PREFERENCE			SOLUTION SELECTION			
System	FDs	CFDs	ERs	Int.T.CFDs	RHS	LHS	Confid.	Currency	Master	Cost	Certain	Card.Min	Sampling
[8]	✓				✓		✓	✓		✓			
[10]	✓	✓			✓	✓	✓	✓		✓			
[23]	✓	✓			✓	✓				✓			
[18]			✓		✓				✓		✓		
[7]	✓				✓	✓						✓	✓
LLUNATIC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	ext. dependencies				chase proced.		partial order			cost manager			

Table 1: Feature Comparison.

the ones in Figures 2 and 3, 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.

CUSTOMERS

	SSN	NAME	PHONE	STR	CITY	CC#
t_1	111	M. White	408-3334	Red Ave.	**SF**	112321
t_2	222	**F. Lennon**	122-1876	**Sky Dr.**	SF	**L₀**
t_3	222	**F. Lennon**	**122-1876**	**Sky Dr.**	SF	**L₀**

TREATMENTS

	SSN	SALARY	INSUR.	TREAT	DATE
t_4	111	**25K**	Abx	Dental	10/1/2011
t_5	111	25K	Abx	**Dental**	8/12/2012
t_6	222	30K	Med	Eye surg.	6/10/2012

Figure 2: Repaired Instance #1.

CUSTOMERS

	SSN	NAME	PHONE	STR	CITY	CC#
t_1	**L₁**	M. White	408-3334	Red Ave.	NY	112321
t_2	**L₂**	L. Lennon	122-1876	NULL	SF	781658
t_3	222	L. Lennon	000-0000	Fly Dr.	SF	784659

TREATMENTS

	SSN	SALARY	INSUR.	TREAT	DATE
t_4	111	**25K**	Abx	Dental	10/1/2011
t_5	111	25K	**L₃**	Choles.	8/12/2012
t_6	222	30K	Med	Eye surg.	6/10/2012

Figure 3: Repaired Instance #2.

(iv) We develop an implementation of the chase engine, called LLUNATIC. To the best of our knowledge, LLUNATIC is the first system that runs over the DBMS to compute repairs. We devote special care in implementing our parallel chase – which may generate large trees of repairs – in a scalable way. A key ingredient of our solution is the development of an ad-hoc representation systems for solutions, called *delta relations*. 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.

We believe that these contributions make a significant advancement with respect to the state-of-the-art. To start, our proposal generalizes many previous approaches. Table 1 summarizes the features of LLUNATIC with respect to some of these approaches. LLUNATIC is the first proposal to achieve such a level of generality. Even more important, this work sheds some light on the crucial aspect of data-cleaning problems, namely the trade-offs between the quality of solutions and the complexity of repairing algorithms. This allows us to select data-repairing algorithms with good scalability and superior quality with respect to previous proposals, as our experiments show.

Organization of the Paper The preliminaries are in Section 2. In Sections 3, 4, and 5 we introduce the key components of the semantics of a cleaning scenario, which is defined in Section 6. The chase algorithm is described in Sections 7 and 8. Our experiments are reported in Section 9. Related work is described in Section 10.

2. PRELIMINARIES

We start by presenting some background notions and introducing the constraint language used in the paper.

A *schema* $\mathcal{S}$ 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. An *instance* $I = (I_1, \ldots, I_k)$ of $\mathcal{S}$ 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 $\mathcal{S}$ 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. 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 $\langle \mathcal{S}, \mathcal{T}\rangle$.

A relational atom over $\mathcal{T}$ is a formula of the form $R(\overline{x})$ with $R \in \mathcal{T}$ and $\overline{x}$ is a tuple of (not necessarily distinct) variables. Traditionally, 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 $\overline{x}$.

To express data-cleaning contraints, we rely on a specific form of egd. More specifically, besides relation atoms, we also consider *equation atoms* of the form $t_1 = t_2$, where t_1, t_2 are either constants in CONSTS or variables, and allow for both source and target atoms in the premise. In our approach, a *cleaning egd* is then 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 *and* equation 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.

Egds for our running example are expressed as follows:

$e_1.\ \mathit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p, s, c, cc), \mathit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p', s', c', cc') \rightarrow p = p'$
$e_2.\ \mathit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p, s, c, cc), \mathit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p', s', c', cc') \rightarrow cc = cc'$
$e_3.\ \mathit{Treat}(ssn, s, ins, tr, d), ins = \text{‘}\mathit{Abx}\text{’} \rightarrow tr = \text{‘}\mathit{Dental}\text{’}$
$e_4.\ \mathit{Cust}(\boldsymbol{ssn}, n, \boldsymbol{p}, s, c, cc), \mathit{MD}(\boldsymbol{ssn}, n', \boldsymbol{p}, s', c') \rightarrow n = n'$
$e_5.\ \mathit{Cust}(\boldsymbol{ssn}, n, \boldsymbol{p}, s, c, cc), \mathit{MD}(\boldsymbol{ssn}, n', \boldsymbol{p}, s', c') \rightarrow s = s'$
$e_6.\ \mathit{Cust}(\boldsymbol{ssn}, n, \boldsymbol{p}, s, c, cc), \mathit{MD}(\boldsymbol{ssn}, n', \boldsymbol{p}, s', c') \rightarrow c = c'$
$e_7.\ \mathit{Cust}(\boldsymbol{ssn}, n, p, str, c, cc), \mathit{Treat}(\boldsymbol{ssn}, sal, ins, tr, d),$
$\qquad ins = \text{‘}\mathit{Abx}\text{’} \rightarrow c = \text{‘}\mathit{SF}\text{’}$
$e_8.\ \mathit{Treat}(\boldsymbol{ssn}, s, ins, tr, d), \mathit{Treat}(\boldsymbol{ssn}, s', ins', tr', d') \rightarrow s = s'$

An immediate observation is that constants in egds can be avoided altogether, by encoding them in additional tables in the source database. Consider dependency e_3 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_{e3} with attributes INSUR and TREAT, corresponding to the "constant" attributes in e_3. Furthermore, we instantiate CST_{e3} with the single tuple t_{e_3} : (Abx, Dental). Given this, e_3 can be expressed as an egd without constants, as follows:

$$e'_3.\, \mathit{Treat}(ssn, s, \boldsymbol{ins}, tr, d), \mathit{Cst}_{e_3}(\boldsymbol{ins}, tr') \rightarrow tr = tr'$$

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 [16]. 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 paper.

Further extensions of egds with, e.g., built-in predicates, matching functions and negated atoms, are needed to encode matching dependencies and constraints for numerical attributes [20, 9]. We do not consider them in this paper for simplicity of exposition.

3. CLEANING SCENARIOS AND LLUNS

Our uniform framework for data repairing is centered around the concept of a *cleaning scenario*. A cleaning scenario consists essentially of a source schema $\mathcal{S}$, a target schema $\mathcal{T}$, and a set of constraints Σ. Here, $\mathcal{S}$ and $\mathcal{T}$ represent the two databases involved in the repairing process (see Example 1): (i) $\mathcal{S}$, the *source database*, provides clean and reliable information as input for the repairing process (like, for example, master data). We assume that source databases cannot be changed and consist of constants from CONSTS only; (ii) $\mathcal{T}$, the *target database*, corresponds to the database that is dirty relative to Σ, and that needs to be repaired. The target database may contain constants from CONSTS and null values from NULLS. Such null values indicate missing or unknown values. However, we also allow the target database to contain a third class of values, called *lluns* (pronounced "loons"), which we introduce next.

