**Noname manuscript No.**
(will be inserted by the editor)

# Cleaning Data with Llunatic

**Floris Geerts · Giansalvatore Mecca · Paolo Papotti · Donatello Santoro**

April 29, 2019

**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, declarative 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, encompassing many existing repair semantics, 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 [7,21,31]. The corresponding *data-repairing* process consists in removing these inconsistencies in order to clean the database. The modeling and repairing of dirty data represents a crucial activity in many real-life information systems. Indeed, unclean data often incurs economic loss and erroneous decisions [21,31]. For these reasons, interesting constraint-based data quality approaches have recently been put forward in the database community. These can be distinguished based on the following three facets:

– *Facet 1: Data-quality rules*. 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, to conditional functional dependencies [22,21], to editing-rules [24] and fixing rules [42], among others.

– *Facet 2: Conflict resolution*. Repairing strategies for constraint languages are based on extra information indicating how to modify the dirty data. In most cases, values are changed into "*preferred*" values. Preferred values can be found from, e.g., master data [36], tuple-certainty and value-accuracy [25], freshness and currency [23], just to name a few.

– *Facet 3: Repair selection*. 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 [9,10] and certain fixes [24]. Alternatively, sampling techniques are put in place to randomly select repairs [9].

We refer to [21] and [31] for recent overviews of constraint-based approaches to data quality. We note, however, that there is currently *no uniform framework* to handle all of the three facets in a flexible and efficient way. To remedy this situation, in this paper we describe a flexible and efficient repairing system referred to as LLUNATIC. A system's overview of LLUNATIC is shown in Figure 1. We provide detailed running examples and descriptions of all ingredients of the system later in the paper. We here briefly highlight the key components of LLUNATIC.

Floris Geerts
University of Antwerp

Giansalvatore Mecca
Università della Basilicata

Paolo Papotti
Eurecom

Donatello Santoro
Università della Basilicata

The LLUNATIC system consists of two core components: ($i$) an *initial labeled instance* and ($ii$) a *disk-based chase engine* over labeled instances.

($i$) The initial labeled instance is a generalization of a standard database instance in which additional information is stored alongside the values in the input dirty instance. This information is provided *by the user at the start of the repairing process*. Intuitively, when conflicts need to be resolved at some point, this information allows inferring the *most preferred way of modifying the data*. More precisely, each location in the database will be adorned with a set of preference labels each consisting of a preference level and a value, where preference levels are elements of a *partial order*. The partial order will allow us to define a notion of *most preferred value* which will be used to resolve a conflict. This value can either be a normal domain value or, when insufficient information is encoded in the partial order, a special value which we refer to as a llun[1] (hence the name LLUNATIC). The use of partial order information is what allows users to *plug-in any kind of preference information* (cfr. Facet 2) into the repairing process. We illustrate in the next sections how information from constraints, master data, user-input, and special attributes (holding preference information) can be encoded in the partial order and initial labeled instance.

($ii$) The second component is the chase engine. A key insight underlying LLUNATIC is that most data-repairing methods behave like the well-known *chase procedure* [2,4], i.e., as long as violations (conflicts) of data-quality constraints exist, some update rules are triggered to resolve these violations. Since we use labeled instances, we *completely overhaul* the standard chase procedure such that it *works on labeled instances*. That is, it uses partial-order information to resolve conflicts. This requires a revision of the formalization of the chase to ensure that it will generate repairs, which we develop in this paper. Furthermore, we provide a *disk-based implementation* of the revised chase procedure, with great attention for optimizations to ensure scalability. To our knowledge, LLUNATIC is the first repairing framework that works with data residing on disk. Furthermore, we allow a fine-grained control of the chase by the user by means of a *cost manager*. This manager allows users to effectively reason about the trade-offs between performance and accuracy and let the chase only generate certain kinds of repairs (cfr. Facet 3).

($iii$) The use of labeled instances and more specifically the partial order information on preference levels allows us to model a variety of data-quality constraint formalisms in a uniform way (cfr. Facet 1). More precisely, LLUNATIC supports variable and constant *equality-generating dependencies* (egds). Constant egds are an extension of classical egds

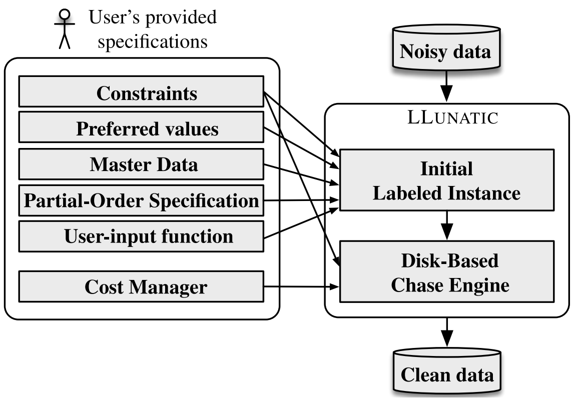


Fig. 1: System overview of LLUNATIC.

[4] which can enforce the presence of certain constants. We provide ample examples of egds and how labeled instances and the chase interact with those in the next sections.

In summary, we propose LLUNATIC as a data repairing framework which addresses all three facets in a flexible way.

We remark that this paper is based on our previous work [26]. Due to the complexity of the formalization used in [26] we believe that some of the key insights behind our approach were unclear. We therefore completely changed the underlying formalization: Everything is now modeled in terms of labeled instances. Not only does this result in a more elegant way of describing how the partial order information is used to resolve conflicts during the chase, it also provides a more clear semantics of repairs. Moreover, labeled instances and the integration of the partial order in the chase process may be of interest in its own right. The current formalization is closer to the standard chase and it is now clearer how our ideas can be adopted in other contexts as well where now only the standard chase is available. We only consider variable and constant egds in this paper. In [27], we extended our approach to more powerful constraints (tuple-generating dependencies, tgds). The labeled instance formalization can be extended to this more general setting but for the sake of clarity of exposition, we do not consider tgds in this paper. We remark that we provide many more details compared to [26, 27] and also include a more extensive experimental evaluation than in our previous work.

**Organization of the paper.** We start with preliminaries in Section 2. Motivation for using labeled instances and the chase for repairing can be found in Section 3. We provide a formalization of our approach in Section 4 and detail the chase procedure in Section 5. How to use LLUNATIC and how ideas from other approaches can be integrated can be found in Sections 5.3 and 6, respectively. Optimizations and implementation details are described in Section 7. In Section 8 we report our experimental findings. Finally, related work and more applications for our partial order are discussed in Sections 9 and 10, respectively.

[1] llun stands for the reverse of null.

$D$(octors)

| Tid | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $c_1$ | $c_{10}$ | $c_{11}$ | $c_{12}$ | $c_{13}$ | $c_{14}$ | $c_{15}$ |
| 1 | 111 | Robert | Chase | surg | 0.9 | PPTH |
| $c_2$ | $c_{20}$ | $c_{21}$ | $c_{22}$ | $c_{23}$ | $c_{24}$ | $c_{25}$ |
| 2 | 111 | Frank | Chase | urol | 0.1 | $\perp_1$ |
| $c_3$ | $c_{30}$ | $c_{31}$ | $c_{32}$ | $c_{33}$ | $c_{34}$ | $c_{35}$ |
| 3 | 222 | $\perp_2$ | House | diag | 1 | $\perp_1$ |

$M$(aster Data)

| Tid | NPI | Name | Surname | Spec | Hospital |
|---|---|---|---|---|---|
| $m$ | 222 | Greg | House | diag | PPTH |

$$e_1 = D(\text{tid},\text{npi},\text{nm},\text{sur},\text{spec},\text{hosp}) \wedge D(\text{tid}',\text{npi},\text{nm}',\text{sur}',\text{spec}',\text{hosp}') \rightarrow \text{nm} = \text{nm}'$$
$$e_2 = D(\text{tid},\text{npi},\text{nm},\text{sur},\text{spec},\text{hosp}) \wedge D(\text{tid}',\text{npi},\text{nm}',\text{sur}',\text{spec}',\text{hosp}') \rightarrow \text{sur} = \text{sur}'$$
$$e_3 = D(\text{tid},\text{npi},\text{nm},\text{sur},\text{spec},\text{hosp}) \wedge D(\text{tid}',\text{npi},\text{nm}',\text{sur}',\text{spec}',\text{hosp}') \rightarrow \text{spec} = \text{spec}'$$
$$e_4 = D(\text{tid},\text{npi},\text{nm},\text{sur},\text{spec},\text{hosp}) \wedge D(\text{tid}',\text{npi},\text{nm}',\text{sur}',\text{spec}',\text{hosp}') \rightarrow \text{hosp} = \text{hosp}'$$
$$e_5 = D(\text{tid},\text{npi},\text{Greg},\text{sur},\text{spec},\text{hosp}) \rightarrow \text{nm} = \text{Gregory}$$
$$e_6 = D(\text{tid},222,\text{nm},\text{sur},\text{spec},\text{hosp}) \rightarrow \text{nm} = \text{Greg}$$
$$e_7 = D(\text{tid},222,\text{nm},\text{sur},\text{spec},\text{hosp}) \rightarrow \text{sur} = \text{House}$$
$$e_8 = D(\text{tid},222,\text{nm},\text{sur},\text{spec},\text{hosp}) \rightarrow \text{spec} = \text{diag}$$
$$e_9 = D(\text{tid},222,\text{nm},\text{sur},\text{spec},\text{hosp}) \rightarrow \text{hosp} = \text{PPTH}$$

Fig. 2: Running example: Instances and equality-generating dependencies.

## 2 Preliminaries

We fix a countably infinite domain of *constant values*, denoted by CONSTS, and a countably infinite set of *labeled nulls*, denoted by NULLS, distinct from CONSTS. Nulls will be denoted by $\perp_0$, $\perp_1$, $\perp_2$, ..., and are used to represent incomplete databases, i.e., databases in which some values are unknown. Furthermore, we fix a countable set, TIDS, of *tuple identifiers*.

**Databases instances and cells.** A *schema* $\mathscr{R}$ is a finite set $\{R_1,\ldots,R_k\}$ of *relation symbols*. For $i \in [1,k]$, the relation symbol $R_i$ in $\mathscr{R}$ has a fixed set of attributes, $\mathsf{Tid}, A_1,\ldots,A_{n_i}$, where $\mathsf{Tid}$ is a special attribute whose domain is TIDS. All other attributes have CONSTS $\cup$ NULLS as domain. For $i \in [1,k]$, an *instance of* $R_i$ is a finite subset $I_i$ of TIDS $\times$ (CONSTS $\cup$ NULLS$)^{n_i}$. A (database) *instance* $\mathbf{I} = (I_1,\ldots,I_k)$ *of* $\mathscr{R}$ consists of instances $I_i$ of $R_i$, for $i \in [1,k]$. A tuple *tuple* $t$ in $\mathbf{I}$ is an element in one of the instances $I_i$ in $\mathbf{I}$. Let $t$ be a tuple in $I_i$. We denote by $t[A_j]$ the value of tuple $t$ in attribute $A_j$. We assume that every tuple in $\mathbf{I}$ has a unique $\mathsf{Tid}$-value. If $t$ is a tuple in $I$ with $t[\mathsf{Tid}] = \text{tid}$, then we also refer to this (unique) tuple by $t_{\text{tid}}$. Given an instance $I$ over $R$, a *cell* in $I$ is a location specified by a tuple id/attribute pair $\langle \text{tid}, A_i\rangle$ (or $\langle \text{tid}, \mathsf{Tid}\rangle$), where tid is an identifier for a tuple in $I$ and $A_i$ is an attribute in $R$. The *value* of a cell $\langle \text{tid}, A_i\rangle$ in $I$ is the value of tuple $t_{\text{tid}}$ in attribute $A_i$, i.e., $t_{\text{tid}}[A_i]$. Similarly, the value of $\langle \text{tid}, \mathsf{Tid}\rangle$ in $I$ is tid. We denote by $\text{cells}(I)$ the set of all *cells* in $I$. Similarly, for an instance $\mathbf{I} = (I_1,\ldots,I_k)$ of $\mathscr{R}$ we define $\text{cells}(\mathbf{I}) = \bigcup\{\text{cells}(I_i) \mid i \in [1,k]\}$.

*Example 1* An instance $I$ of relation $D(\mathsf{Tid},\mathsf{NPI},\mathsf{Name},\mathsf{Surname},\mathsf{Spec},\mathsf{Hospital})$ consisting of tuples $t_1$, $t_2$ and $t_3$ (with ids 1, 2 and 3, respectively) containing information about doctors is shown in Figure 2. Also depicted is a master data instance $J$, containing correct data, of schema $M(\mathsf{Tid},\mathsf{NPI},\mathsf{Name},\mathsf{Surname},\mathsf{Spec},\mathsf{Hospital})$ consisting of a single tuple $t_m$. In instance $I$ we also depict the cells in $\text{cells}(I)$. For cells corresponding to the $\mathsf{Tid}$-attribute, we simply denote the cell by $c_i$, where $i$ is the tuple identifier for the tuple. For example, $c_1 = \langle 1, \mathsf{Tid}\rangle$. For cells corresponding to other attributes, we represent cells by $c_{ij}$. For example, $c_{10} = \langle 1, \mathsf{NPI}\rangle$, $c_{11} = \langle 1, \mathsf{Name}\rangle$, and so on. We ignore the attribute $\mathsf{Conf}$(idence) for the moment. We do not include cells for the master data instance. We will show below that we can eliminate master data instances by means of an encoding in data-quality rules. $\diamondsuit$

**Data-quality rules.** We will use *equality-generating dependencies* (or *egds* for short) to express data-quality rules (constraints). In fact, we use two types of egds: *variable* and *constant* egds. Unlike the seminal paper [4], our egds need not to be typed and can contain constants. A variable egd is defined as follows. A *relational atom over* $\mathscr{R}$ is a formula of the form $R(\bar{s})$ with $R \in \mathscr{R}$ and $\bar{s}$ is a tuple of (not necessarily distinct) constants and variables. Then, a *variable egd over* $\mathscr{R}$ is a formula $e$ of the form $\forall \bar{x}\,(\phi(\bar{x}) \rightarrow x_i = x_j)$, where $\phi(\bar{x})$ is a conjunction of relational atoms over $\mathscr{R}$ with variables $\bar{x}$, and $x_i$ and $x_j$ are variables in $\bar{x}$. A *constant egd* $e$ is of the form $\forall \bar{x}\,(\phi(\bar{x}) \rightarrow x = a)$ where $x$ is a variable in $\bar{x}$ and $a$ is a constant in CONSTS. It is common to write an egd $e$ without writing the universal quantification. So, in what follows $\phi(\bar{x}) \rightarrow x_i = x_j$ corresponds to $\forall \bar{x}\,(\phi(\bar{x}) \rightarrow x_i = x_j)$. Similarly for constant egds.