Recall from Example 1 that t_2 and t_3 form a violation for the dependency e_2 (stating that customers with equal ssns and names should have equal credit-card numbers), and that the target database could be repaired by equating t_2.CC# $= t_3$.CC#. However, as discussed before, 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_2.CC# and t_3.CC# into the llun L_0, that is to indicate that we need to introduce a new value for the credit-card number 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 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 to be resolved.

With this in mind, given an instance J of $\mathcal{T}$, along with an instance I of $\mathcal{S}$, the goal is to compute a *repair* of J, i.e., a set of updates to J such that the resulting instance satisfies the constraints in Σ.

Early works about database repairing [3, 21] followed an approach that relied on tuple-insertions and tuple-deletions. Since tuple-deletions may bring to unnecessary loss of information, the recent literature has concentrated on tuple updates, instead. Roughly speaking, we may say that the semantics adopted in these works are centered around three main ideas. First, a repair is seen as a set of changes to the *cells* of the database (each cell being an attribute of a tuple). Second, the logic to repair conflicting values is hardcoded into the semantics. Third, cost functions are used to (heuristically) compare different repairs and choose the "good" ones.

In the following sections, we develop a new semantics for cleaning scenarios that departs from this standard in three significant ways. Our first intuition is that, in order to generalize the semantics to larger classes of constraints and different ways to pick-up preferred values, it is not sufficient to reason about single-cell updates. On the contrary, we need to introduce a notion of "repairs with a lineage", called *cell groups*, in the sense that: (a) we keep track of cells that need to be repaired together; (b) we keep track of the provenance of their values, especially if they come from the source database.

A second, key idea, 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 cell groups. 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.

Finally, we introduce a principled way to check when a repaired instance satisfies the constraints, and to compare repairs with one another. This is based on an extension to data cleaning of the notion of instance homomorphism [12] that is typically used to compare the relative information content of database instances.

The next sections are devoted to these notions.

4. CELL GROUPS AND REPAIRS

Given instance $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{T}\rangle$, we represent the set of changes made to repair the target database J in terms of *cell groups*. As the name suggests, cell groups are groups of *cells*, i.e., locations in a database specified by tuple/attribute pairs $t_{tid}.A_i$. For example, t_2.CC# and t_3.CC# are two cells in the CUSTOMERS table. Observe the following:

(a) As our example shows, to repair inconsistencies, different cells are often changed *together*, i.e., they are either changed all at the same time or not changed at all. For example, t_2.CC# and t_3.CC# are both modified to the same llun value in Figure 2. Cell groups thus need to specify a set of target cells, called *occurrences* of the group, and a value to update them.

(b) In addition, in some cases the target cells to repair receive their value from the source database; consider Example 1 and dependency e_5. When repairing t_2, cell t_2.STREET gets the value 'Sky Dr.' from cell t_m.STREET. Since 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 c has a set of associated source cells carrying provenance information about the repair in terms of cells of the source database. We call these source cells the *justifications* for c, since they provide lineage information for the change we make to the target cells in c, i.e., to its occurrences. Occurrences and justifications need to be kept separate since we can only update target cells, while source cells are immutable.

(c) Cell groups provide an elegant way of describing repairs. Indeed, in order to specify a repair it suffices to provide the original target database together with the set of cell groups to modify. In other words, cell groups can be seen as partial repairs with lineage.

These observations are captured by the following definitions:

Definition 1 [CELL GROUP] A *cell group* g over an instance $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{T}\rangle$ is a triple $\langle v \rightarrow \mathcal{C}, by\, \mathcal{C}_s\rangle$ where: (i) v is a value in CONSTS $\cup$ NULLS $\cup$ LLUNS; (ii) $\mathcal{C}$ is a finite set of cells of the target instance, J, called the *occurrences* of g, denoted by $occ(g)$;

(iii) $\mathcal{C}_s$ is a finite set of cells of the source instance, I, called the *justifications* of g, denoted by *just*(g).

A cell group $g = \langle v \rightarrow \mathcal{C}, by\, \mathcal{C}_s \rangle$ can be read as "change the target cells in $\mathcal{C}$ to value v, justified by the source cells in $\mathcal{C}_s$". We define a *repair* to an instance $\langle I, J \rangle$ as a set of cell groups.

Definition 2 [REPAIR] A *repair* $\mathsf{Rep} = \{g_0, \ldots, g_k\}$ for instances $\langle I, J \rangle$ is a (possibly empty) finite set of cell groups over $\langle I, J \rangle$, such that each cell of J occurs in at most one cell group g_i.

That is, each cell in J is either unchanged in a repair or it is modified in a unique way as described by the cell group to which it belongs. We denote by $g_{\mathsf{Rep}}(c)$ the cell-group of cell c according to Rep.

Example 2: In our example, consider the repair Rep_1, consisting of the following cell groups referred to as $g_1, \ldots, g_7$:

$$\begin{aligned}
\mathsf{Rep}_1 = \{\, & g_1 : \langle L_0(\mathit{781658, 784659}) \rightarrow \{t_2.\text{CC\#}, t_3.\text{CC\#}\}, by\, \emptyset \rangle \\
& g_2 : \langle \mathit{F.Lennon} \rightarrow \{t_2.\text{NAME}, t_3.\text{NAME}\}, by\, \{t_m.\text{NAME}\} \rangle \\
& g_3 : \langle \mathit{122\text{-}1876} \rightarrow \{t_2.\text{PHN}, t_3.\text{PHN}\}, by\, \emptyset \rangle \\
& g_4 : \langle \mathit{Sky\ Dr.} \rightarrow \{t_2.\text{STR}, t_3.\text{STR}\}, by\, \{t_m.\text{STR}\} \rangle \\
& g_5 : \langle \mathit{Dental} \rightarrow \{t_5.\text{TREAT}\}, by\, \{t_{c_3}.\text{TREAT}\} \rangle \\
& g_6 : \langle \mathit{SF} \rightarrow \{t_1.\text{CITY}\}, by\, \{t_{c_7}.\text{CITY}\} \rangle \rangle \\
& g_7 : \langle \mathit{25K} \rightarrow \{t_4.\text{SALARY}, t_5.\text{SALARY}\}, by\, \emptyset \rangle \}
\end{aligned}$$

Cell group g_1 fixes credit-card numbers for dependency e_2; it has empty justifications because no source relation is involved in e_2. On the contrary, cell groups g_2 and g_4 repair tuples t_2, t_3 for dependencies e_4, e_5; justifications for these groups contain the respective cells in the master-data tuple t_m. Similarly, for cell groups g_5, g_6; here t_{c_3}, t_{c_7} are the tuples in the CST_{e_3}, CST_{e_7} tables, encoding the constants in the original CFDs.

When applied to the original database in Figure 1, repair Rep_1 yields the repaired instance shown in Figure 2. Clearly, other repairs are possible. For example, to resolve e_3 one may consider changing the value of the cell $t_5.\text{INSURANCE}$ into a new llun value L_1, i.e., an unknown value that improves 'Abx'. The following repair, Rep_2, follows the same approach to satisfy all dependencies, and yields the repaired instance shown in Figure 3:

$$\begin{aligned}
\mathsf{Rep}_2 = \{ & g_7,\ \langle L_1 \rightarrow \{t_1.\text{SSN}\}, by\, \emptyset \rangle, \langle L_2 \rightarrow \{t_2.\text{SSN}\}, by\, \emptyset \rangle \\
& \langle L_3 \rightarrow \{t_5.\text{INSURANCE}\}, by\, \emptyset \rangle \}
\end{aligned}$$

Note that J itself can be seen as the empty repair, $\mathsf{Rep}_\emptyset$.

Given $\langle I, J \rangle$, we say that a repair is *complete* if each cell of J occurs in a cell group in Rep, i.e., all cells in J are covered by the repair. We may assume, without loss of generality, that a repair is always complete. Indeed, a repair Rep can be easily completed into a complete repair Rep', as follows:

(i) initially, we let $\mathsf{Rep}' = \mathsf{Rep}$;

(ii) for each cell c of J that is not changed by Rep, if *val*$(c) \in$ CONSTS, then we add to Rep' the cell group $\langle \mathit{val}(c) \rightarrow \{c\}, by\, \emptyset \rangle$;

(iii) for each cell c of J that is not changed by Rep, if *val*$(c) \in$ NULLS, then we add to Rep' 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, we always assume a repair to be complete, and we blur the distinction between a repair Rep and the instance $\mathsf{Rep}(J)$ obtained by applying Rep to J.

5. THE PARTIAL ORDER

We are now ready to introduce another crucial ingredient of our framework: the partial order. The partial order is the core element of the semantics of our repairs and, as already mentioned, is used to indicate preferred upgrades to the target database. We want users to be able to specify different partial orders for different repairing problems in a simple manner. To do this, the user only has to specify for each attribute in the target schema when two values are preferred over each other. This is done by specifying an assignment Π of so-called *ordering attributes* to $\mathcal{T}$. As we will see shortly, such an assignment automatically induces a partial order on cell groups.

A Hierarchy of Information Content. In order to define our partial order, let us first introduce a simple hierarchy between the three kinds of values that appear in a database, namely nulls, constants, and lluns. More specifically, given two values $v_1, v_2 \in$ NULLS $\cup$ CONSTS $\cup$ 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 one of the following holds: (i) $v_1 \in$ NULLS, i.e., the first value is a null value; or (ii) $v_2 \in$ LLUN, i.e., the second value is a llun.

User-Specified preferred values. 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, 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. An *assignment* of ordering attributes to attributes in $\mathcal{T}$ is denoted by Π. For reasons that become clear shortly, Π is referred to as the *partial order specification*.

In our example, the DATE attribute in the TREATMENTS table, and the confidence column, CONF, in the CUSTOMERS table have ordered values (to simplify the treatment, we consider CONF as an attribute of the table). 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; however, the natural ordering of values does not reflect our notion of a preferred value. To model the correct notion of preference, we use DATE as the ordering attribute for SALARY (we prefer most recent salaries). Finally, attributes like SSN will have an empty associated poset, i.e., all constant values are equally preferred. Below is a summary of the assignment Π of ordering attributes in our example (attributes not listed have an empty poset):

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

Partial order on cell values. Given an assignment Π, we can define a corresponding partial order $\preceq_J^{\Pi}$ for the values of cells of the target instance J as follows. For any pair of values v_1, v_2 we say that $v_1 \preceq_J^{\Pi} v_2$ iff one of the following holds:

(i) either $v_1 = v_2$ or $v_1 \lhd v_2$, i.e., the values are equal or the second one is more informative than the first;

(ii) v_1 appears in cell $t_1.A_1$, v_2 in cell $t_2.A_2$ in J, 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, $v_1 \preceq_J^{\Pi} v_2$ iff $v_1 = v_2$ or $v'_1 < v'_2$ according to $\mathcal{P}_{A'_1} = \mathcal{P}_{A'_2}$.

We also consider values of the source instance I. In our approach, source values are immutable, and all equally preferable. So, we assume that the partial order $\preceq_I$ over values in I is based on rule (i) only. We call $\preceq^{\sqcap}_{\langle I,J\rangle}$ the partial order over values of cells in $\langle I,J\rangle$ obtained by the union of $\preceq^{\sqcap}_J$ and $\preceq_I$, with the additional rule that values of source cells are always preferable to values of target cells, i.e., for each target cell $t.A_t$ and source cell $t'.A_s$, it is always the case that $val(t.A_t) \preceq^{\sqcap}_{\langle I,J\rangle} val(t'.A_s)$. In fact, we always give preference to values from the source, like master-data or constant values in dependencies.

Given the partial order $\preceq^{\sqcap}_{\langle I,J\rangle}$, in the following we want to be able to compute upper bounds for cell values. To do this, we use lluns. Indeed, for any set C of cells we denote by $lub_{\preceq^{\sqcap}_{\langle I,J\rangle}}(C)$ the value that is (i) either the least upper bound for values of all cells in C according to $\preceq^{\sqcap}_{\langle I,J\rangle}$, if it exists; (ii) a new value N_i not in J, if all cells in C have null values; (iii) a new llun value L_j otherwise.

Partial order on cells groups. The partial order $\preceq^{\sqcap}_{\langle I,J\rangle}$ over cell values induces a partial order on the cell groups of $\langle I,J\rangle$. Before we turn to the definition, we want to exclude from the comparison cell groups that correspond to unjustified ways of changing the target. In order to do this, we say that a cell group g has a *valid value* if one of the following conditions holds. Consider the value val_{lub} that is the least upper bound of values in $occ(g) \cup just(g)$ according to $\preceq^{\sqcap}_{\langle I,J\rangle}$, i.e., $val_{lub} = lub_{\preceq^{\sqcap}_{\langle I,J\rangle}}(just(g) \cup occ(g))$. Then, either $val(g) = val_{lub}$, i.e., the cell group takes the value of the least upper bound, or $val_{lub} \lhd val(g)$, i.e., the cell group takes an even more informative value.

Given cell groups g and g' with valid values, we say that $g \preceq_{\sqcap} g'$ iff (i) $occ(g) \subseteq occ(g')$ and $just(g) \subseteq just(g')$, and (ii) either $val(g)$ and $val(g')$ are values of the same type (null, constant, or llun), or $val(g) \lhd val(g')$. In essence, we say that a cell group g' can only be preferred over a cell group g according to the partial order, if a *containment property* is satisfied, and the value of g' is at least as informative as the value of g. If the containment property is not satisfied for g and g' then these cell groups are incomparable relative to the partial order. Indeed, cell groups that change unrelated groups of cells represent incomparable ways to modify a target instance.