To prevent any interaction of the egds with the tuple identifiers we assume that: (i) every relation atom $R(\bar{s})$ in $\phi(\bar{x})$ carries a variable in its first position, i.e., $s_1$ is a variable, and this variable does not occur anywhere else in $\bar{s}$ and also does not occur in any other relation atom in $\phi(\bar{x})$; and (ii) if $\phi(\bar{x}) \rightarrow x_i = x_j$ or $\phi(\bar{x}) \rightarrow x = a$, then neither $x_i$, $x_j$ nor $x$ can be a variable that occurs in the first position of a relation atom in $\phi(\bar{x})$. These conditions basically indicate that the egds do not pose constraints on the tuple identifiers.

In the constraint-based data quality approach, dependencies are used to assess the cleanliness of data [7,21,31]. More specifically, an instance $\mathbf{I}$ of $\mathscr{R}$ *satisfies an egd* $e$, variable or constant, denoted by $\mathbf{I} \models e$, if it satisfies $e$ according to satisfaction of first-order logic with (i) a Herbrand interpretation of the constants in $e$, and (ii) the universe of discourse of the first-order structure being CONSTS $\cup$ NULLS (and TIDS for the $\mathsf{Tid}$-attributes) [2]. In particular, nulls are interpreted as constants and as a consequence $\perp = a$ is false for $a \in$ CONSTS. Similarly, $\perp_i = \perp_j$ is false for two different nulls.

*Example 2* Examples of variable egds include functional and (variable) conditional functional dependencies. Constant conditional functional dependencies can be expressed as constant egds. In Figure 2, the variable egds $e_1$–$e_4$ correspond to the functional dependency expressing that attribute NPI is a key for relation $D$ (we again ignore the attribute Conf and also do not take into account the Tid-attribute). Furthermore, the constant egd $e_5$ corresponds to the constant conditional functional dependency which requires the standardization of the name "Greg" into "Gregory". Finally, constant egds $e_6$–$e_9$ originate from a so-called editing rule, which states that any tuple $s$ in $I$ which shares the same NPI-value with a tuple $t$ in the master data $J$ must agree on all common other attributes with $t$. Such editing rules are easily seen to be equivalent to a set of constant egds by introducing a constant egds for each tuple in the master data and each attribute (except for the Tid-attribute) in the master data's schema. In our example, $t_3[\mathsf{NPI}] = t_m[\mathsf{NPI}] = 222$ and the editing rule is translated into the four constant egds $e_6$–$e_9$ basically requiring the four remaining attributes of $t_3$ to take the corresponding values from $t_m$. In this way, master data and editing rules are represented entirely by constant egds. It is readily verified that $I \models \{e_2, e_7, e_8\}$ but $I$ does not satisfy any of the remaining egds. For example, $t_1[\mathsf{Name}] =$ Robert and $t_2[\mathsf{Name}] =$ Frank should be equal according to the egd $e_1$. Hence, $I$ is a dirty instance relative to these egds. $\diamondsuit$

The previous example shows that egds are expressive enough to capture a *wide variety of existing data-quality formalisms*: Functional dependencies [2], conditional functional dependencies [22], and editing rules [24]. Furthermore, one can also verify that egds can express fixing rules (without negative patterns) [42], and certain classes of denial constraints (basically, denial constraints which are logically equivalent to an egd) [7,21]. Motivated by this, we focus on egds. Not supported in LLUNATIC are, for example, matching dependencies [20], metric dependencies [35], differential dependencies [39], and general denial constraints [7,21]. We defer to future work to include a larger variety of data-quality constraints in LLUNATIC.

## 3 LLUNATIC: Finding repairs using the chase

With the constraint formalism fixed, we next turn to the *repairing* or *cleaning* of the data. Intuitively, repairing a dirty instance $\mathbf{I}$ such that $\mathbf{I} \not\models \Sigma$ for some set $\Sigma$ of egds means finding a clean instance $\mathbf{J}$ such that $\mathbf{J} \models \Sigma$ and $\mathbf{I}$ and $\mathbf{J}$ are "closely related". In recent years, repairing methods have been proposed for several classes of constraints. These methods typically consider only specific types of constraints and different interpretations of "closely related". Furthermore, each of these methods differs in how conflicting values (such as Robert and Frank in the previous example) are resolved. We refer to the Related Work Section 9 for more details.

With LLUNATIC we aim to provide a single-node *scalable algorithmic framework for finding repairs* of instances for a set of egds, hereby covering different classes of constraints in a *uniform way*. Moreover, *different conflict resolution strategies* should be *easy to incorporate* in the framework, without the need of changing the underlying algorithm (and thus implementation). To this aim, LLUNATIC uses a *generalization of the standard chase procedure* by incorporating *preference information on values in cells* and by producing a *chase tree* consisting of *chase sequences*, each of which leading to a repair.

We next recall the standard chase procedure and then identify why it needs to be revised in order to become a true workhorse for repairing. In particular, we argue the need for better conflict resolution (Section 3.1), support for constant egds (Section 3.2), backward repairs (Section 3.3) and user repairs (Section 3.4). With the help of a motivating example, we informally introduce the main concepts used in LLUNATIC to address these issues. A formal account will be given in Sections 4 and 5.

**The standard chase.** When $\Sigma$ consists of *variable egds* only, the *chase procedure* (or, simply the *chase*) provides an elegant repairing method [4]. It works as follows. Consider a dirty instance $\mathbf{I} \not\models \Sigma$ and variable egd $e : \phi(\bar{x}) \rightarrow x_i = x_j$ in $\Sigma$. A homomorphism $h$ from $\phi(\bar{x})$ to $\mathbf{I} = (I_1, \ldots, I_k)$ is a mapping which assigns to every variable $x$ in $\bar{x}$ a value in CONSTS$\cup$NULLS (and TIDS for the variables in Tid-attributes) such that every relational atom $R_i(\bar{s})$ in $\phi(\bar{x})$ maps onto a tuple $h(\bar{s}) \in I_i$, where $h$ is the identity on CONSTS. When $h(x_i) \neq h(x_j)$, we say that *$e$ can be applied to $\mathbf{I}$ with homomorphism $h$*. The *result of applying $e$ on $\mathbf{I}$ with $h$* is defined as follows: If $h(x_i)$ and $h(x_j)$ are two different constants in CONSTS, then the result of applying $e$ on $\mathbf{I}$ with $h$ is "failure", and we write $\mathbf{I} \xrightarrow{e,h} \lightning$. Otherwise, the result is a new instance $\mathbf{I}'$, defined as follows. When $h(x_i) \in$ CONSTS and $h(x_j) \in$ NULLS, the null value $h(x_j)$ is replaced everywhere in $\mathbf{I}$ by the constant value $h(x_i)$, resulting in $\mathbf{I}'$. When $h(x_i)$ and $h(x_j)$ are both null values, then one is replaced everywhere by the other, resulting in $\mathbf{I}'$ [2]. In both cases we write $\mathbf{I} \xrightarrow{e,h} \mathbf{I}'$. Then, for a set $\Sigma$ of variable egds, a *chase sequence of* $\mathbf{I}$ *with* $\Sigma$ is a sequence of the form $\mathbf{I}_i \xrightarrow{e_i,h_i} \mathbf{I}_{i+1}$ with $i = 0, 1, \ldots$, $\mathbf{I}_0 = \mathbf{I}$, $e_i \in \Sigma$ and $h_i$ a homomorphism from $e_i$ to $\mathbf{I}_i$. A *finite chase of* $\mathbf{I}$ *with* $\Sigma$ is a finite chase sequence $\mathbf{I}_i \xrightarrow{e_i,h_i} \mathbf{I}_{i+1}$, $i \in [0, m-1]$, such that either $\mathbf{I}_m = \lightning$ or no egd $e$ exists in $\Sigma$ for which there is a homomorphism $h$ such that $e$ can be applied to $\mathbf{I}_m$ with $h$. We call $\mathbf{I}_m$ the *result* of such a finite chase and when $\mathbf{I}_m \neq \lightning$, the instance $\mathbf{I}_m$ is called the

[2] Typically, to make this step deterministic, an ordering on null values is assumed and the smaller null value is replaced by the larger one. We assume that $\bot_0 < \bot_1 < \bot_2 < \bot_3 < \cdots$.

*result of a successful chase*. It is known that if $\mathbf{I}_m$ is the result of a successful chase of $\mathbf{I}$ with $\Sigma$, then $\mathbf{I}_m \models \Sigma$ and $\mathbf{I}_m$ is thus clean [4]. The repair $\mathbf{I}_m$ has many other nice theoretical properties (universality, independence of the order in which egds are applied, ...), see e.g., [19]. Our revised chase does not inherit these properties but our primary goal is using the chase as a practical way of generating repairs.

## 3.1 Avoiding failure by conflict resolution

Clearly, when repairing *data*, the standard chase will often be unsuccessful because different constants may need to be equated. This is not surprising. After all, the chase was originally designed to reason about dependencies, where the input instance does not contain constants, and not for repairing data [4]. As an example, we chase our running example with variable egds.

*Example 3* Consider the variable egds $e_1$, $e_2$, $e_3$ and $e_4$ in Figure 2. Since $I \models e_2$, only $e_1$, $e_3$ and $e_4$ are applicable. It is readily verified that there is a homomorphism $h$ for $e_4$ such that $I \overset{e_4,h}{\to} I'$ with $I'$ obtained from $I$ by replacing the null value $\perp_1$ in $t_2$[Hospital] and $t_3$[Hospital] by PPTH, i.e., the value from $t_1$[Hospital]. However, chasing $I'$ further with $e_1$ and $e_4$ results in a failure. Indeed, $e_1$ requires $t_1$[Name] = Robert to be equal to $t_2$[Name] = Frank which are two different constants. Similarly, $e_4$ requires $t_1$[Spec] = surg to be equal to $t_2$[Spec] = urol. $\diamondsuit$

Instead of simply returning failure ($\lightning$) it is desirable, from a data repairing perspective, for the chase to (i) either report the reasons for failure, or (ii) resolve conflicts between constant values based on some additional information. In LLUNATIC this is achieved as follows. Let $\mathbf{I}$ be a database instance.

– The initial step consists of adorning the cells in $\mathbf{I}$ with *preference levels* from a *partially ordered set* $(\mathbf{P}, \preceq_\mathbf{P})$ and combine these with values in the cells. As a result, each cell initially contains a *preference label* of the form $\langle p, v \rangle$ where $p \in \mathbf{P}$ and $v$ is the value in the cell, resulting in the *initial labeled instance* $\mathbf{I}^\circ$. Preference labels allow comparing values based on their preference levels and the order between these levels according to $\preceq_\mathbf{P}$. We use this information to resolve conflicts by taking the most preferred value, i.e., the value with the highest preference level. The partial order $(\mathbf{P}, \preceq_\mathbf{P})$ is fixed at the beginning of the repairing process. For preference levels $p$ and $p'$, we denote by $p \prec_\mathbf{P} p'$, if $p \preceq_\mathbf{P} p'$ and $p' \not\preceq_\mathbf{P} p$.

*Example 4* In Figure 3a we show the initial labeled instance $I^\circ$ obtained from $I$ (cfr. Figure 2) by putting a single preference level together with its value in each cell of $I$. From here on, we do not depict the attribute Tid since it is only used for identifying tuples and the tuple identifiers do not interact with the egds. In $I^\circ$ we find preference levels $p_{\perp_1}$ and $p_{\perp_2}$ assigned to null values $\perp_1$ and $\perp_2$ in cells $c_{25}$, $c_{35}$ and $c_{31}$, respectively. We also find preference levels $p_{0.1}$, $p_{0.9}$ and $p_1$, associated with urol, surg and diag, in cells $c_{23}$, $c_{13}$ and $c_{33}$, respectively. These preference levels encode the confidence of these values according the Conf attribute. By imposing $p_{0.1} \prec_\mathbf{P} p_{0.9} \prec_\mathbf{P} p_1$ we encode that the higher the confidence is, the more preferred the value associated with these levels in the preference labels is. This is an important example showing how external information (in this case confidence) can be encoded in preference levels and labels. We generalize the use of attributes that encode some ordering (such as Conf) to assign preference levels to values in another attribute (such as Spec) later in Section 4.5 in the form a *partial-order specification*. All other preference levels in $I^\circ$ are chosen arbitrarily, except that we impose that $p_{\perp_1} \prec_\mathbf{P} p_{15}$. This is to indicate that when the value in the preference labels of cells $c_{25}$ and $c_{35}$ (i.e., the value $\perp_1$) needs to be equated later on with the value in the label of $c_{15}$ (i.e., PPTH) due to egd $e_4$, that PPTH is preferred over $\perp_1$. We here capture the semantics that constants are preferred to null values, just as in the standard chase. From the labeled instance $I^\circ$ we can obtain a normal instance $\mathsf{inst}(I^\circ)$ simply by selecting the values in the preference labels in each cell with highest preference value (Figure 3b). In $I^\circ$, each cell carries a single preference label holding the original value of that cell in the instance $I$ given in Figure 2. Hence, $\mathsf{inst}(I^\circ) = I$ in this case. The partial order $\preceq_\mathbf{P}$ used is shown in Figure 4 and comes into play when chasing $I^\circ$. In this partial order we more generally assume that $p_{\perp_i} \prec_\mathbf{P} p$ for any preference level $p_{\perp_i}$ associated with null value $\perp_i$ and any preference value $p$ associated with a constant in $I^\circ$. Furthermore, we also assume that $p_{\perp_0} \prec_\mathbf{P} p_{\perp_1} \prec_\mathbf{P} p_{\perp_2}$ in accordance with the ordering on null values (see earlier footnote). $\diamondsuit$

When chasing an initial labeled instance $\mathbf{I}^\circ$ we will obtain labeled instances $\mathbf{I}^\star$ in which each cell is assigned a *set* of preference labels.