Example 3: Consider a simple relation $R(A,B)$, with three dependencies: (i) an FD $A \rightarrow B$, and two CFDs: (ii) $A[a] \rightarrow B[x], A[a] \rightarrow B[y]$. Notice that the two CFDs clearly contradict each other. Assume R contains two tuples: $t_1 : R(a,1), t_2 : R(a,2)$, and that $\mathcal{P}_A$ is A itself. Following is a set of ordered cell groups:

$$\begin{array}{cc}
\langle 1 \rightarrow \{t_1.\mathrm{B}\}, by\ \emptyset\rangle & \preceq_{\sqcap} \\
\langle 2 \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, by\ \emptyset\rangle & \preceq_{\sqcap} \\
\langle x \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, by\ \{t_{c1}.x\}\rangle & \preceq_{\sqcap} \\
\langle L \rightarrow \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, by\ \{t_{c1}.x, t_{c2}.y\}\rangle &
\end{array}$$

Partial order on repairs. Given an instance $\langle I,J\rangle$, a partial order $\preceq_{\sqcap}$ over cell groups in $\langle I,J\rangle$, and two complete repairs, Rep, Rep', we say that Rep' *upgrades* Rep, denoted by $\mathsf{Rep} \preceq_{\sqcap} \mathsf{Rep}'$, if for each group $g \in \mathsf{Rep}$ there exists a group $g' \in \mathsf{Rep}'$ such that $g \preceq_{\sqcap} g'$. If $\mathsf{Rep} \preceq_{\sqcap} \mathsf{Rep}'$ and the converse does not hold, then we write $\mathsf{Rep} \prec_{\sqcap} \mathsf{Rep}'$. A repair Rep' is thus preferable to Rep whenever $\mathsf{Rep} \preceq_{\sqcap} \mathsf{Rep}'$. This is where the real strength of the partial order lies: it provides a uniform way of incorporating information on preferred repairs.

Proposition 1: *Given an assignment $\sqcap$ of ordering attributes to attributes in $\mathcal{T}$, the corresponding partial order $\preceq^{\sqcap}_{\langle I,J\rangle}$ over values of cells in $\langle I,J\rangle$ induces a partial order $\preceq_{\sqcap}$ over the cell groups and repairs of $\langle I,J\rangle$. In fact, $\preceq_{\sqcap}$ is semi-join lattice.*

Notice that, besides the standard rules above, users may specify additional custom rules to plug-in other value-selection strategies and refine the lattice of cell groups. As an example, a *frequency rule* may state that the lub of cell groups g and g' with constant values c_1 and c_2 and empty justifications should take as value c_1 (c_2, resp.) if $|occ(g_1)| > |occ(g_2)|$ ($|occ(g_2)| > |occ(g_1)|$, resp.).

6. SEMANTICS

With the partial order specification $\sqcap$ in place, we now define a cleaning scenario as a quadruple $\mathcal{CS} = \{\langle \mathcal{S}, \mathcal{T}\rangle, \Sigma, \sqcap\}$. Given a cleaning scenario and an instance $\langle I,J\rangle$, we address the problem of defining a *solution* for $\mathcal{CS}$ over $\langle I,J\rangle$. Intuitively, a solution is a repair for $\langle I,J\rangle$ that satisfies the set Σ of egds and is an upgrade of the original target instance J relative to $\preceq_{\sqcap}$. We next formalize these notions.

Consider an instance $\langle I, \mathsf{Rep}(J)\rangle$ and a set Σ of constraints. Usually, $\mathsf{Rep}(J)$ is called a solution if $\langle I, \mathsf{Rep}(J)\rangle$ satisfies Σ using the standard semantics of first-order logic. Since we want to —rather ambitiously— ensure that there is *always* a solution we need to revise this semantics. In contrast, previous proposals often fail to return a repair or are stuck in an endless loop during repairing, as is illustrated next.

Consider dependency e_3 from Example 1. Suppose that a contradictory dependency $e''_3.Treat(ssn, s, ins, tr, d)$, $ins =$ '*Abx*' $\rightarrow tr =$ '*Cholest.*' is specified. In addition, assume that only modifications to the TREAT attribute-values are allowed. Clearly, there is no repair made of constants that can satisfy both dependencies [16]. However, one may consider of changing 'Dental' and 'Cholest.' to a llun L that improves both original values. In essence, the llun has the role of indicating to the user that the constraints are contradictory. In our setting, we want to regard this repair as a solution of a conflicting cleaning scenario.

Consider an egd $e : \forall x\, \phi(\bar{x}) \rightarrow x = x'$. First, recall that, in the standard semantics, $\langle I, \mathsf{Rep}(J)\rangle$ satisfies e if for any *homomorphism* h that maps the variables $\bar{x}$ into values of $\langle I, \mathsf{Rep}(J)\rangle$ such that $\phi(h(\bar{x}))$ is true, then also $h(x) = h(x')$ must be true. We want this to hold in our semantics as well. However, we want more. That is, we allow $h(x) \neq h(x')$ as long as the cell group corresponding to $h(x)$ is an *upgrade* to the cell groups corresponding to $h(x')$, or vice versa.

To make this precise, we need to extend h to a mapping from variables to cell groups. Since h associates values to variables, it also associates with each variable $x_i \in \bar{x}$ a set of cells from $\langle I, \mathsf{Rep}(J)\rangle$, called $cells_h(x_i)$, one for each occurrence of x_i and all with the same value, $h(x_i)$. We use these to define the cell group of x_i according to h, as follows.

Given a formula $\phi(\overline{x})$, a repair Rep, an homomorphism h of $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J)\rangle$, and a variable $x_i \in \bar{x}$, the *cell group of x_i according to h* is defined as $g_h(x_i) = \langle h(x_i) \rightarrow \mathcal{C}, by\ \mathcal{C}_s\rangle$ where $\mathcal{C}$ (resp. $\mathcal{C}_s$) is the union of all occurrences (resp. justifications) of cell groups $g_{\mathsf{Rep}}(c_i)$ in Rep, for each cell $c_i \in cells_h(x_i)$. In addition, $\mathcal{C}_s$ contains all cells in $cells_h(x_i)$ that belong to the source I.

We are now ready to introduce our extended notion of satisfaction, namely *satisfaction after repairs*:

Definition 3 [SATISFACTION AFTER REPAIRS] Given an egd $e : \forall \overline{x}\ \phi(\overline{x}) \rightarrow x = x'$, an instance $\langle I,J\rangle$, and a repair Rep, we say that $\langle I, \mathsf{Rep}(J)\rangle$ *satisfies after repairs* e wrt the partial order $\preceq_{\sqcap}$ if, whenever there is an homomorphism h of $\phi(\overline{x})$ into $\langle I, \mathsf{Rep}(J)\rangle$, then (i) either the value of $h(x)$ and $h(x')$ are equal, or (ii) it is the case that $g_h(x) \preceq_{\sqcap} g_h(x')$ or $g_h(x') \preceq_{\sqcap} g_h(x)$.