– More precisely, in LLUNATIC we use a *modified chase procedure* which works on labeled instances. Intuitively, whenever a variable egd $e : \phi(\bar{x}) \to x_i = x_j$ applies with a homomorphism $h$, the set of preference labels corresponding to the cells in $h(x_i)$ and $h(x_k)$ are merged. This merging represents that these cells must carry the same value (according to $e$) and that the choice of value should take into account preference level information present in the set of preference labels of all cells involved. If a unique *preferred value*, i.e., a value with maximal preference level exists, that value will be used for repairing and find its way to the standard instance $\mathsf{inst}(\mathbf{I}^\star)$ corresponding to $\mathbf{I}^\star$.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ $\{\langle p_{10}, 111\rangle\}$ | $c_{11}$ $\{\langle p_{11}, \text{Robert}\rangle\}$ | $c_{12}$ $\{\langle p_{12}, \text{Chase}\rangle\}$ | $c_{13}$ $\{\langle p_{0.9}, \text{surg}\rangle\}$ | $c_{14}$ $\{\langle p_{14}, 0.9\rangle\}$ | $c_{15}$ $\{\langle p_{15}, \text{PPTH}\rangle\}$ |
| $t_2$ | $c_{20}$ $\{\langle p_{20}, 111\rangle\}$ | $c_{21}$ $\{\langle p_{21}, \text{Frank}\rangle\}$ | $c_{22}$ $\{\langle p_{22}, \text{Chase}\rangle\}$ | $c_{23}$ $\{\langle p_{0.1}, \text{urol}\rangle\}$ | $c_{24}$ $\{\langle p_{24}, 0.1\rangle\}$ | $c_{25}$ $\{\langle p_{\perp_1}, \perp_1\rangle\}$ |
| $t_3$ | $c_{30}$ $\{\langle p_{30}, 222\rangle\}$ | $c_{31}$ $\{\langle p_{\perp_2}, \perp_2\rangle\}$ | $c_{32}$ $\{\langle p_{32}, \text{House}\rangle\}$ | $c_{33}$ $\{\langle p_1, \text{diag}\rangle\}$ | $c_{34}$ $\{\langle p_{34}, 1\rangle\}$ | $c_{35}$ $\{\langle p_{\perp_1}, \perp_1\rangle\}$ |

(a) Initial labeled instance $I^{\circ}$

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ 111 | $c_{11}$ Robert | $c_{12}$ Chase | $c_{13}$ surg | $c_{14}$ 0.9 | $c_{15}$ PPTH |
| $t_2$ | $c_{20}$ 111 | $c_{21}$ Frank | $c_{22}$ Chase | $c_{23}$ urol | $c_{24}$ 0.1 | $c_{25}$ $\perp_1$ |
| $t_3$ | $c_{30}$ 222 | $c_{31}$ $\perp_2$ | $c_{32}$ House | $c_{33}$ diag | $c_{34}$ 1 | $c_{35}$ $\perp_1$ |

(b) Corresponding instance $\mathsf{inst}(I^{\circ})$.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ $\{\langle p_{10}, 111\rangle\}$ | $c_{11}$ $\{\langle p_{11}, \text{Robert}\rangle, \langle p_{21}, \text{Frank}\rangle\}$ | $c_{12}$ $\{\langle p_{12}, \text{Chase}\rangle\}$ | $c_{13}$ $\{\langle p_{0.9}, \text{surg}\rangle, \langle p_{0.1}, \text{urol}\rangle\}$ | $c_{14}$ $\{\langle p_{14}, 0.9\rangle\}$ | $c_{15}$ $\{\langle p_{15}, \text{PPTH}\rangle, \langle p_{\perp_1}, \perp_1\rangle\}$ |
| $t_2$ | $c_{20}$ $\{\langle p_{20}, 111\rangle\}$ | $c_{21}$ $\{\langle p_{11}, \text{Robert}\rangle, \langle p_{21}, \text{Frank}\rangle\}$ | $c_{22}$ $\{\langle p_{22}, \text{Chase}\rangle\}$ | $c_{23}$ $\{\langle p_{0.9}, \text{surg}\rangle, \langle p_{0.1}, \text{urol}\rangle\}$ | $c_{24}$ $\{\langle p_{24}, 0.1\rangle\}$ | $c_{25}$ $\{\langle p_{15}, \text{PPTH}\rangle, \langle p_{\perp_1}, \perp_1\rangle\}$ |
| $t_3$ | $c_{30}$ $\{\langle p_{30}, 222\rangle\}$ | $c_{31}$ $\{\langle p_{\perp_2}, \perp_2\rangle\}$ | $c_{32}$ $\{\langle p_{32}, \text{House}\rangle\}$ | $c_{33}$ $\{\langle p_1, \text{diag}\rangle\}$ | $c_{34}$ $\{\langle p_{34}, 1\rangle\}$ | $c_{35}$ $\{\langle p_{15}, \text{PPTH}\rangle, \langle p_{\perp_1}, \perp_1\rangle\}$ |

(c) Labeled instance $I_1^{\star}$ chased using variable egds in $\Sigma$.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ 111 | $c_{11}$ $\ell_0$ | $c_{12}$ Chase | $c_{13}$ surg | $c_{14}$ 0.9 | $c_{15}$ PPTH |
| $t_2$ | $c_{20}$ 111 | $c_{21}$ $\ell_0$ | $c_{22}$ Chase | $c_{23}$ surg | $c_{24}$ 0.1 | $c_{25}$ PPTH |
| $t_3$ | $c_{30}$ 222 | $c_{31}$ $\perp_2$ | $c_{32}$ House | $c_{33}$ diag | $c_{34}$ 1 | $c_{35}$ PPTH |

(d) Corresponding instance $\mathsf{inst}(I_1^{\star})$.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ $\{\langle p_{10}, 111\rangle\}$ | $c_{11}$ $\{\langle p_{11}, \text{Robert}\rangle, \langle p_{21}, \text{Frank}\rangle\}$ | $c_{12}$ $\{\langle p_{12}, \text{Chase}\rangle\}$ | $c_{13}$ $\{\langle p_{0.9}, \text{surg}\rangle, \langle p_{0.1}, \text{urol}\rangle\}$ | $c_{14}$ $\{\langle p_{14}, 0.9\rangle\}$ | $c_{15}$ $\{\langle p_{15}, \text{PPTH}\rangle, \langle p_{\perp_1}, \perp_1\rangle, \langle p_{au}, \text{PPTH}\rangle\}$ |
| $t_2$ | $c_{20}$ $\{\langle p_{20}, 111\rangle\}$ | $c_{21}$ $\{\langle p_{11}, \text{Robert}\rangle, \langle p_{21}, \text{Frank}\rangle\}$ | $c_{22}$ $\{\langle p_{22}, \text{Chase}\rangle\}$ | $c_{23}$ $\{\langle p_{0.9}, \text{surg}\rangle, \langle p_{0.1}, \text{urol}\rangle\}$ | $c_{24}$ $\{\langle p_{24}, 0.1\rangle\}$ | $c_{25}$ $\{\langle p_{15}, \text{PPTH}\rangle, \langle p_{\perp_1}, \perp_1\rangle\}$ |
| $t_3$ | $c_{30}$ $\{\langle p_{30}, 222\rangle\}$ | $c_{31}$ $\{\langle p_{\perp_2}, \perp_2\rangle, \langle p_{au}, \text{Gregory}\rangle, \langle p_{au}, \text{Greg}\rangle\}$ | $c_{32}$ $\{\langle p_{32}, \text{House}\rangle\}$ | $c_{33}$ $\{\langle p_1, \text{diag}\rangle\}$ | $c_{34}$ $\{\langle p_{34}, 1\rangle\}$ | $c_{35}$ $\{\langle p_{15}, \text{PPTH}\rangle, \langle p_{\perp_1}, \perp_1\rangle\}$ |

(e) Labeled instance $I_2^{\star}$ chased using $\Sigma$.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ 111 | $c_{11}$ $\ell_0$ | $c_{12}$ Chase | $c_{13}$ surg | $c_{14}$ 0.9 | $c_{15}$ PPTH |
| $t_2$ | $c_{20}$ 111 | $c_{21}$ $\ell_0$ | $c_{22}$ Chase | $c_{23}$ surg | $c_{24}$ 0.1 | $c_{25}$ PPTH |
| $t_3$ | $c_{30}$ 222 | $c_{31}$ $\ell_1$ | $c_{32}$ House | $c_{33}$ diag | $c_{34}$ 1 | $c_{35}$ PPTH |

(f) Corresponding instance $\mathsf{inst}(I_2^{\star})$.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ $\{\langle p_{10}, 111\rangle\}$ | $c_{11}$ $\{\langle p_{11}, \text{Robert}\rangle\}$ | $c_{12}$ $\{\langle p_{12}, \text{Chase}\rangle\}$ | $c_{13}$ $\{\langle p_{0.9}, \text{surg}\rangle\}$ | $c_{14}$ $\{\langle p_{14}, 0.9\rangle\}$ | $c_{15}$ $\{\langle p_{15}, \text{PPTH}\rangle\}$ |
| $t_2$ | $c_{20}$ $\{\langle p_{20}, 111\rangle, \langle p_{\times}, \times\rangle\}$ | $c_{21}$ $\{\langle p_{21}, \text{Frank}\rangle\}$ | $c_{22}$ $\{\langle p_{22}, \text{Chase}\rangle\}$ | $c_{23}$ $\{\langle p_{0.9}, \text{uorl}\rangle\}$ | $c_{24}$ $\{\langle p_{24}, 0.1\rangle\}$ | $c_{25}$ $\{\langle p_{\perp_1}, \perp_1\rangle, \langle p_{au}, \text{PPTH}\rangle\}$ |
| $t_3$ | $c_{30}$ $\{\langle p_{30}, 222\rangle\}$ | $c_{31}$ $\{\langle p_{\perp_2}, \perp_2\rangle, \langle p_{au}, \text{Gregory}\rangle, \langle p_{au}, \text{Greg}\rangle\}$ | $c_{32}$ $\{\langle p_{32}, \text{House}\rangle\}$ | $c_{33}$ $\{\langle p_1, \text{diag}\rangle\}$ | $c_{34}$ $\{\langle p_{34}, 1\rangle\}$ | $c_{35}$ $\{\langle p_{\perp_1}, \perp_1\rangle, \langle p_{au}, \text{PPTH}\rangle\}$ |

(g) Labeled instance $I_3^{\star}$ chased using $\Sigma$ with a backward repair.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ 111 | $c_{11}$ Robert | $c_{12}$ Chase | $c_{13}$ surg | $c_{14}$ 0.9 | $c_{15}$ PPTH |
| $t_2$ | $c_{20}$ $\ell_2$ | $c_{21}$ Frank | $c_{22}$ Chase | $c_{23}$ urol | $c_{24}$ 0.1 | $c_{25}$ PPTH |
| $t_3$ | $c_{30}$ 222 | $c_{31}$ $\ell_1$ | $c_{32}$ House | $c_{33}$ diag | $c_{34}$ 1 | $c_{35}$ PPTH |

(h) Corresponding instance $\mathsf{inst}(I_3^{\star})$.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ $\{\langle p_{10}, 111\rangle\}$ | $c_{11}$ $\{\langle p_{11}, \text{Robert}\rangle\}$ | $c_{12}$ $\{\langle p_{12}, \text{Chase}\rangle\}$ | $c_{13}$ $\{\langle p_{0.9}, \text{surg}\rangle\}$ | $c_{14}$ $\{\langle p_{14}, 0.9\rangle\}$ | $c_{15}$ $\{\langle p_{15}, \text{PPTH}\rangle\}$ |
| $t_2$ | $c_{20}$ $\{\langle p_{20}, 111\rangle, \langle p_{\times}, \times\rangle, \langle p_{\top}, 112\rangle\}$ | $c_{21}$ $\{\langle p_{21}, \text{Frank}\rangle\}$ | $c_{22}$ $\{\langle p_{22}, \text{Chase}\rangle\}$ | $c_{23}$ $\{\langle p_{0.9}, \text{uorl}\rangle\}$ | $c_{24}$ $\{\langle p_{24}, 0.1\rangle\}$ | $c_{25}$ $\{\langle p_{\perp_1}, \perp_1\rangle$ $\{\langle p_{au}, \text{PPTH}\rangle\}$ |
| $t_3$ | $c_{30}$ $\{\langle p_{30}, 222\rangle\}$ | $c_{31}$ $\{\langle p_{\perp_2}, \perp_2\rangle, \langle p_{au}, \text{Gregory}\rangle, \langle p_{au}, \text{Greg}\rangle, \langle p_{\top}, \text{Gregory}\rangle\}$ | $c_{32}$ $\{\langle p_{32}, \text{House}\rangle,$ | $c_{33}$ $\{\langle p_1, \text{diag}\rangle,$ | $c_{34}$ $\{\langle p_{34}, 1\rangle\}$ | $c_{35}$ $\{\langle p_{\perp_1}, \perp_1\rangle, \langle p_{aut}, \text{PPTH}\rangle\}$ |

(i) Labeled instance $I_4^{\star}$ chased with $\Sigma$ and a user repair.

$D$(octors)

| | NPI | Name | Surname | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ 111 | $c_{11}$ Robert | $c_{12}$ Chase | $c_{13}$ surg | $c_{14}$ 0.9 | $c_{15}$ PPTH |
| $t_2$ | $c_{20}$ 112 | $c_{21}$ Frank | $c_{22}$ Chase | $c_{23}$ urol | $c_{24}$ 0.1 | $c_{25}$ PPTH |
| $t_3$ | $c_{30}$ 222 | $c_{31}$ Gregory | $c_{32}$ House | $c_{33}$ diag | $c_{34}$ 1 | $c_{35}$ PPTH |

(j) Corresponding instance $\mathsf{inst}(I_4^{\star})$.

Fig. 3: Running example: Labeled instances and their corresponding (standard) instances at different times during the LLUNATIC chase process.

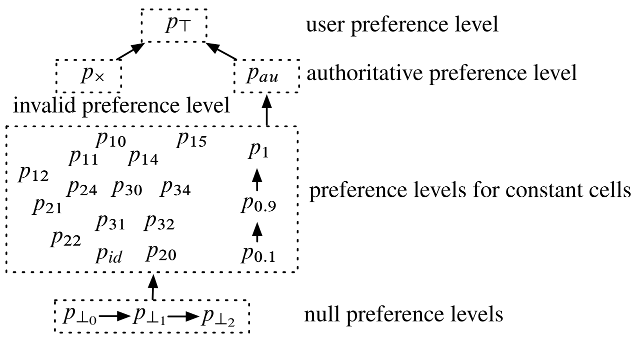

$p_\top$
user preference level
$p_\times$
$p_{au}$
authoritative preference level
invalid preference level
$p_{10}$ $p_{15}$ $p_{11}$ $p_{14}$ $p_{12}$ $p_{24}$ $p_{30}$ $p_{34}$ $p_{21}$ $p_{31}$ $p_{32}$ $p_{22}$ $p_{id}$ $p_{20}$
$p_1$ $p_{0.9}$ $p_{0.1}$
preference levels for constant cells
$p_{\perp_0}$ $p_{\perp_1}$ $p_{\perp_2}$
null preference levels

Fig. 4: Partial order $\preceq_{\mathbf{P}}$ on preference levels used in our running example. Arrows between preference levels denote $\prec_{\mathbf{P}}$ (strictly less). No arrow means incomparable. Arrows between dotted boxes mean that all levels in one box are strictly less preferred than those in the other box.

– Moreover, when a set of preference labels of a cell does contains different preferred values (distinct maximal preference levels), this implies that not enough information is present to resolve the conflict. Instead of returning failure, we mark such cells in $\mathsf{inst}(\mathbf{I}^\star)$ with a *special constant* which we refer to as a *llun*.

*Example 5* We chase the initial labeled instance $I^\circ$ by our revised chase procedure. The result labeled instance $I_1^\star$ of this chase, using the variable egds in $\Sigma$ is shown in Figure 3c. As an example, $t_1[\mathsf{Spec}]=$ surg and $t_2[\mathsf{Spec}]=$ urol need to be equated, due to egd $e_3$. The chase will consider the preference labels in $I^\circ$ of the cells $c_{13}$ and $c_{23}$ and merge these, resulting in the set of preference labels $\{\langle p_{0.1},\text{urol}\rangle, \langle p_{0.9},\text{surg}\rangle\}$. To resolve the conflict between $t_1[\mathsf{Spec}]=$ surg and $t_2[\mathsf{Spec}]=$ urol, one finds that "surg" has a higher preference level ($p_{0.9}$) than "urol" ($p_{0.1}$) according to $\preceq_{\mathbf{P}}$. This implies that when turning the labeled instance $I_1^\star$ back into a normal instance $\mathsf{inst}(I_1^\star)$, the cells $c_{12}$ and $c_{23}$ will carry value "surg", as shown in Figure 3d.

We next focus on preference level $p_{15}$ in $\langle p_{15},\text{PPTH}\rangle$ in cell $c_{15}$. In the standard chase, PPTH will replace both occurrence of $\perp_1$ in $I$, in cells $c_{25}$ and $c_{35}$. The revised chase will again merge preference labels in $I^\circ$ for the cells $c_{15}$, $c_{25}$ and $c_{35}$ resulting in $\{\langle p_{15},\text{PPTH}\rangle, \langle p_{\perp_1},\perp_1\rangle\}$ as shown in Figure 3c. Since in our partial order (Figure 4) $p_{\perp_1} \prec_{\mathbf{P}} p_{15}$, in instance $\mathsf{inst}(I_1^\star)$ we select PPTH as the most preferred value for $c_{15}$, $c_{25}$ and $c_{35}$ as shown in Figure 3d. So indeed, nulls are less preferred than constants when they need to be equated, just as in the standard chase.

To illustrate the use of lluns, we consider the set of preference labels $\{\langle p_{11},\text{Robert}\rangle, \langle p_{21},\text{Frank}\rangle\}$ in $I_1^\star$ obtained by merging the preference labels of cells $c_{11}$ and $c_{21}$ to satisfy $e_2$. Here, we do not have a most preferred value as $\preceq_{\mathbf{P}}$ does not have information about how the preference levels $p_{11}$ and $p_{21}$ compare (cfr. Figure 4). In this case, in $\mathsf{inst}(I_1^\star)$, these cells are populated by a fresh llun value $\ell_0$ to indicate that there is a conflict as shown in Figure 3d.

It is easily verified that $\mathsf{inst}(I_1^\star)$ satisfies the variable egds in $\Sigma$. $\diamondsuit$

## 3.2 Supporting constant egds

The next issue is that the standard chase does not support constant egds. These are, however, crucial to model popular data-quality rule formalisms such as conditional functional dependencies and editing rules. In LLUNATIC, we benefit from the use of labeled instances to revise the chase so that it works with constant egds, as follows:

– We reserve a special *authoritative preference level*, $p_{\mathsf{au}}$, to indicate *authoritative values*. By imposing that $p_{\mathsf{au}}$ is more preferred according to $\preceq_{\mathbf{P}}$ than most other preference levels (except for the invalid and user preference level to be introduced below), whenever a preference label $\langle p_{\mathsf{au}},v\rangle$ is present in a set of preference labels, the value $v$ will be picked.
– In the revised chase, when a constant egds $e:\phi(\bar{x})\to x=a$ can be applied with a homomorphism $h$, we simply put $\langle p_{\mathsf{au}},a\rangle$ in the preference labels of all cells related to $h(x)$. This is to indicate that there are constraints (constant egds) that indicate which value particular cells preferably should take.

This once more shows the usefulness of working with labeled instance and the partial order $\preceq_{\mathbf{P}}$.

*Example 6* The labeled instance $I_2^\star$ shown in Figure 3e reflects the situation after chasing the labeled instance $I_1^\star$ from the previous example with the constant egds in $\Sigma$. For example, the cell $c_{31}$ now holds preference labels $\{\langle p_{\perp_2},\perp_2\rangle, \langle p_{\mathsf{au}},\text{Gregory}\rangle, \langle p_{\mathsf{au}},\text{Greg}\rangle\}$ due to the application of the constant egds $e_5$ and $e_6$. Since these constant egds are *inconsistent* with each other, i.e., they require to apply inconsistent changes, we find two values (Gregory and Greg) both with the authorative preference level $p_{\mathsf{au}}$. When moving from the labeled instance $I_2^\star$ to the normal instance $\mathsf{inst}(I_2^\star)$ shown in Figure 3f, we end up in a situation in which no single most preferred value exists (both Greg and Gregory are candidates) and hence $c_{31}$ obtains a llun value $\ell_1$. This illustrates that LLUNATIC will still generate a repair even when the egds are in conflict with each other. Suppose that only $e_5$ would be present in $\Sigma$ then $c_{31}$ would have ended up with preference labels $\{\langle p_{\perp_2},\perp_2\rangle, \langle p_{\mathsf{au}},\text{Gregory}\rangle\}$ and the value Gregory would be selected in $\mathsf{inst}(I_2^\star)$. In Figure 4, we show how $p_{\mathsf{au}}$ relates to other preference levels in $(\mathbf{P},\preceq_{\mathbf{P}})$. $\diamondsuit$

### 3.3 Backward repairing

So far, when chasing with egds we always enforced that the antecedent of an egd was satisfied. Another way is to *invalidate the premise* of an egd, i.e., performing a so-called *backward repair*. Again, labeled instances and preference levels make it easy to incorporate such backward repairs.

- In the revised chase, whenever an egd $e : \phi(\bar{x}) \to x_i = x_j$ or $e : \phi(\bar{x}) \to x = a$ applies with a homomorphism $h$, we perform a backward repair by introducing a special *invalid preference label*, $\langle p_\times, \times\rangle$, into the set of preference labels of cells on which relation atoms in $\phi(\bar{x})$ are mapped into by $h$. We refer to $p_\times$ as the *invalid preference level* and to $\times$ as an invalid value (we assume that $\times$ is in CONSTS). By positioning $p_\times$ in $\preceq_{\mathbf{P}}$ such that it is incomparable to most other preference levels, we can force cells to take a llun value indicating that the original value was incorrect.

*Example 7* Consider the labeled instance $I_3^\star$ shown in Figure 3g. It is obtained by enforcing the constant egds, as before, and by invaliding the value 111 in cell $c_{20}$ by inserting $\langle p_\times, \times\rangle$ into its set of preference labels. Intuitively, we are ensuring that none of the variable egds apply (backward repair). Assuming that $p_\times$ and $p_{20}$ are incomparable, in $\mathsf{inst}(I_3^\star)$ the cell $c_{20}$ will obtain a llun value $\ell_2$ since no single most preferred value exists in $\{\langle p_{20}, 111\rangle, \langle p_\times, \times\rangle\}$. Clearly, $\mathsf{inst}(I_3^\star)$ satisfies all egds in $\Sigma$. We also note that although the constant egd $e_9$ only applies to tuple $t_3$, when the cell $c_{35}$ obtains preference label $\langle p_{\mathsf{au}}, \text{PPTH}\rangle$, also $c_{25}$ obtains this preference labels. This is in accordance with how the standard chase replaces all occurrences of the same null value with constant. How $p_\times$ relates to other preference levels is shown in Figure 4. $\diamond$

### 3.4 User repairs

A final important aspect is the incorporation of user knowledge in the repairing process. In LLUNATIC we allow users to change the value of sets of preference labels and solve any incompleteness or conflict marked by lluns, as follows.

- We reserve a special *user preference level*, $p_\top$, such that when $\langle p_\top, v\rangle$ is present in a set of preference labels then this implies that the user specified the "correct" value $v$. User interaction gracefully embeds in our revised chase as it suffices to add the user provided label $\langle p_\top, v\rangle$ to the preference labels of relevant cells. The preference level $p_\top$ will be the maximal level in our partial order.

*Example 8* Consider labeled instance $I_4^\star$ shown in Figure 3i which is obtained from $I_3^\star$ by injecting two pieces of information from a user: $t_2[\mathsf{NPI}]$ should be 112 and $t_3[\mathsf{Name}]$ should be Gregory. This is represented in $I_4^\star$ by inserting $\langle p_\top, 112\rangle$ in the preference labels of cell $c_{20}$ and inserting $\langle p_\top, \text{Gregory}\rangle$ in the preference labels of $c_{31}$. Since $p_\top$ is the most preferred preference level, $\mathsf{inst}(I_4^\star)$ will carry value 112 (instead $\ell_2$ in $\mathsf{inst}(I_3^\star)$) and Gregory (instead of $\ell_1$ in $\mathsf{inst}(I_3^\star)$) in the corresponding cells. How $p_\top$ relates to other preference levels is shown in Figure 4. $\diamond$

In summary, by using labeled instances, preference levels and labels, and revising the chase, we obtain a flexible mechanism of repairing data for sets of egds.

## 4 The formalization underlying LLUNATIC

Having informally described the main concepts underlying LLUNATIC in the previous section, we next formalize labeled instances and how to go from a standard instance to a labeled instance and back (Section 4.1), what it means for an egd to be satisfied on a labeled instance (Section 4.2) as this is needed to understand the semantics, introduce user functions (Section 4.3) and finally define when one labeled instance is an upgrade of another labeled instance (Section 4.4) as this will enable us to link repairs obtained from our revised chase procedure to the original dirty instance.

### 4.1 Labeled instances

Given an instance $\mathbf{I}$, a labeled instance assigns a *set* of *preference labels* to each cell in $\mathbf{I}$. Preference labels consist of values taken from CONSTS or NULLS (and TIDS for the Tid-attributes), together with *preference levels*. We model preference levels by values taken from a partially-ordered, countable set $\mathbf{P}$. The partial order on $\mathbf{P}$ is denoted by $\preceq_{\mathbf{P}}$ and reflects how different preference levels compare with each other. For two preference levels $p$ and $p'$ in $\mathbf{P}$, we denote by $p \prec_{\mathbf{P}} p'$ if $p \preceq_{\mathbf{P}} p'$ and $p' \not\preceq_{\mathbf{P}} p$.

**Definition 1** A *preference label* over $\mathbf{P}$ is a pair $\langle p, v\rangle$, where $p$ is a *preference level* in $\mathbf{P}$, and $v$ is a value from CONSTS $\cup$ NULLS (and TIDS for the Tid-attributes). A *labeled instance* $\mathbf{I}^\star$ *over* $\mathbf{P}$ *of instance* $\mathbf{I}$ is a mapping that associates a *non-empty* finite set of preference labels over $\mathbf{P}$ with *each cell* $c$ in $\mathbf{I}$, denoted by $\mathbf{I}^\star(c)$. $\square$

Intuitively, in a labeled instance $\mathbf{I}^\star$ all cells in $\mathbf{I}$ come equipped with a set of preference labels indicating possible values that can be put in the cells. Instances $I^\circ$, $I_1^\star$, $I_2^\star$, $I_3^\star$ and $I_4^\star$ in Figure 3 are examples of labeled instances over $\mathbf{P}$, shown in Figure 4, of instance $I$ given in Figure 2. As an example, $I_1^\star(c_{21}) = \{\langle p_{11}, \text{Robert}\rangle, \langle p_{21}, \text{Frank}\rangle\}$.

In the LLUNATIC framework everything starts by inspecting the dirty instance $\mathbf{I}$ and (i) extract preference levels $p_c$ for the values in *all* cells $c$ in $\mathbf{I}$; (ii) extract partial order information about these preference levels, to form $(\mathbf{P}, \preceq_{\mathbf{P}})$; and

(iii) create an initial labeled instance based on this information. In Example 4 we have put arbitrary preference levels $p_{10}$, $p_{11}$, $p_{12}$, $p_{14}$, $p_{15}$, $p_{20}$, $p_{21}$, $p_{22}$, $p_{24}$, $p_{30}$, $p_{32}$, $p_{34}$ in cells for which we have no further information, $p_{\perp_1}$ and $p_{\perp_2}$ for cells containing null values, and $p_{0.1}$, $p_{0.9}$, $p_1$ for cells for which confidence information was available. For cells $c_i = \langle i, \mathsf{Tid}\rangle$ we always assign preference label $\langle p_{id}, i\rangle$ for some arbitrary fixed preference level $p_{id}$.

These preference levels relate to each other by $\preceq_{\mathbf{P}}$ as shown in Figure 4. We come back to this important initialization phase in more detail in Section 4.5. For now, we assume that $(\mathbf{P}, \preceq_{\mathbf{P}})$ is given and assume that for each cell $c$ of $\mathbf{I}$ we have a preference level $p_c \in \mathbf{P}$ associated with it. Given this, we can easily associate a labeled instance to a normal instance.

**Definition 2** The *initial labeled instance* $\mathbf{I}^\circ$ over $\mathbf{P}$ of $\mathbf{I}$ is defined as the labeled instance in which each cell $c = \langle \text{tid}, A_i\rangle$ of $\mathbf{I}$ is assigned $\langle p_c, v\rangle$ where $v = t_{\text{tid}}[A_i]$, that is, $\mathbf{I}^\circ(c) = \{\langle p_c, v\rangle\}$. $\square$

The labeled instance $I^\circ$ depicted in Figure 3a is obtained from the instance $I$ and $(\mathbf{P}, \preceq_{\mathbf{P}})$ shown in Figure 4.

We also associate a unique (standard) instance to a labeled instance by leveraging the partial order information in the preference labels. More specifically, we associate with a set of preference labels a unique value. Intuitively, this value is the value associated with the "highest" preference level among all preference labels. When no such unique value exists, we assign it a special constant value, which we refer to as a *llun*. More specifically, we denote by LLUNS $= \{\ell_0, \ell_1, \ell_2, \ldots\}$, an infinite set of constants, disjoint from CONSTS, TIDS and NULLS. These constants are used to solve conflicts. That is, when the correct value of a cell is currently unknown, we mark it by a llun so that it might be resolved later on into a constant, e.g., by asking for user input. Lluns allow us to always infer a unique value for a set of preference labels.