We can now find a repair that satisfies the conflicting egds e_3 and e''_3 above. Given a tuple t in the target, consider Rep that repairs t.TREAT with L, and justifies the change with both cells in the source corresponding to constants '*Dental*' and '*Cholest.*'. Now, despite the fact that L is not equal to any of the constants in the dependencies, both dependencies are satisfied after repairs by $\mathsf{Rep}(J)$.

Definition 4 [SOLUTION] Given a cleaning scenario $\mathcal{CS} = \{\langle \mathcal{S}, \mathcal{T} \rangle, \Sigma, \sqcap\}$ and instance $\langle I, J \rangle$ a *solution* for $\mathcal{CS}$ over $\langle I, J \rangle$ is a repair Rep such that: (i) $J \preceq_{\sqcap}$ Rep, i.e., Rep upgrades J; and (ii) $\langle I, \mathsf{Rep}(J) \rangle$ satisfies after repairs Σ wrt $\preceq_{\sqcap}$.

An important property of cleaning scenarios is that every input instance has a solution, albeit a solution that is not necessarily minimal and is rather uninformative.

Theorem 2: *Given a scenario $\mathcal{CS} = \{\langle \mathcal{S}, \mathcal{T} \rangle, \Sigma, \sqcap\}$ 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 repair that changes all cells of J to a single llun L, and justifies it by all cells in I, i.e., $\mathsf{Rep}_{trivial} = \langle L \rightarrow \mathit{cells}(J), \mathit{by}\ \mathit{cells}(I) \rangle$. $\square$

Among all possible repairs, we are interested in those that minimally upgrade the dirty instance.

Definition 5 [MINIMAL SOLUTION] A *minimal solution* for a cleaning scenario is any solution Rep that is minimal wrt $\prec_{\sqcap}$, i.e., such that there exists no other solution Rep$'$ such that Rep$' \prec_{\sqcap}$ Rep.

The repair Rep_1 in Example 2 is a minimal solution for the scenario in Example 1: it is an upgrade of J, it satisfies the dependencies, and by undoing any of its changes violations arise. Minimal solutions are not unique. Indeed, also repair Rep_2 in Example 2 is a minimal solution. As an example of a non-minimal solution, one can add to Rep_2 the cell group $\langle L_4 \rightarrow \{t_2.\text{NAME}\}, \mathit{by}\ \emptyset \rangle$. The resulting repair Rep_3 is still a solution but not a minimal one ($\mathsf{Rep}_2 \prec_{\sqcap} \mathsf{Rep}_3$). Consider now repair Rep_4, obtained by adding a cell group $\langle \mathit{111111} \rightarrow \{t_2.\ \text{CC\#}\}, \mathit{by}\ \emptyset \rangle$ to Rep_2. In this case, Rep_4 is not a solution because the last cell group is totally unjustified wrt the partial order, and therefore it is not true that $\mathsf{Rep}_4(J)$ is an upgrade of the original target instance.

An important property is that two repairs can be efficiently compared wrt to the partial order. We assume here that the partial order of two values $v \preceq^{\sqcap}_{\langle I,J \rangle} v'$ can be checked in constant time.

Theorem 3: *Given two solutions* Rep, Rep$'$ *for a scenario $\mathcal{CS}$ over instance $\langle I, J \rangle$, one can check* Rep $\preceq_{\sqcap}$ Rep$'$ *in $O(n+kmlog(m))$ time, where n is the number of cells in J, k is the maximum number of cell groups in* Rep, Rep$'$*, and m is the maximum size of a cell group in* Rep, Rep$'$.

Given a cleaning scenario $\mathcal{CS}$ and an instance $\langle I, J \rangle$, the *data repairing problem* consists of computing all minimal solutions for $\mathcal{CS}$ over $\langle I, J \rangle$. We provide a chase-based algorithm for the data repairing problem in the next section.

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 repairs, 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 3 in Section 5. The cell group $\langle L \rightarrow \{t_1.\text{B}, t_2.\text{B}\}, \mathit{by}\ \{t_{c1}.x, t_{c2}.y\} \rangle$ 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 – where involved, and also in terms of source values that justify the change.

7. COMPUTING SOLUTIONS

In order to generate solutions for cleaning scenarios, we resort to a variant of the traditional chase procedure for egds [12]. However, our chase is a significant departure from the standard one, for several reasons: (i) during the chase, we shall make extensive use of the partial order, $\preceq_{\sqcap}$; (ii) to generate all possible solutions, a dependency may be chased both *forward*, to satisfy its conclusion, or *backward*, to falsify its premise; this, in turn, means that the we need to consider a *disjunctive chase*, which generates a tree of alternative repairs; (iii) finally, and most important, we shall not consider violations at the tuple level, as it is common [12], but at the higher level of *equivalence classes*.

To explain this latter difference, consider a simple functional dependency $A \rightarrow B$ over relation $R(A, B, C)$, with tuples $t_1 = R(1, 2, x)$, $t_2 = R(1, 2, y)$, $t_3 = R(1, 4, z)$, $t_4 = R(2, 5, w)$, $t_5 = R(2, 5, v)$. It is highly inefficient to analyze the violations of this FD at the tuple level; in fact, eventually, the B value of t_1, t_2, t_3 will all become equal, and therefore one may prefer grouping and fixing them together. In the literature [8, 15] this has been formalized by means of *equivalence classes*. We want to introduce a similar concept into our chase algorithm. Given the higher generality of our dependency language, we need a number of preliminary definitions.

Preliminary Notions 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. We first introduce the notions of *witness* and *witness variable* for a dependency e. 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 6 [WITNESS] Let $e : \forall \bar{x}\ (\phi(\bar{x}) \rightarrow x = x')$ be an egd. A *witness variable* for e is a variable $x \in \bar{x}$ that has multiple occurrences in $\phi(\bar{x})$. For 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.

Consider, for example, dependency e_8 in Example 1 (we omit some of the variables for the sake of conciseness): e_8. $\mathit{Treat}(\mathbf{ssn}, s, \ldots), \mathit{Treat}(\mathbf{ssn}, s', \ldots) \rightarrow s = s'$. Assume that the target instance TREATMENTS contains tuples $t_4 = (\mathit{ssn} : 222, \mathit{salary} : 10K, \ldots)$, $t_5 = (\mathit{ssn} : 222, \mathit{salary} : 25K, \ldots)$. We have an homomorphism h that maps the first atom of e_8 into t_4, and the second one into t_5. 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 222.

Definition 7 [EQUIVALENCE CLASS] Given a repair Rep, and an egd $e : \forall \bar{x}\ (\phi(\bar{x}) \rightarrow x = x')$, let $\bar{x}_w \subseteq \bar{x}$ be the witness variables of e. An *equivalence* class for Rep and e, $\mathcal{H}$, is a set of homomorphisms of $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J) \rangle$ such that all $h_i \in \mathcal{H}$ have equal witness values $h_i(\bar{x}_w)$.