**Definition 3** Given a set of preference labels $\mathscr{L} = \{\langle p_1, v_1\rangle, \ldots, \langle p_k, v_k\rangle\}$ over $\mathbf{P}$, the *preferred value* of $\mathscr{L}$, denoted by pval$(\mathscr{L})$, is obtained as follows. Consider the set $M$ of maximal elements in $\mathscr{L}$ according to $\preceq_{\mathbf{P}}$, i.e., the set of all $\langle p, v\rangle \in \mathscr{L}$ such that there exists no $\langle p', v'\rangle \in \mathscr{L}$ for which $p \prec_{\mathbf{P}} p'$ holds. Then:

1. if all preference labels in $M$ have exactly the same value $v$, then pval$(\mathscr{L}) = v$;
2. otherwise pval$(\mathscr{L})$ is a fresh llun value in LLUNS. $\square$

For example, for $\mathscr{L} = \{\langle p_{0.1}, \text{urol}\rangle, \langle p_{0.9}, \text{surg}\rangle\}$ in $I_1^\star$ we have that pval$(\mathscr{L}) = $ surg because $p_{0.1} \prec_{\mathbf{P}} p_{0.9}$. By contrast, for $\mathscr{L} = \{\langle p_{11}, \text{Robert}\rangle, \langle p_{21}, \text{Frank}\rangle\}$ in $I_1^\star$, we have that pval$(\mathscr{L}) = \ell_1 \in$ LLUNS because $p_{11}$ and $p_{21}$ are incomparable according to $\preceq_{\mathbf{P}}$. Moreover, pval$(\{\langle p_{id}, i\rangle\})$ is always $i$ for cells $\langle i, \mathsf{Tid}\rangle$. With this notion in place, we can now assign a unique standard instance to a labeled instance.

**Definition 4** Given a labeled instance $\mathbf{I}^\star$ over $\mathbf{P}$ of $\mathbf{I}$, we define the *instance associated with* $\mathbf{I}^\star$ as the standard instance, denoted by $\mathsf{inst}(\mathbf{I}^\star)$, obtained by assigning each cell $c$ in $\mathbf{I}$ the value pval$(\mathbf{I}^\star(c))$. We note that $\mathsf{inst}(\mathbf{I}^\star)$ consists of relations taking values from CONSTS $\cup$ NULLS $\cup$ LLUNS (and TIDS for the $\mathsf{Tid}$-attributes). $\square$

In Figure 3 we have shown various labeled instances and their associated instances. As a sanity check, we observe that $\mathsf{inst}(\mathbf{I}^\circ) = \mathbf{I}$. That is, the standard instance associated with the initial labeled instance of $\mathbf{I}$ is $\mathbf{I}$ itself. This holds because in $\mathbf{I}^\circ$, every cell is associated with a single preference label and pval$\{\langle p, v\rangle\} = v$ with $v$ the value of the cell in $\mathbf{I}$.

## 4.2 Satisfaction of egds for labeled instances.

We next define what it means for a labeled instance to satisfy an egd. We distinguish between variable and constant egds. For variable egds, we simply use the standard notion of satisfaction of first-order logic [3] on the instance associated with a labeled instance.

**Definition 5** Given a variable egd $e : \phi(\bar{x}) \to x_i = x_j$, an instance $\mathbf{I}$ and a labeled instance $\mathbf{I}^\star$ over $\mathbf{P}$ of $\mathbf{I}$, we say that $\mathbf{I}^\star$ *satisfies* $e$, denoted $\mathbf{I}^\star \models_\ell e$, if $\mathsf{inst}(\mathbf{I}^\star) \models e$. $\square$

The motivation behind this definition is as follow. Let $\mathbf{I}^\star = (I_1^\star, \ldots, I_k^\star)$ be a labeled instance of $\mathbf{I} = (I_1, \ldots, I_k)$, $e$ be a variable egd $\phi(\bar{x}) \to x_i = x_j$, and let $\mathsf{inst}(\mathbf{I}^\star) = (\mathsf{inst}(I_1^\star), \ldots, \mathsf{inst}(I_k^\star))$ be the instance associated with $\mathbf{I}^\star$. We next associate cells with homomorphisms and variables of $\phi(\bar{x})$.

**Definition 6** Let $h$ be a homomorphism from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{I}^\star)$ and let $x$ be a variable in $\phi(\bar{x})$. The *set of cells associated with $x$ and $h$*, denoted by cells$_h(x)$, is the smallest subset of cells in cells$(\mathbf{I})$ such that for every atom $R_i(\bar{s})$ in $\phi(x)$, if $x$ occurs at position $j$ in $\bar{s}$, then cells$_h(x)$ contains $\langle tid, A_j\rangle$ where $tid$ is the tuple identifier of the tuple $h(\bar{s}) \in \mathsf{inst}(I_i^\star)$ and $A_j$ is the attribute corresponding to position $j$ in $R_i$. $\square$

If $\mathsf{inst}(\mathbf{I}^\star) \models e$ for a variable egd $e$, then $h(x_i) = h(x_j)$ for any homomorphism $h$ from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{I}^\star)$. This in turn implies that for any pair of cells $c$ and $c'$ in cells$_h(x_i)$ and cells$_h(x_j)$, the preferred value of $\mathbf{I}^\star(c)$ and $\mathbf{I}^\star(c')$ agree, which is precisely what $e$ demands. It is easily verified that the labeled instances $I_2^\star$, $I_3^\star$, $I_4^\star$ in Figure 3 satisfy all variable egds in $\Sigma$ because their associated instances $\mathsf{inst}(I_2^\star)$, $\mathsf{inst}(I_3^\star)$ and $\mathsf{inst}(I_4^\star)$ do.

Satisfaction of constant egds is defined differently. Consider a constant egd $e : \phi(\bar{x}) \to x = a$. If, as in the variable egd case, $\mathsf{inst}(\mathbf{I}^\star) \models e$ then we also say that $\mathbf{I}^\star$ satisfied $e$,

[3] Of course, here the universe of discourse of the first-order structure being CONSTS $\cup$ NULLS $\cup$ LLUNS (and TIDS for the $\mathsf{Tid}$-attributes). Similarly to constants and nulls, lluns are treated as constants.

denoted by $\mathbf{I}^\star \models_\ell e$. We also consider another way for a constant egd to be satisfied, as is explained next. Consider the set of cells $\text{cells}_h(x)$. We want to ensure that the set of preference labels in $\mathbf{I}^\star$ associated with cells in $\text{cells}_h(x)$ carry information that the constant $a$ is preferred as described by the constant egd. As previously explained, to this aim we introduce a special *authoritative* preference level $p_{\mathsf{au}}$ in $\mathbf{P}$ and preference label $\langle p_{\mathsf{au}}, a\rangle$ where $a$ is the constant in the constant egd $e$.

**Definition 7** Given a constant egd $e : \phi(\bar{x}) \rightarrow x = a$, an instance $\mathbf{I}$ and a labeled instance $\mathbf{I}^\star$ over $\mathbf{P}$ of $\mathbf{I}$, we say that $\mathbf{I}^\star$ *satisfies* $e$, denoted $\mathbf{I}^\star \models_\ell e$, if either $\mathsf{inst}(\mathbf{I}^\star) \models e$, or for every homomorphism $h$ from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{I}^\star)$, for every cell $c \in \text{cells}_h(x)$, $\mathbf{I}^\star(c)$ contains $\langle p_{\mathsf{au}}, a\rangle$. $\square$

By positioning $p_{\mathsf{au}}$ in the partial order $\preceq_{\mathbf{P}}$, the preference of $p_{\mathsf{au}}$ compared with other preference levels can be adjusted. See, for example $\preceq_{\mathbf{P}}$ in Figure 4. In this way, when $\mathbf{I}^\star \models_\ell e$ holds, the preferred values used to obtain $\mathsf{inst}(\mathbf{I}^\star)$ took into account that a constant egd required certain cells to have a specific constant value, despite that the preferred value of the cell's preference labels may not agree with constants required by the constant egd.

In Figure 3, the labeled instance $I_2^\star$ satisfies the constant egds $e_7$–$e_9$ because $\mathsf{inst}(I_2^\star)$ does so. Furthermore, $e_5$ and $e_6$ are satisfied by $I_2^\star$ because $I_2^\star(c_{31}) = \{\langle p_{\perp_2}, \perp_2\rangle, \langle p_{\mathsf{au}}, \text{Greg}\rangle, \langle p_{\mathsf{au}}, \text{Gregory}\rangle\}$ and hence $I_2^\star$ has encoded that $e_5$ tells that the value should be "Greg" and $e_6$ tells that the value should be "Gregory". Note, however, that $\mathsf{inst}(I_2^\star)$ does not satisfy $e_5$ and $e_6$ since the conflicting information is resolved by a llun value $\ell_1$. Similarly, the labeled instances $I_3^\star$ and $I_4^\star$ satisfies all constant egds in $\Sigma$. We note that the labeled instance $I_1^\star$ does not satisfy $e_5$.

Given a set $\Sigma$ of egds, a labeled instance $\mathbf{I}^\star$ over $\mathbf{P}$ of $\mathbf{I}$ *satisfies* $\Sigma$, denoted $\mathbf{I}^\star \models_\ell \Sigma$, if $\mathbf{I}^\star \models_\ell e$ for all $e \in \Sigma$.

**Definition 8** A labeled instance $\mathbf{I}^\star$ over $\mathbf{P}$ of $\mathbf{I}$ is *clean* relative to a set $\Sigma$ of egds when $\mathbf{I}^\star \models_\ell \Sigma$. It is called dirty, otherwise. $\square$

From our earlier observations it now follows that $I_2^\star$, $I_3^\star$ and $I_4^\star$ in Figure 3 are all clean, and $I^\circ$ and $I_1^\star$ are dirty.

## 4.3 User functions

Labeled instances provide an excellent formalism for dealing with user corrections. In LLUNATIC, we abstract user inputs by seeing the user as an oracle over sets of preference labels. Such an oracle represents the ultimate way to change the preferred value for a cell with a given set of labels.

**Definition 9** We call a *user-input function* a (computable) partial function User that takes as input any set of preference labels, $\mathscr{L}$, and returns a value $v \in \textsc{consts}$, denoted by $\mathsf{User}(\mathscr{L})$, to indicate that the clean value of any cell annotated with $\mathscr{L}$ is $v$. $\square$

Note that User is by definition a partial function, and it may thus be undefined for some sets of labels. User-specified clean values will be encoded by means of special *user* preference label $\langle p_\top, v\rangle$ and we require, when $\mathsf{User}(\mathscr{L}) = v$ is defined, that $\mathscr{L}$ is always extended with $\langle p_\top, v\rangle$.

**Definition 10** A labeled instance $\mathbf{I}^*$ over $\mathbf{P}$ of $\mathbf{I}$ is said to be *user-corrected according* to User if there exists *no* cell $c \in \text{cells}(\mathbf{I})$ such that for some $v \in \textsc{consts}$, $\mathsf{User}(\mathbf{I}^\star(c)) = v$ but $\langle p_\top, v\rangle \notin \mathbf{I}^\star(c)$. $\square$

Similarly to the level $p_{\mathsf{au}}$, we can adjust the preference of $p_\top$ compared with other preference values in $\mathbf{P}$ by positioning $p_\top$ in the partial order $\preceq_{\mathbf{P}}$ of $\mathbf{P}$. See, for example $\preceq_{\mathbf{P}}$ in Figure 4. It is now easy to see that the labeled instance $I_4^\star$ in Figure 3 is user-corrected according to the user-input function given by $\mathsf{User}(\{\langle p_{20}, 111\rangle, \langle p_\times, \times\rangle\}) = 112$, $\mathsf{User}(\{\langle p_{\perp_2}, \perp_2\rangle, \langle p_{\mathsf{au}}, \text{Greg}\rangle, \langle p_{\mathsf{au}}, \text{Gregory}\rangle\}) = \text{Gregory}$.

## 4.4 Upgrades and repairs

Given a dirty instance $\mathbf{I}$, a set $\Sigma$ of egds and a user-input function User, we will consider labeled instances $\mathbf{I}^*$ that are (i) clean relative to $\Sigma$; and (ii) user-corrected according to User. What is missing from the picture is how such labeled instances $\mathbf{I}^*$ are related to $\mathbf{I}$. We formalise this using the notion of *upgrade*.

We start from $\mathbf{I}$, consider the initial labeled instance $\mathbf{I}^\circ$ and now want to assess whether a labeled instance $\mathbf{I}^\star$ of $\mathbf{I}$ is of better "quality" than $\mathbf{I}^\circ$. More generally, we want to compare two labeled instances in terms of the information stored in their preference labels. Intuitively, a set of preference labels is of higher quality than another set of of preference labels when it contains at least the same preference labels. This lifts to labeled instances in a natural way.

**Definition 11** Given labeled instances $\mathbf{I}_1^\star$ and $\mathbf{I}_2^\star$ over $\mathbf{P}$ of $\mathbf{I}$, we say that $\mathbf{I}_1^\star$ *upgrades* $\mathbf{I}_2^\star$, denoted by $\mathbf{I}_2^\star \preceq \mathbf{I}_1^\star$, if for each cell $c$ of $\mathbf{I}$, it is the case that the set of labels of cell $c$ in $\mathbf{I}_1^\star$ contains the set of labels of $c$ in $\mathbf{I}_2^\star$, i.e., $\mathbf{I}_2^\star(c) \subseteq \mathbf{I}_1^\star(c)$. We say that $\mathbf{I}_1^\star$ *strictly upgrades* $\mathbf{I}_2^\star$, denoted by $\mathbf{I}_2^\star \prec \mathbf{I}_2^\star$, if $\mathbf{I}_1^\star \preceq \mathbf{I}_2^\star$ and $\mathbf{I}_2^\star \not\preceq \mathbf{I}_1^\star$. $\square$

Indeed, upgrades capture our intended semantics in that values are replaced by more preferred values. Intuitively, whenever instance $\mathbf{I}_1^\star$ upgrades $\mathbf{I}_2^\star$, then, for each cell $c$ of $\mathbf{I}$, it must be the case that the value assigned to $c$ in $\mathsf{inst}(\mathbf{I}_1^\star)$ is more (or equally) preferred over the corresponding value assigned in $\mathsf{inst}(\mathbf{I}_2^\star)$.

A labeled instance $\mathbf{I}^\star$ over $\mathbf{P}$ of $\mathbf{I}$ is said to *upgrade instance* $\mathbf{I}$ if it upgrades the initial labeled instance of $\mathbf{I}^\circ$, i.e.,

$\mathbf{I}^\circ \preceq \mathbf{I}^\star$. Similarly, if $\mathbf{I}^\circ \prec \mathbf{I}^\star$ holds, then the labeled instance $\mathbf{I}^\star$ is said to be a strict upgrade of $\mathbf{I}$.

In Figure 3, we see by simply checking containment of the sets of preference labels in cells that $I^\circ \prec I_1^\star \prec I_2^\star$ and $I^\circ \prec I_3^\star \prec I_4^\star$. We note, however, that $I_1^\star$ and $I_3^\star$ are incomparable. We are now finally ready to define what we mean by a repair.