Notice that equivalence classes induce classes of tuples in a natural way. In our example above, the tuples are partitioned into two equivalence 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 equivalence 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 set of *witness groups*, $\textsf{w-groups}_{\mathcal{H}}$, and *conclusion groups*, $\textsf{c-groups}_{\mathcal{H}}$, for $\mathcal{H}$ and e, as the set of cell groups associated by any homomorphism $h \in \mathcal{H}$ with the witness variables, $\bar{x}_w$, and the conclusion variables, x, x', respectively:

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

We say that an equivalence class for $\textsf{Rep}$ 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 $g_1, g_2 \in \textsf{c-groups}_{\mathcal{H}}$ such that $\textit{val}(g_1) \neq \textit{val}(g_2)$ and neither $g_1 \preceq_{\Pi} g_2$ nor $g_2 \preceq_{\Pi} g_1$. In this case, we say that e is *applicable* to $\langle I, \textsf{Rep}(J)\rangle$ with $\mathcal{H}$.

The Chase We are now ready to define the notion of a chase step. 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 equivalence class, which provides the hook to introduce the notion of a cost manager in the next section.

A *repair strategy* $rs_{\mathcal{H}}$ for $\mathcal{H}$ is a mapping from the set of conclusion cell-groups, $\textsf{c-groups}_{\mathcal{H}}$ of $\textsf{Rep}$ and $\mathcal{H}$, into the set $\{f, b\}$ (where f stands for "forward", and b for "backward"). We call the *forward groups*, $\textsf{forw-g}_{rs_{\mathcal{H}}}$, of $rs_{\mathcal{H}}$ the set of groups g_i such that $rs_{\mathcal{H}}(g_i) = f$, and the *backward groups*, $\textsf{back-g}_{rs_{\mathcal{H}}}$, those such that $rs_{\mathcal{H}}(g_i) = b$.

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

(i) it belongs to the same tuple as c_i;

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

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

Observe that we do not chase backward in two cases: first, when cells 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 of the repair; second, when cell values have a 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.

Each chase step is defined based on a specific repair strategy.

Definition 8 [CHASE STEP] Given a cleaning scenario $\mathcal{CS} = \{\mathcal{S}, \mathcal{T}, \Sigma, \Pi\}$, and a complete repair $\textsf{Rep}$ of J, let $e : \forall \overline{x}\ (\phi(\overline{x}) \rightarrow x = x')$ be an egd in Σ, applicable to $\langle I, \textsf{Rep}(J)\rangle$ with $\mathcal{H}$. For each repair strategy $rs_{\mathcal{H}}$, a *chase step* generates a new repair $\textsf{Rep}_{rs_{\mathcal{H}}}$ defined as follows:

(i) to start, we initialize $\textsf{Rep}_{rs_{\mathcal{H}}} = \textsf{Rep}$

(ii) then, we replace all forward groups by their least upper bound:

$$\textsf{Rep}_{rs_{\mathcal{H}}} = \textsf{Rep}_{rs_{\mathcal{H}}} - \textsf{forw-g}_{rs_{\mathcal{H}}} \cup \textit{lub}_{\preceq_{\Pi}}(\textsf{forw-g}_{rs_{\mathcal{H}}})$$

(iii) finally, we add the backward repairs, i.e, for each backward group $g \in \textsf{back-g}_{rs_{\mathcal{H}}}$, and cell $c_i \in \textit{occ}(g)$, we replace $g_i = g_{\textsf{Rep}}($ $\textsf{w-cell}_{rs_{\mathcal{H}}}(c_i))$ by the cell group g'_i that is an immediate successor of g_i according to $\preceq_{\Pi}$ as follows:

$$\textsf{Rep}_{rs_{\mathcal{H}}} = \textsf{Rep}_{rs_{\mathcal{H}}} - \{g_i\} \cup \{g'_i\}$$

Note that such a successor always exists. Indeed, $g'_i = \langle L_i \rightarrow \textit{occ}(g_i), \textit{by}\ \emptyset\rangle$, where L_i is a new LLUN value, is a successor of g_i.

Given $\textsf{Rep}$, each repair strategy $rs^i_{\mathcal{H}}$ for $\mathcal{H}$ generates a different step, $\textsf{Rep}_{rs^i_{\mathcal{H}}}$. We simultaneously consider all these chase steps, in parallel, and write $\textsf{Rep} \rightarrow_{e,\mathcal{H}} \textsf{Rep}_{rs^0_{\mathcal{H}}}, \textsf{Rep}_{rs^1_{\mathcal{H}}} \ldots, \textsf{Rep}_{rs^n_{\mathcal{H}}}$.

Consider again dependency e_8 in Example 1, and the equivalence class associated with witness $\textit{ssn} = 222$. The cell groups for the conclusion cells are: $g = \langle 10K \rightarrow \{t_4.\text{SALARY}\}, \textit{by}\ \emptyset\rangle$ and $g' = \langle 25K \rightarrow \{t_5.\text{SALARY}\}, \textit{by}\ \emptyset\rangle$. Notice that the two cell groups are incomparable, and therefore we have a violation. The chase procedure generates three different repairs for the violation: (a) the forward repair is: $\textsf{Rep}_{f,f} = \langle 25K \rightarrow \{t_4.\text{SALARY}, t_5.\text{SALARY}\}, \textit{by}\ \emptyset\rangle$ ($25K$ 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) the first backward repair, which changes the first occurrence of the witness variable $\textit{ssn}$ to a llun L_1: $\textsf{Rep}_{b,f} = \langle L_1 \rightarrow \{t_4.\text{SSN}\}, \textit{by}\ \emptyset\rangle$; (c) the second backward repair, changing the second occurrence of $\textit{ssn}$ to L_2: $\textsf{Rep}_{f,b} = \langle L_2 \rightarrow \{t_5.\text{SSN}\}, \textit{by}\ \emptyset\rangle$.

Definition 9 [CHASE TREE] Given a cleaning scenario $\mathcal{CS} = \{\mathcal{S}, \mathcal{T}, \Sigma, \Pi\}$, a *chase* of $\langle I, J\rangle$ with Σ is a tree whose root is $\langle I, J\rangle$, i.e., the empty repair, and for each node $\textsf{Rep}$, the children of $\textsf{Rep}$ are the repairs $\textsf{Rep}_0, \textsf{Rep}_1, \ldots, \textsf{Rep}_n$ such that, for some $e \in \Sigma$ and some $\mathcal{H}$, it is the case that $\textsf{Rep} \rightarrow_{e,\mathcal{H}} \textsf{Rep}_0, \textsf{Rep}_1, \ldots, \textsf{Rep}_n$. The leaves are repairs $\textsf{Rep}_\ell$ such that there is no dependency applicable to $\langle I, \textsf{Rep}_\ell(J)\rangle$ with some equivalence class $\mathcal{H}$. Any leaf in the chase tree is called a *result* of the chase of $\langle I, J\rangle$ with Σ.

Note that, as usual, the chase procedure is sensitive to the order of application of the dependencies. Different orders of application of the dependencies may lead to different chase sequences and therefore to different results.

We next show that the chase procedure always generates solutions, i.e., it is sound, and it terminates after a finite number of steps. Furthermore, all minimal solutions can be obtained in this way, i.e., the chase is complete for minimal solutions.