**Definition 12** A *repair* $\mathbf{I}^\star$ of $\mathbf{I}$ is a labeled instance over $\mathbf{P}$ of $\mathbf{I}$ which is (i) clean relative to $\Sigma$; (ii) user-corrected according to User; and (iii) is an upgrade of $\mathbf{I}$. Moreover, $\mathbf{I}^\star$ is a minimal repair if any other repair $\mathbf{K}^\star$ of $\mathbf{I}$ satisfies $\mathbf{I}^\star \preceq \mathbf{K}^\star$. □

In Figure 3 only the labeled instance $I_4^\star$ is a repair of $I$ for the egds in $\Sigma$ and user-input function User given earlier.

The computational challenge is now to compute such repairs. We do this by means of a revised chase procedure on labeled instances. Before explaining the LLUNATIC chase, we provide some more information how to extract the initial labeled instance from a dirty instance, as this initial labeled instance will form the starting point of the chase.

## 4.5 Partial-order specification

From the discussion so far, it should be clear that the preference levels and preference labels in the initial labeled instance are *fixed up front* and are used later on to select the preferred value from a set of preference labels. Although any partially ordered set $(\mathbf{P}, \preceq_\mathbf{P})$ could be used (as long as it supports $p_\times$, $p_{\mathsf{au}}$ and $p_\top$), in practical settings we assume that $(\mathbf{P}, \preceq_\mathbf{P})$ is structured as in Figure 4. More precisely, we assume that $\mathbf{P}$:

- contains a null preference level $p_{\bot_i}$ for each null value $\bot_i$ in NULLS, such that $p_{\bot_0} \prec_\mathbf{P} p_{\bot_1} \prec_\mathbf{P} p_{\bot_2} \prec_\mathbf{P} \cdots$, and furthermore $p_{\bot_i} \prec_\mathbf{P} p$ for any other $p \in \mathbf{P} \setminus \{p_\times\}$;
- is such that for any $p \in \mathbf{P} \setminus \{p_{\mathsf{au}}, p_\top\}$, $p \prec_\mathbf{P} p_{\mathsf{au}}$, i.e., the authoritative preference level is higher than any other preference level, except for the user preference level $p_\top$;
- for any $p \in \mathbf{P} \setminus \{p_{\mathsf{au}}, p_\top\}$, $p$ is incomparable with the invalid preference level $p_\times$; and
- for any $p \in \mathbf{P} \setminus \{p_\top\}$, $p \prec_\mathbf{P} p_\top$, i.e., the user preference level trumps any other preference level.

This to ensure that the preference levels $p_{\bot_i}$, $p_\times$, $p_{\mathsf{au}}$ and $p_\top$ have the desired effect when present in a set of preference labels. We also assume $p_{\bot_i}$ only to be present in the preference label $\langle p_{\bot_i}, \bot_i\rangle$ and $p_\times$ in $\langle p_\times, \times\rangle$.

We next describe how a user can create the initial labeled instance. More precisely, in the initial labeled instance $\mathbf{I}^\circ$ over $\mathbf{P}$ of $\mathbf{I}$, one initalizes

- for any cell $c = \langle tid, A_i\rangle$ in $\mathbf{I}$ such that $t_{tid}[A_i] = \bot_j$:
$$\mathbf{I}^\circ(c) := \{\langle p_{\bot_j}, \bot_j\rangle\},$$
- and for all other cells $c = \langle tid, A_i\rangle$ in $\mathbf{I}$ such that $t_{tid}[A_i] = v \in$ CONSTS:
$$\mathbf{I}^\circ(c) := \{\langle p_c, v\rangle\},$$
where $p_c \in \mathbf{P} \setminus \{p_{\bot_i}, p_\times, p_{\mathsf{au}}, p_\top\}$. Further inspection of the data is needed to select these preference levels $p_c$ and fixing their relationship in the partial order.

To better understand what we mean here, just recall how cells $c_{13}$ and $c_{23}$ in $I$ in our running example were labeled in $I^\circ$ by preference labels $\langle p_{0.9}, \text{surg}\rangle$ and $\langle p_{0.1}, \text{urol}\rangle$, respectively, such that $p_{0.1} \prec_\mathbf{P} p_{0.9}$, based on the confidence information stored in the Conf attribute. Hence, when later on a conflict between “surg” and “urol” needed to be resolved, “surg” will be the most preferred value and be used to resolve the conflict. In principle there is no restriction on how the preference levels $p_c$ relate to each other, however, we next describe a *practical way of extracting partial order information* on preference levels $p_c$ associated with the constant values in cells in $\mathbf{I}$.

We propose the use of ordering attributes. An *ordering attribute $A$* in $\mathscr{R}$ is such that tuples $t$ in instances $\mathbf{I}$ of $\mathscr{R}$ have values $t[A]$ coming from a domain *equipped* with a natural partial order. For example, Conf is an ordering attribute over the rational numbers. Other examples are time-stamp attributes, or other numerical attributes. We then define a *partial-order specification* as a partial function $\sqcap$ from the set of attributes in $\mathscr{R}$ to the set of ordering attributes in $\mathscr{R}$. For example, in our running example $\sqcap$ maps attribute Spec to Conf in relation $D$. Although Spec by itself is not an ordering attribute, the partial-order specification $\sqcap$ can now be used to extract partial-order information on preference levels in $\mathbf{P}$ for cells related to Spec.

In general, consider an instance $\mathbf{I}$ of $\mathscr{R}$ and cells $c_1 = \langle tid_1, A\rangle$ and $c_2 = \langle tid_2, A\rangle$ in $\mathbf{I}$. Let $p_{c_1}$ and $p_{c_2}$ be two new preference levels in $\mathbf{P}$ used to create the initial instance $\mathbf{I}^\circ$. That is, in the initial labeled instance $\mathbf{I}^\circ$ we have that $\mathbf{I}^\circ(c_1) = \{\langle p_{c_1}, t_{tid_1}[A]\rangle\}$ and $\mathbf{I}^\circ(c_2) = \{\langle p_{c_2}, t_{tid_2}[A]\rangle\}$. We then define

$$p_{c_1} \preceq_\mathbf{P} p_{c_2} \text{if and only if } t_{tid_1}[\sqcap(A)] \leq t_{tid_2}[\sqcap(A)].$$

That is, we order $p_{c_1}$ and $p_{c_2}$ in $\preceq_\mathbf{P}$ in accordance to the (ordered) attribute values $t_{tid_1}[\sqcap(A)]$ and $t_{tid_2}[\sqcap(A)]$ (recall that $c_1$ and $c_2$ are cells in the $A$ attribute in $\mathbf{I}$.)

In this way we can use temporal information (e.g., timestamps) to give certain cell values higher preference in $\mathbf{I}^\circ$ based on their date of creation. We can also add an ordering attribute as part of a preprocessing step which records the frequency of values in another attribute. In $\mathbf{I}^\circ$ we then give more preference to frequent values. We tie the use of partial-order specification to other repairing methods in Section 6. We emphasize that this is only one of the possible ways of specifying the desired partial order, and by no means it is the most general. Yet, we believe that it represents a good compromise between simplicity and expressiveness.

## 5 The LLUNATIC chase

As anticipated, we now revise the standard chase procedure such that it works on labeled instances and generates repairs according to Definition 12. Intuitively, starting from the initial labeled instance $\mathbf{I}^{\circ}$ over $\mathbf{P}$ of $\mathbf{I}$, in each step of the chase we generate an upgrade by either merging sets of preference labels or extending sets of preference labels. By contrast to the standard chase, the revised chase produces a *chase tree*, i.e., a tree in which each branch corresponds to a different chase sequence. To guarantee that the chase ends after a finite number of steps we do impose a restriction on the labeled instances (upgrades) that can be generated by the chase. More precisely, for a given set $\mathscr{L}$ of preference labels, the set of all cells having $\mathscr{L}$ as their set of preference labels in a labeled instance $\mathbf{I}^{\star}$, is referred to as the *cell group of $\mathscr{L}$ in $\mathbf{I}^{\star}$. We will change all cells in the same cell group in the same way.* For example, in an initial labeled instance $\mathbf{I}^{\circ}$ all cells having $\langle p_{\perp_i}, \perp_i\rangle$ as their preference label will be in the same cell group. We have remarked earlier that these should indeed be changed in the same way, in accordance with the standard chase. As another example, when the chase merges two sets of preference labels, for example in cells $c_{11}$ and $c_{12}$ in $I_1^{\star}$ in Figure 3, this implies that these two cells should carry the same preferred value. By putting these cells in the same cell group, we guarantee that this is preserved during further chase steps. Typically, cells will belong to the same cell group if a previous application of an egd required the two cells to carry the same information. We next detail the chase steps (Section 5.1) and then describe the result of chase and some of its properties (Section 5.2).

### 5.1 Chase steps

Let $\mathbf{I}^{\star}$ be a labeled instance over $\mathbf{P}$ of $\mathbf{I}$ and consider the corresponding instance $\mathsf{inst}(\mathbf{I}^{\star})$. Let $e : \phi(\bar{x}) \to x_i = x_j$ or $e : \phi(\bar{x}) \to x = a$ be a variable and constant egd, respectively. We first define what it means for an egd $e$ to be applicable to $\mathbf{I}^{\star}$. Let $h$ be a homomorphism from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{I}^{\star})$. We associate, similar to Definition 6, cells with variables and constants in $\phi(\bar{x})$. A more fine-grained association is needed for the backward chase step (see below) since we have to be able to distinguish between cells corresponding to *different occurrences of the same variable* in $\phi(\bar{x})$ and also need to identify cells corresponding to *constants* in $\phi(\bar{x})$.

**Definition 13** Consider an homomorphism $h$ from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{I}^{\star}) = (\mathsf{inst}(I_1^{\star}), \dots, \mathsf{inst}(I_k^{\star}))$. Let $F = R_i(\bar{s}) = R_i(tid, s_1, \dots, s_{n_i})$ be an atom in $\phi(\bar{x})$ and let $j \in [1, n_i + 1]$. We define the *cell associated with h, atom F and position j*, denoted by $\mathrm{cell}_h(F, j)$, as the (single) cell $\langle tid, A_j\rangle$, where *tid* is the tuple identifier of the tuple $h(\bar{s}) \in \mathsf{inst}(I_i^{\star})$ and $A_j$ is the attribute name of position $j$ in $R_i$. $\square$

We remark that $\mathrm{cells}_h(x)$ (cfr. Definition 6) is just the union of $\mathrm{cell}_h(F, j)$ where $F$ and $j$ range over all atoms $F$ in $\phi(x)$ containing $x$ at a position $j$.

We further expand $\mathrm{cell}_h(F, j)$ by the cells in its cell group, i.e., we define

$$\overline{\mathrm{cell}}_h(F, j) := \{c' \in \mathrm{cells}(\mathsf{inst}(\mathbf{I}^{\star})) \mid \mathbf{I}^{\star}(c') = \mathbf{I}^{\star}(\mathrm{cell}_h(F, j))\}.$$

Intuitively, $\overline{\mathrm{cell}}_h(F, j)$ contains all cells that need to be changed in the same way as $\mathrm{cell}_h(F, j)$ during the chase, as remarked earlier. The value to which all cells in $\overline{\mathrm{cell}}_h(F, j)$ need to be changed is determined by $\mathbf{I}^{\star}(\mathrm{cell}_h(F, j))$.

We can lift these definitions to variables $x$ in $\phi(x)$ in a natural way. More precisely, $\overline{\mathrm{cells}}_h(x)$ is the union of all $\overline{\mathrm{cell}}_h(F, j)$ where, as before, $F$ and $j$ range over all atoms $F$ in $\phi(x)$ containing $x$ at position $j$. The value to which all cells in $\overline{\mathrm{cells}}_h(x)$ need to be changed is determined by the union of the set preference labels associated to cells in $\overline{\mathrm{cells}}_h(x)$. We denote this union by $\mathscr{L}_h(x)$ which is the union of $\mathbf{I}^{\star}(\mathrm{cell}_h(F, j))$ where $F$ and $j$ range over all atoms $F$ in $\phi(x)$ containing $x$ at position $j$.

Then, when $e$ is a variable egd, we say that *$e$ can be applied to $\mathbf{I}^{\star}$ with homomorphism $h$* when $h(x_i) \neq h(x_j)$. When $e$ is a constant egd, we say that *$e$ can be applied to $\mathbf{I}^{\star}$ with homomorphism $h$* when either $h(x) \neq a$ or when $\langle p_{\mathsf{au}}, a\rangle$ is not part of $\mathscr{L}_h(x)$. These conditions basically check whether $\mathbf{I}^{\star} \not\models_{\ell} e$ (cfr. Section 4.2). We now define the *result of applying $e$ on $\mathbf{I}^{\star}$ with homomorphism $h$* as a new labeled instance $\mathbf{J}^{\star}$ over $\mathbf{P}$ of $\mathbf{I}$ defined as follows.

**Forward chase step variable egd:** In this case, $\mathbf{I}^{\star}$ and $\mathbf{J}^{\star}$ agree on all cells in $\mathbf{I}$ except for those corresponding to $\overline{\mathrm{cells}}_h(x_i)$ and $\overline{\mathrm{cells}}_h(x_j)$. More precisely, for all cells $c \in \overline{\mathrm{cells}}_h(x_i) \cup \overline{\mathrm{cells}}_h(x_j)$ we define

$$\mathbf{J}^{\star}(c) := \mathscr{L}_h(x_i) \cup \mathscr{L}_h(x_j).$$

In other words, we merge all sets of preference labels of cells related to $h(x_i)$ and $h(x_j)$.

**Forward chase step constant egd:** In this case, $\mathbf{I}^{\star}$ and $\mathbf{J}^{\star}$ agree on all cells in $\mathbf{I}$ except for those corresponding to $\overline{\mathrm{cells}}_h(x)$. More precisely, for all cells $c \in \overline{\mathrm{cells}}_h(x)$ we define

$$\mathbf{J}^{\star}(c) := \mathscr{L}_h(x) \cup \{\langle p_{\mathsf{au}}, a\rangle\}.$$

In both cases we write $\mathbf{I}^{\star} \xrightarrow{e,h} \mathbf{J}^{\star}$. It is readily verified that there exist homomorphisms $h_1, h_2, h_3$ such that $I^{\circ} \xrightarrow{e_1,h_1} J_1^{\star} \xrightarrow{e_3,h_2} J_2^{\star} \xrightarrow{e_4,h_3} I_1^{\star}$ for the labeled instances $I^{\circ}$ and $I_1^{\star}$ and variable egds in $\Sigma$ given in Figure 3. Furthermore, there exist homomorphisms $h_4$ and $h_5$ such that $I_1^{\star} \xrightarrow{e_5,h_4} J_3^{\star} \xrightarrow{e_6,h_5} I_2^{\star}$ for $I_2^{\star}$ and constant egds in $\Sigma$ in Figure 3.