Theorem 4: *Given a cleaning scenario $\mathcal{CS} = \{\mathcal{S}, \mathcal{T}, \Sigma, \Pi\}$ and an instance $\langle I, J\rangle$, the chase of $\langle I, J\rangle$ with Σ (i) terminates; (ii) it generates a finite set of results, each of which is a solution for $\mathcal{CS}$ over $\langle I, J\rangle$; and (iii) it generates all minimal solutions.*

Complexity It is well-known [5] 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. 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 repairs such that $f(\textsf{Rep}) < f(\textsf{Rep}')$ whenever $\textsf{Rep} \rightarrow_{e,\mathcal{H}} \textsf{Rep}'$ during the chase. Intuitively, f depends on the number of llun values and sizes of cell groups in the repairs. 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. We discuss techniques to handle this high complexity in the next section.

8. A SCALABLE CHASE

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 number of techniques that improve the scalability of the chase, namely: a central component of our framework, the *cost manager*, and a representation systems for chase trees, called *delta databases*.

8.1 Introducing 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 repairs [3, 8, 7], certain regions [18], and sampling [7]. 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 10 [COST MANAGER] Given a cleaning scenario, $\mathcal{CS}$ and instance $\langle I, J\rangle$, a *cost manager* for $\mathcal{CS}$ and $\langle I, J\rangle$ is a predicate CM over repair strategies to be used during the chase. For each repair strategy $rs_{\mathcal{H}}$ for equivalence class $\mathcal{H}$, it may either accept it ($\text{CM}(rs_{\mathcal{H}}) = \textit{true}$), or refuse it ($\text{CM}(rs_{\mathcal{H}}) = \textit{false}$).

During the chase, we shall systematically make use of the cost manager. Whenever we need to chase an equivalence class, we only generate repairs corresponding to repair strategies accepted by the cost manager. The standard cost manager is the one that accepts all repair 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 repairs 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 *frequency* cost manager (FR): in order to repair equivalence class $\mathcal{H}$ for dependency e, FR adopts the following rules; 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;

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

Notice that combinations of these strategies are possible, to obtain, e.g., a FR-S5 or a FR-S1-FO cost manager. The FR-S5 relies on value frequencies and, in addition, it considers five different permutations of the dependencies, and for each of them will compute one repair. Alternative cost managers may implement different pruning strategies, to incorporate the notion of a *certain region*, and refute all steps in which changes are made to attributes of the target that are considered to be "fixed", i.e., reliable, or perform different forms of sampling. In the following, we shall always assume that a cost manager has been selected in order to perform the chase.

8.2 Delta Databases

Even with cost managers in place, the parallel nature of our chase algorithm imposes to store a possibly large tree of repairs. 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* [2], 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, *stepId*$(\textit{father}(n))$ is a prefix of *stepId*(n), for each 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.

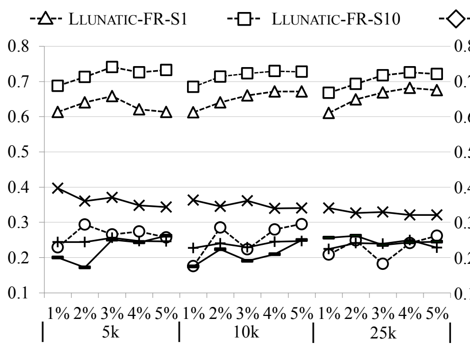
Definition 11 [DELTA DATABASE] Given a target database schema $\mathcal{R} = \{R_1, \ldots, R_k\}$, a *delta database* for $\mathcal{R}$ contains the following tables: (i) a *delta table* $R_i\_A_j$ with attributes (t_{id}, *stepId, value*), for each R_i and each attribute A_j of R_i; (ii) a table *occurrences*, 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 $(t_{id}, \textit{stepId}, v)$. 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).

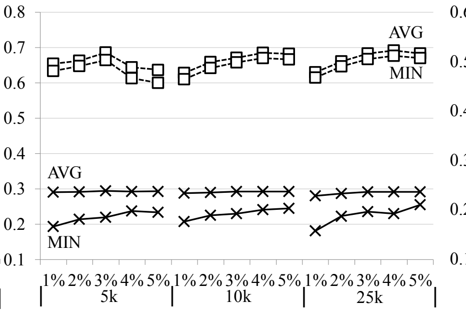
In the next section we show how the combination of our advanced chase procedure and its implementation under the form of delta databases scale to large repairing problems with millions of tuples and large chase trees.

9. EXPERIMENTS

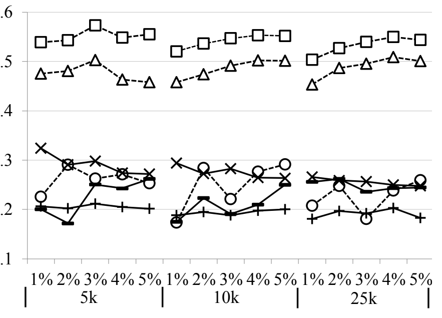
The proposed algorithms have been implemented in a working prototype of the LLUNATIC system, written in Java. In this section, we consider several cleaning scenarios, of different nature and sizes, and study both the quality of the repairs computed by our system, and the scalability of the chase algorithm. We show that our algorithm produce repairs 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 a Intel i7 machine with 2.6Ghz processor and 8GB of RAM under Linux. The DBMS was PostgreSQL 9.2.



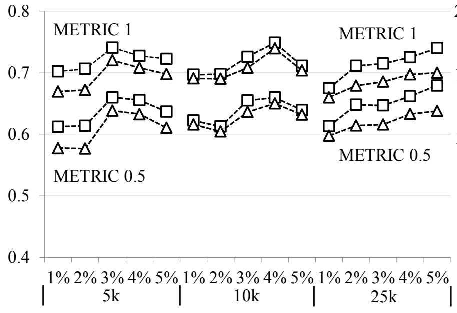
(a) Max F-Measure (Metric 1) for HOSPITAL



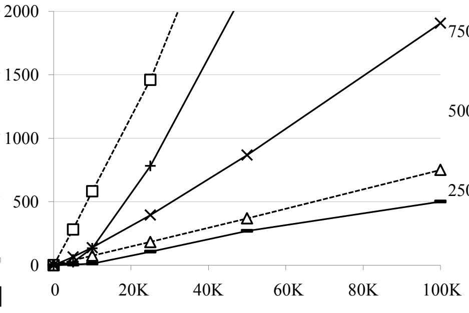
(b) Avg, Min F-Measure (Metric 1) for HOSPITAL



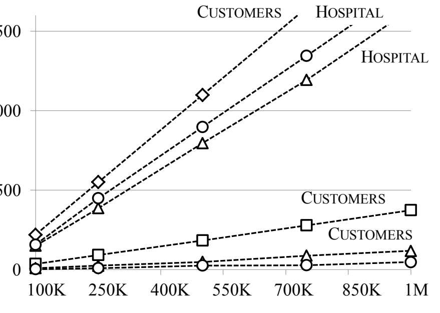
(c) Max F-Measure (Metric 0.5) for HOSPITAL



(d) Max F-Measure for the CUSTOMERS Scenario



(e) HOSPITAL Execution Times (sec)



(f) Large Datasets Execution Times (sec)

Figure 4: Experimental results for HOSPITAL and CUSTOMERS.