**Backward chase step egd:** We want to create a labeled instance $\mathbf{J}^{\star}$ over $\mathbf{P}$ of $\mathbf{I}$ such that, when $e$ is applicable to $\mathbf{I}^{\star}$ with a homomorphism $h$ from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{I}^{\star})$, then $h$ is not a homomorphism anymore from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{J}^{\star})$.

We distinguish between the following two cases, depending on whether we "disable" a constant or an equality between variables in $\phi(\bar{x})$. Let $F = R_i(\bar{s}) = R_i(tid, s_1, \ldots, s_{n_i})$ be an atom in $\phi(x)$. Suppose that $j \in [1, n_i]$ is such that $s_j \in$ CONSTS. We then ensure that $\mathbf{J}^\star(c)$, for all $c \in \overline{cell_h}(F, j)$, contains the invalid preference label $\langle p_\times, \times\rangle$. In other words, for such cells $c$, $\mathsf{inst}(\mathbf{J}^\star)$ will not hold constant value $s_j$ anymore, ensuring that $h(\bar{s}) \notin \mathsf{inst}(\mathbf{J}^\star)$. More precisely, we create a new labeled instance $\mathbf{J}^\star$ over $\mathbf{P}$ of $\mathbf{I}$ which agrees with $\mathbf{I}^\star$ on all cells in $\mathbf{I}$ except for those in $\overline{cell_h}(F, j)$. For cells $c$ in $\overline{cell_h}(F, j)$ we define

$$\mathbf{J}^\star(c) := \mathbf{I}^\star(c) \cup \{\langle p_\times, \times\rangle\},$$

as just explained.

For the second case, let $x$ be a variable in $\phi(x)$ that occurs multiple times in $\phi(\bar{x})$. If no such variable exists, then this case does not apply. Consider relation atoms $F = R_i(\bar{s})$ and $F' = R_j(\bar{s}')$ in $\phi$ and assume that $s_\ell = x$ and $s'_{\ell'} = x$. When $F = F'$, then we must have that $\ell \neq \ell'$.

Let $h$ be a homomorphism from $\phi(\bar{x})$ to $\mathsf{inst}(\mathbf{I}^\star)$ and consider $c = \mathrm{cell}_h(F, \ell)$ and $c' = \mathrm{cell}_h(F', \ell)$. We only backward chase when $\mathrm{pval}(\mathbf{I}^\star(c)) = \mathrm{pval}(\mathbf{I}^\star(c'))$ is a constant *and* $I^\star(c) \neq I^\star(c')$. Here, the second condition implies that $\overline{\mathrm{cell}}_h(F, \ell)$ and $\overline{\mathrm{cell}}_h(F', \ell')$ are disjoint. We then create a new labeled instance $\mathbf{J}^\star$ over $\mathbf{P}$ of $\mathbf{I}$ which agrees with $\mathbf{I}^\star$ on all cells in $\mathbf{I}$ except for those in either $\overline{\mathrm{cell}}_h(F, \ell)$ or $\overline{\mathrm{cell}}_h(F', \ell')$. In one of those sets of cells, we will add to all cells the invalid preference label $\langle p_\times, \times\rangle$ to their set of preference labels. More precisely, say that we pick $\overline{\mathrm{cell}}_h(F, \ell)$. Then for all $c \in \overline{\mathrm{cell}}_h(F, \ell)$ we define

$$\mathbf{J}^\star(c) = \mathbf{I}^\star(c) \cup \{\langle p_\times, \times\rangle\}.$$

The reason to restrict the application of backward chase steps when $\mathrm{pval}(\mathbf{I}^\star(c)) = \mathrm{pval}(\mathbf{I}^\star(c'))$ is a constant is that we feel that invalidating a null or a llun does not make sense semantically. (Of course, in general one may allow this.)

In both cases (disable constant, disable equality) we write $\mathbf{I}^\star \xrightarrow{e,h,F,\ell} \mathbf{J}^\star$ indicating which atom ($F$) and position ($j$) in $\phi(\bar{x})$ we invalidate.

As an example, consider variable egd $e_1 : D(tid, \mathrm{npi}, \mathrm{nm}, \mathrm{sur}, \mathrm{spec}, \mathrm{hosp}) \wedge D(tid', \mathrm{npi}, \mathrm{nm}', \mathrm{sur}', \mathrm{spec}', \mathrm{hosp}') \rightarrow \mathrm{nm} = \mathrm{nm}'$. Both these atoms, let us denote them by $F$ and $F'$, have variable npi at position 2. It is now readily verified that there exist homomorphisms $h_1$, $h_2$, $h_3$ and $h_4$ such that $I^\circ \xrightarrow{e_1,h_1,F,2} J_1^\star \xrightarrow{e_5,h_2} J_2^\star \xrightarrow{e_6,h_3} J_3^\star \xrightarrow{e_9,h_4} I_3^\star$ for $I_3^\star$ and egds in $\Sigma$ in Figure 3.

**User chase step:** When a user-input function User is given, we say that this function is *applicable on* $\mathbf{I}^\star$ when there are cells $c$ such that $\mathsf{User}(\mathbf{I}^\star(c)) = v$ is defined. In this case we create a new labeled instance $\mathbf{J}^\star$ over $\mathbf{P}$ of $\mathbf{I}$ which agrees with $\mathbf{I}^\star$ on all cells in $\mathbf{I}$ except for those cells $c$ in which $\mathsf{User}(\mathbf{I}^\star(c))$ is defined. More precisely, for all such cells $c$ we define

$$\mathbf{J}^\star(c) := \mathbf{I}^\star(c) \cup \{\langle p_\top, \mathsf{User}(\mathbf{I}^\star(c))\rangle\}.$$

In other words, we add $\mathsf{User}(\mathbf{I}^\star(c)) = v$ together with the user preference level $p_\top$ to the set of preference labels. Note that this step changes all cells in the same cell group in the same way. Indeed, cells in a cell group have the same set of preference labels. We write $\mathbf{I}^\star \xrightarrow{\mathsf{User},\mathscr{L}} \mathbf{J}^\star$ where $\mathscr{L} = \mathbf{I}^\star(c)$ for which this chase step is applied. As an example, we have that $I_3^\star \xrightarrow{\mathsf{User},\mathscr{L}_1} J_1^\star \xrightarrow{\mathsf{User},\mathscr{L}_2} I_4^\star$ for $I_4^\star$ shown in Figure 3, where $\mathscr{L}_1 = \{\langle p_{20}, 111\rangle, \langle p_\times, \times\rangle\}$, $\mathscr{L}_2 = \{\langle p_{\perp_2}, \perp_2\rangle, \langle p_{\mathsf{au}}, \mathrm{Greg}\rangle, \langle p_{\mathsf{au}}, \mathrm{Gregory}\rangle\}$, and User is the user-input function given earlier.

## 5.2 The LLUNATIC chase and its properties

Given a set $\Sigma$ of egds, constant or variable, a user-input function User and a labeled instance $\mathbf{I}^\star$ over $\mathbf{P}$ of $\mathbf{I}$, a *chase sequence of* $\mathbf{I}^\star$ *with* $\Sigma$ *and* User is a sequence of labeled instances $\mathbf{I}_i^\star$ with $i = 0, 1, \ldots$, such that $\mathbf{I}_0^\star = \mathbf{I}^\star$ and for every $i$, either $\mathbf{I}_i^\star \xrightarrow{e,h} \mathbf{I}_{i+1}^\star$ (forward step), $\mathbf{I}_i^\star \xrightarrow{e,h,F,j} \mathbf{I}_{i+1}^\star$ (backward step), or $\mathbf{I}_i^\star \xrightarrow{\mathsf{User},\mathscr{L}_i} \mathbf{I}_{i+1}^\star$ (user step). The *chase tree* $\mathbf{I}^\star$ *with* $\Sigma$ *and* User, denoted by $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^*)$, is a tree whose root is $\mathbf{I}^*$ and all branches correspond to finite chase sequences of $\mathbf{I}^\star$ with $\Sigma$ and User such that no further chase steps can be applied to the last labeled instance in the sequence. We note that our chase steps *never return failure* $\lightning$.

We next show that every branch in the chase is a finite chase sequence and that the leaves of the chase are repairs.

**Theorem 1** *Given a labeled instance* $\mathbf{I}^\star$ *over* $\mathbf{P}$ *of* $\mathbf{I}$*, a set* $\Sigma$ *of egds and user-input function* User*. Then, every chase sequence in* $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^*)$ *is finite and furthermore, every labeled instance in a leaf of* $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^*)$ *is a repair of* $\mathbf{I}^\star$*.*

*Proof* To show that every chase sequence in $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^*)$ is finite, it suffices to observe that every chase step, whether it is a forward, a backward or a user chase step, either strictly increases the size of cell groups (cells that carry the same set of preference labels), or strictly increases the size of sets of preference labels. There is clearly an upper bound on how many times cell groups can be expanded as in the worst case all cells in an instance belong to the same cell group. Similarly, since $\Sigma$ contains a finite number of constant egds, the number of times the corresponding authoritative preference level can be added is bounded. The same holds for the invalid preference level and user preference level. Hence every chase sequence is bounded in length. From the definition of chase steps, it is clear that when no further chase steps can be executed on a labeled instance, it satisfies all egds in $\Sigma$ and is user-corrected according to User. Furthermore, by definition, a chase step from a labeled instance $\mathbf{I}^\star_i$ to a labeled instance $\mathbf{I}^\star_{i+1}$ ensures that $\mathbf{I}^\star_{i+1}$ is an upgrade of $\mathbf{I}^\star_i$. Hence, every leaf in $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^*)$ is a repair of $\mathbf{I}^\star$. $\square$

In LLUNATIC, we will of course apply the chase on the initial labeled instance $\mathbf{I}^\circ$ of a dirty instance $\mathbf{I}$. As an example, $I^\star_4$ in Figure 3 is a repair generated by $\mathsf{chase}_{\Sigma,\mathsf{User}}(I^\circ)$. We return both $I^\star_4$ and its corresponding instance $\mathsf{inst}(I^\star_4)$ to the user.

Furthermore, some properties of the standard chase carry over to our revised chase.

**Theorem 2** *Given an initial labeled instance* $\mathbf{I}^\circ$ *over* $\mathbf{P}$ *of* $\mathbf{I}$*, a set* $\Sigma$ *of egds and user-input function* User*, we have that:*

- *the number of repairs in the leaves in* $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^\circ)$ *is at most exponential in the size* $|\mathbf{I}|$ *of* $\mathbf{I}$*; and*
- *every chase sequence in* $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^\circ)$ *is of length polynomial in the size* $|\mathbf{I}|$ *of* $\mathbf{I}$*,*

*where* $|\mathbf{I}|$ *is the number of tuples in* $\mathbf{I}$ *and where, as usual, we consider the schema, the set of constraints and user-input function to be fixed (data complexity).*

*Proof* Repairs are obtained from $\mathbf{I}^\circ$ by expanding the set of preference labels associated with cells in $\mathbf{I}$. For each cell, the possible set of preference labels is bounded by the constants appearing in $\mathbf{I}$ and preference levels in $\mathbf{I}^\circ$ (together with the special levels $p_\times$, $p_{\mathsf{au}}$ and $p_\top$). Hence, there are most $2^{\mathscr{O}(|\mathbf{I}|)}$ different sets of preference labels and since the number of cells is bounded by $\mathscr{O}(|\mathbf{I}|)$ (recall the schema is fixed), we have at most $2^{\mathscr{O}(|\mathbf{I}|)}$ possible labeled instances over $\mathbf{P}$ of $\mathbf{I}$ which upgrade $\mathbf{I}^\circ$. Consequently, the leaves of $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^\circ)$ represent at most an exponential number of repairs of $\mathbf{I}$.

To see that every chase sequence in $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^\circ)$ is of length polynomial in the size $|\mathbf{I}|$ of $\mathbf{I}$, we have seen in the proof of the previous theorem that each step either strictly increases the size of cell groups (cells that carry the same set of preference labels), or strictly increases the size of sets of preference labels. One can now associate an integer-valued function $f$ to labeled instances based on the size of cell groups and sizes of sets of preference labels such that $f(\mathbf{J}_1) < f(\mathbf{J}_2)$ when $\mathbf{J}_2$ is the result of a chase step on $\mathbf{J}_1$. It suffices now to observe that $f$ is bounded by $\mathscr{O}(|\mathbf{I}|)$ on every upgrade generated in a chase sequence in $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^\circ)$. Hence, such a chase sequence must of length bounded by $\mathscr{O}(|\mathbf{I}|)$. $\square$

We conclude this section by observing that the LLUNATIC chase can be seen as a *conservative extension* of the standard chase. Indeed, we instantiate $\mathbf{P}$ by preference levels for nulls and assign a unique preference level for each cell carrying a constant value in $\mathbf{I}$. We relate these to the null preference levels as before. By redefining $\mathsf{pval}(\mathscr{L})$ such that it returns $\lightning$ (instead of a llun) when no single preferred value can be obtained from $\mathscr{L}$, then it is readily verified that the LLUNATIC chase on $\mathbf{I}^\circ$ coincides with the standard chase when no backward chase steps are performed.

## 5.3 LLUNATIC in Action

We now show how a user would interact with the chase while performing data cleaning with our LLUNATIC open-source data-repairing system[4]. The GUI of the LLUNATIC system is reported in Figure 5. Any experience with the system starts by specifying a scenario with at least one non empty database. Among the databases, users may indicate some that are considered as *authoritative* – like master data. Databases can be browsed to inspect the data, as illustrated in (*frame (1)* for dataset “cust”. Ordering attributes can be specified by selecting columns from any relation.

The next step is concerned with specifying the data quality constraints. These can be specified in a declarative form with logical formulas, but the system also provides a graphical user interface for this task, as reported in *frame (2)*.