Datasets and Scenarios. We selected two scenarios. (i) The first one, HOSPITAL, is based on a dataset from US Department of Health & Human Services (http://www.medicare.gov/hospitalcompare/). The database contains a single table with 100K tuples and 19 attributes, over which we specified 9 functional dependencies. (ii) The second one, CUSTOMERS, corresponds to our running example in Figure 1. The database schema contains 3 tables with 16 attributes, plus 2 additional tables encoding constants in CFDs. Dependencies are the ones in Section 1. We synthetically generated up to 1M tuples for the 2 target relations, with a proportion of 40% in the CUSTOMERS table, and 60% in TREATMENTS; the master-data table contains 20% of the tuples present in CUSTOMERS. We consider master-data tuples outside the total, as they cannot be modified. For this scenario, we defined the partial order as discussed in Section 5.

It is worth noting that these scenarios somehow represent opposite extremes of the spectrum of data-repairing problems. In fact, the HOSPITAL scenario contains functional dependencies only, and therefore is quite standard in terms of constraints; however, it 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 scenario contains a complex mix of dependencies; this increased complexity of the constraints is compensated by the fact that data is stored as two normalized tables, with no redundancy, and clear preference strategies are given for some of the attributes.

Errors. In order to test our algorithms with different levels of noise, we introduced errors in the two 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 dataset 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. 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.

For both datasets, we generated dirty copies with a number of noisy cells ranging from 1% to 5% of the total. Changes to the original values were done only for attributes involved in dependencies, in order to maximize the probability of generating detectable violations.

Algorithms. We tested LLUNATIC with several cost managers chosen among those presented in Section 8. 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 also the following FD repair algorithms from the literature, implemented as separate systems: (a) *Mimimum Cost* [8] (MIN. COST); (b) *Vertex Cover* [23] (VERTEX COVER); (c) *Repair Sampling* [7] (SAMPLING), for which, for each experiment, we took 500 samples, as done in the original paper.

Notice that these systems support a smaller class of constraints wrt to the ones expressible with cleaning egds (essentially FDs and, in some cases, CFDs). Several of the constraints in the CUSTOMERS scenario are outside of this class, and therefore cannot handled by these algorithms. We therefore performed the comparison on the HOSPITAL scenario only.

Quality Metrics. We used precision-recall metrics. 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 as repairs – like our lluns – we calculated two different metrics.

The first one is the one adopted in [7], 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.

Since our scenarios may require a consistent number of variables, due to the need for backward repairs, 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.

Whenever an algorithm returned more than one repair for a database, we calculated P, R, and F for each repair; in the graphs, we report the maximum, minimum, and average values.

Quality Figure 4 shows quality and scalability results. 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%). 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.

The maximum F-measure for Metric 1 is in Figure 4.(a); for the two algorithms that return more than one solution, the minimum and average F-measures are reported in Figure 4.(b). The maximum F-measure for Metric 0.5 is in Figure 4.(c). Quality results for algorithms MIN. COST, VERTEX COVER, and REP. SAMPLING are consistent with those reported in [7], which also conducted a comparison of these three algorithms on scenarios in which left and right-hand-side 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 $A \to B$ 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 the right-hand side of the dependency, thus missing the correct repair. Finally, even when a left-hand-side 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 right and left-hand sides of dependencies 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 is a more systematic way, and this explains its improvements in quality.

Figures 4.(d) reports results for the CUSTOMERS 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; this is a consequence of the clear user-specified preference rules.

Scalability The trade-offs between quality and scalability are shown in Figures 4.(e) and 4.(f). Figure 4.(e) compares execution times for the various algorithms on the HOSPITAL scenario 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 had better performances than some of the main memory implementations. We may therefore say that graphs (c) and (e) in Figure 4 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 REP. SAMPLING algorithm: we noticed that taking 1000 samples instead of 500 doubles execution times, but it does not produce significant improvements in quality.

Figure 4.(f) clearly shows 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 [8, 7] 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 figure shows that LLUNATIC scales in both scenarios to large databases. For the HOSPITAL scenario we replicated the original dataset ten times with 1% errors. In these cases, execution times in the order of the hours 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. It is interesting to note that performances were significantly better on the CUSTOMERS scenario. This is not surprising: as we discussed above, the CUSTOMERS database contains non redundant, normalize tables. In fact, this clearly shows the benefit of a constraint language that allows to express inter-table cleaning constraints.

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.

10. RELATED WORK

Several classes of constraints have been proposed to characterize and improve the quality of data (see [13, 15] for surveys). Most relevant to this paper are the (semi-)automated repairing algorithms for these constraints [7, 8, 10, 18, 19, 23]. These methods differ in the constraints that they admit, e.g., FDs [7, 8], CFDs [10, 23], inclusion dependencies [8], and editing rules [18], and the underlying techniques used to improve their effectiveness and efficiency, e.g., statistical inference [10], measures of the reliability of the data [8, 18], and user interaction [10, 25].

All of these methods work for a specific class of constraints only, with the exception of [19, 9]. 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.

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. More complex operations are also partially available, but lack the support for constraints.

We do allow for forward and backward chasing. Similarly, [10, 23, 7] 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 [23, 7] 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 [22] 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 [6], 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 paper, 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 [6] – 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.

A comparison of the features supported by existing methods and our repairing method is given in Table 1. We believe that this work makes a concrete step forward towards the goal of developing a uniform formalism for data cleaning, and may stimulate further research on this subject. With a similar spirit, [11] has developed a unifying view of previous approaches by abstracting different classes of constraints with respect to a different problem, that of query answering over inconsistent data.

Our framework can be seen as an extension of the data exchange setting [12]. With respect to the standard chase algorithms for egds, our chase *always terminates* and *never fails*, by leveraging the partial order. We are not aware of any extension of the data exchange setting that allows the introduction of special values (like lluns) to avoid failing chase computations. In fact, we are currently extending our formalism to accommodate for *mapping and cleaning* scenarios, in such a way to maintain the results from the data exchange literature and enlarge them to data repairing.

11. REFERENCES

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

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

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

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

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

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

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

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

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

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

[11] T. Eiter, M. Fink, G. Greco, and D. Lembo. Repair Localization for Query Answering from Inconsistent Databases. *ACM TODS*, 33(2):1–51, 2008.

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

[13] W. Fan. Dependencies Revisited for Improving Data Quality. In *PODS*, pages 159–170, 2008.

[14] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDB J.*, 20(4):495–520, 2011.

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

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

[17] W. Fan, F. Geerts, and J. Wijsen. Determining the Currency of Data. In *PODS*, pages 71–82, 2011.

[18] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *PVLDB*, 3(1):173–184, 2010.

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

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

[21] G. Greco, S. Greco, and E. Zumpano. A Logical Framework for Querying and Repairing Inconsistent Databases. *TKDE*, 15(6):1389–1408, 2003.

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

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

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

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