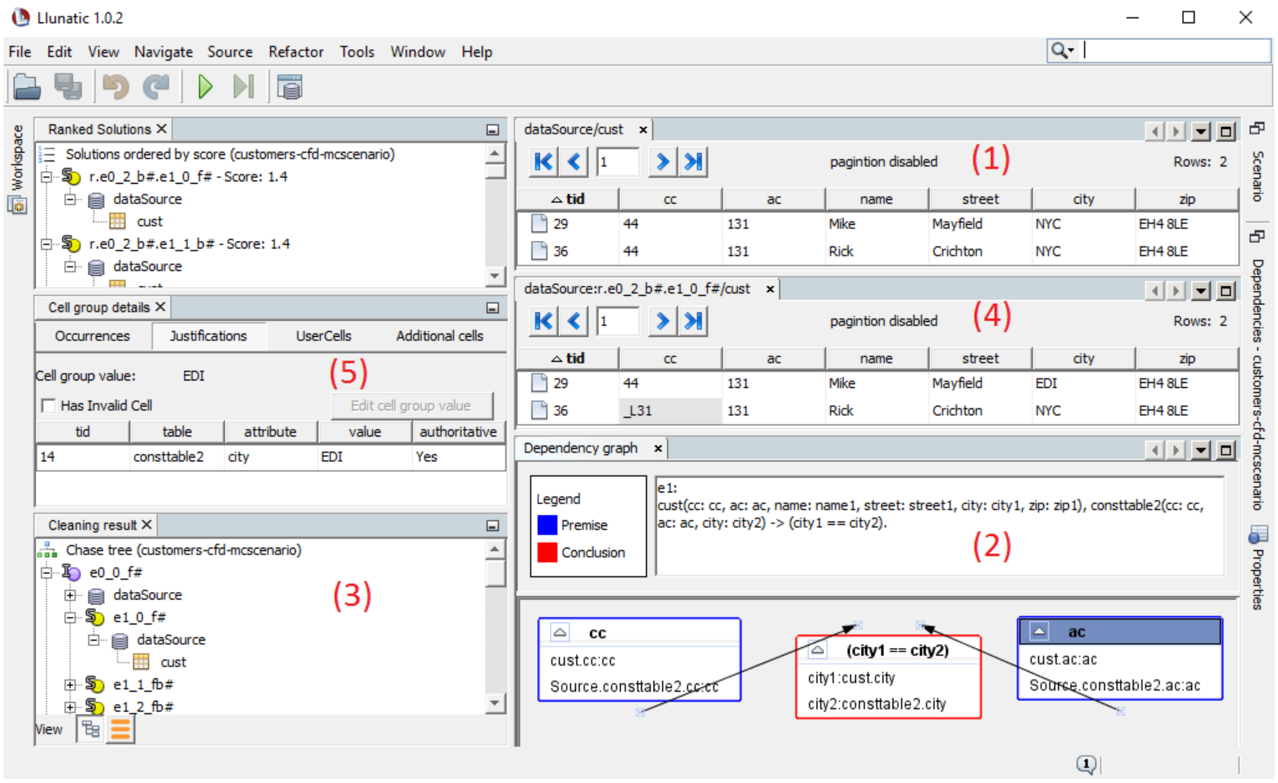
Given a dataset with its ordering attributes and a set of data quality rules, LLUNATIC compiles the information in the initial labeled instance and then start to compute a set of *solutions*, i.e., target instances that satisfy the constraints according to its semantics. To do this, it generates a chase tree, reported in *frame (3)*. Leaves in the chase tree are solutions that can be inspected by users to analyze the modifications to the original database, as shown in *frame (4)*, where value ‘44’ has been update with a llun.

As LLUNATIC models updates to the database in terms of *cell groups*, for each updated value is possible to retrieve its “repair instructions with lineage”, as illustrated in (*frame (5)*. More specifically, it is possible to see for each cell group: (*i*) which conflicting cells were modified, and which were their values; (*ii*) which value was chosen to repair the conflict; (*iii*) whether this value comes from one or more of the cells in the source databases, and if these cells are authoritative. **this must be tied better with the terminology we use now**

When no preference rule is available, LLUNATIC does not make arbitrary choices, and rather marks conflicts with lluns so that users may resolve them later on. A llun is introduced whenever there is no clear way to upgrade the dirty database by changing a cell to a new constant. By analyzing the cell value associated to a llun, the users can look at the tuple involved, focus on the violation at hand and manually define a value.

Users can also stop the chase to provide inputs. They may pick up a node in the chase tree, consult its history in terms of changes to the original database, inspect the lluns that have been introduced, and analyze the associated cell groups. Based on this, informed decisions are taken in order

[4] `https://github.com/donatellosantoro/Llunatic`


Llunatic 1.0.2
File Edit View Navigate Source Refactor Tools Window Help
Workspace
Ranked Solutions
Cell group details
Occurrences Justifications UserCells Additional cells
Edit cell group value
Cleaning result
View
dataSource/cust
dataSource:r.e0_2_b#.e1_0_f#/cust
Dependency graph
Scenario
Dependencies - customers-cfd-mcscenario
Properties
(1)
(2)
(3)
(4)
(5)

Solutions ordered by score (customers-cfd-mcscenario)
- r.e0_2_b#.e1_0_f# - Score: 1.4
  - dataSource
    - cust
- r.e0_2_b#.e1_1_b# - Score: 1.4
  - dataSource

Cell group value: EDI

☐ Has Invalid Cell

| tid | table | attribute | value | authoritative |
|---|---|---|---|---|
| 14 | consttable2 | city | EDI | Yes |

Chase tree (customers-cfd-mcscenario)
- e0_0_f#
  - dataSource
  - e1_0_f#
    - dataSource
      - cust
  - e1_1_fb#
  - e1_2_fb#

dataSource/cust — pagintion disabled — Rows: 2

| tid | cc | ac | name | street | city | zip |
|---|---|---|---|---|---|---|
| 29 | 44 | 131 | Mike | Mayfield | NYC | EH4 8LE |
| 36 | 44 | 131 | Rick | Crichton | NYC | EH4 8LE |

dataSource:r.e0_2_b#.e1_0_f#/cust — pagintion disabled — Rows: 2

| tid | cc | ac | name | street | city | zip |
|---|---|---|---|---|---|---|
| 29 | 44 | 131 | Mike | Mayfield | EDI | EH4 8LE |
| 36 | _L31 | 131 | Rick | Crichton | NYC | EH4 8LE |

Legend
🟦 Premise
🟥 Conclusion

e1:
cust(cc: cc, ac: ac, name: name1, street: street1, city: city1, zip: zip1), consttable2(cc: cc, ac: ac, city: city2) -> (city1 == city2).

cc: cust.cc:cc; Source.consttable2.cc:cc
(city1 == city2): city1:cust.city; city2:consttable2.city
ac: cust.ac:ac; Source.consttable2.ac:ac

Fig. 5: LLUNATIC GUI.

to remove lluns and replace them with the appropriate constants, or discard unwanted repairs. Lllluns and cell groups provide an effective source of information to support users in their choices.

## 6 Comparison to Other Semantics

We further illustrate how partial-order specifications and initial label instances can be used such that the LLUNATIC chase mimics repair semantics used in other work. In particular, we show how:

– frequency information can be used to resolve conflicts. This is motivated by the *Minimum Cost* repair method for functional and conditional functional dependencies introduced in [10, 16]. Here, when conflicts need to be resolved, equivalence classes are formed of cells that need to get the same value. The actual values for such classes are determined at the end and are selected based on a cost function [10]. We focus on the heuristic in which the most frequent value in each equivalence class is selected in an attempt to minimize the number of changes made.
– random conflict resolution can be incorporated. This is motivated by the *Sampling* repair method for functional dependencies [9]. Here, conflicts of functional dependencies are randomly resolved (forward or backward) and special variables or values are randomly selected to repair conflicts.

*Example 9* Consider the initial labeled instance $I^\circ$ shown in Figure 6. Compared to our running example we added one more tuple and expanded the schema with two ordering attributes: Freq, which is to hold the frequency of values appearing in the Name-attribute, and Rnd, which holds random values. The partial-order specification $\Pi$ maps Name to Freq, and Surname to Rnd. The attribute Spec is still mapped to Conf, just as before. This implies, e.g., that $p_{11} \prec_{\mathbf{P}} p_{21} =_{\mathbf{P}} p_{31}$, $p_{33} \prec_{\mathbf{P}} p_{23} \prec_{\mathbf{P}} p_{13} \prec_{\mathbf{P}} p_{43}$, and $p_{25} =_{\mathbf{P}} p_{35} \prec_{\mathbf{P}} p_{15} \prec_{\mathbf{P}} p_{45}$. Here, $x =_{\mathbf{P}} y$ denotes that $x \preceq_{\mathbf{P}} y$ and $y \preceq_{\mathbf{P}} x$ hold. In addition, we still have the standard partial order information related to nulls, invalid, authoritative and user preference levels as in Figure 4. Suppose that we chase $I^\circ$ in a forward way with the variable egds $e_1$–$e_4$ in $\Sigma$, stating that attribute NPI is a key of the relation. Then it should be clear that the LLUNATIC chase resolves conflicts of Name-values based on frequency. We also recall that the chase changes cells belonging to the same cell group in the same way. Intuitively, cell groups can be seen to correspond to the equivalence classes used in [10, 16]. Hence, the chase behaves like the minimum cost repairing method for the Name attribute. Similarly, the chase resolves conflicts of Surname-values in a random way. When complemented with the sampling cost manager (see Section 7), which basically chooses chase steps at random, we obtain a repairing method sim-

*D*(octors)

| | NPI | Name | Freq | Surname | Rnd | Spec | Conf | Hospital |
|---|---|---|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ $\{\langle p_{10}, 111\rangle\}$ | $c_{11}$ $\{\langle p_{11}, \text{Robert}\rangle\}$ | $c_{12}$ $\{\langle p_{12}, 1\rangle\}$ | $c_{13}$ $\{\langle p_{13}, \text{Chase}\rangle\}$ | $c_{14}$ $\{\langle p_{14}, 0.55\rangle\}$ | $c_{15}$ $\{\langle p_{15}, \text{surg}\rangle\}$ | $c_{16}$ $\{\langle p_{16}, 0.9\rangle\}$ | $c_{17}$ $\{\langle p_{15}, \text{PPTH}\rangle\}$ |
| $t_2$ | $c_{20}$ $\{\langle p_{20}, 111\rangle\}$ | $c_{21}$ $\{\langle p_{21}, \text{Frank}\rangle\}$ | $c_{22}$ $\{\langle p_{22}, 2\rangle\}$ | $c_{23}$ $\{\langle p_{23}, \text{Chasee}\rangle\}$ | $c_{24}$ $\{\langle p_{24}, 0.21\rangle\}$ | $c_{25}$ $\{\langle p_{25}, \text{urol}\rangle\}$ | $c_{26}$ $\{\langle p_{26}, 0.1\rangle\}$ | $c_{27}$ $\{\langle p_{\perp_1}, \perp_1\rangle\}$ |
| $t_3$ | $c_{30}$ $\{\langle p_{30}, 111\rangle\}$ | $c_{31}$ $\{\langle p_{31}, \text{Frank}\rangle\}$ | $c_{32}$ $\{\langle p_{32}, 2\rangle\}$ | $c_{33}$ $\{\langle p_{33}, \text{Chase}\rangle\}$ | $c_{34}$ $\{\langle p_{34}, 0.03\rangle\}$ | $c_{35}$ $\{\langle p_{35}, \text{urol}\rangle\}$ | $c_{36}$ $\{\langle p_{36}, 0.1\rangle\}$ | $c_{37}$ $\{\langle p_{\perp_1}, \perp_1\rangle\}$ |
| $t_4$ | $c_{40}$ $\{\langle p_{40}, 222\rangle\}$ | $c_{41}$ $\{\langle p_{\perp_2}, \perp_2\rangle\}$ | $c_{42}$ $\{\langle p_{42}, 1\rangle\}$ | $c_{43}$ $\{\langle p_{43}, \text{House}\rangle\}$ | $c_{44}$ $\{\langle p_{44}, 0.812\rangle\}$ | $c_{45}$ $\{\langle p_{45}, \text{diag}\rangle\}$ | $c_{46}$ $\{\langle p_{46}, 1\rangle\}$ | $c_{47}$ $\{\langle p_{\perp_1}, \perp_1\rangle\}$ |

(a) Initial labeled instance $I^\circ$

*D*(octors)

| | NPI | Name | Surname | Spec | Hospital |
|---|---|---|---|---|---|
| $t_1$ | $c_{10}$ 111 | $c_{11}$ Frank | $c_{13}$ Chase | $c_{15}$ surg | $c_{17}$ PPTH |
| $t_2$ | $c_{20}$ 111 | $c_{21}$ Frank | $c_{23}$ Chase | $c_{25}$ surg | $c_{27}$ PPTH |
| $t_3$ | $c_{30}$ 111 | $c_{31}$ Frank | $c_{33}$ Chase | $c_{35}$ surg | $c_{37}$ PPTH |
| $t_4$ | $c_{40}$ 222 | $c_{41}$ $\perp_2$ | $c_{43}$ House | $c_{45}$ diag | $c_{47}$ PPTH |

(b) Corresponding instance $\mathsf{inst}(I^\circ)$.

Fig. 6: Extended running example with extra ordered attributes Freq (frequency) and Rnd (random).

ilar to the Sampling method [9]. The instance $\mathsf{inst}(I^\circ)$ obtained by chasing $I^\circ$ with the variable egds is shown in Figure 6 (we omitted the ordering attributes). We also remark that although we explicitly added ordering attributes to the schema, one can of course regard these as virtual attributes and compute frequencies or random value on the fly, when needed. $\diamondsuit$

We want to stress that these are just two examples. By adding ordering attributes related to string similarity, distance functions, timestamps, and others, one can encode complex relationships between preference levels by using appropriate partial-order specifications. These in turn affect how conflicts are resolved during the chase and what kind of repairs one obtains.

## 7 Implementing the chase

The computation of the chase tree of all chase sequences of $\mathbf{I}^\circ$ with $\Sigma$ and User, i.e., $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^\circ)$, is the core algorithmic component in LLUNATIC. In this Section, we describe some underlying internal optimizations and an external mechanism, called the *cost manager*, to control the chase in a fine-grained manner.

To accommodate for large datasets, LLUNATIC is built around a *disk-based* chase engine. The chase logic in controlled by a Java program that handles the heuristic decisions we describe next, such as when to go forward or backward, computation of value similarity, and caching strategies. Disk support, essential for scalability, is provided by exploiting a DBMS for data access. This is a natural choice for our setting, as a DBMS is faster and exposes data operations closer to our needs than the OS file system.

### 7.1 Chasing on top of a DBMS

Due to space limitations we only provide a high-level description of some internal implementation choices.

**Storing the delta's.** It is clearly infeasible to materialize the entire chase tree since each of its nodes corresponds to an upgrade, i.e., a labeled instance obtained by a chase step, and we may have exponentially many repairs. In LLUNATIC, we therefore only store the changes (the "delta's") made after each chase step, i.e., how the preference labels in a labeled instance are changed in each step. We use a relational representation in which changes to the labeled instance made in one chase step are grouped together by means of the same value for a special StepId-attribute. We store strings in StepId which uniquely identify nodes in the chase tree and such that ancestor nodes are identified by prefixes of those strings. By means of SQL queries we can check easily for violations of the egds and user-input function, and for each set of violating tuples (i.e., for homomorphisms that make egds applicable), we add the changes as determined by the chase steps to the preference labels of the cells involved.

**Caching of cell groups.** In Section 5 we explained how the chase changes together all cells in the same cell group. Speeding up the identification and management of the cell groups involved at each step is crucial for performance. We therefore introduce three *caching strategies* for cell groups: (i) the *lazy* strategy, in which a cell group is first searched in the cache; in case it is missing, it is loaded from the database and stored in the cache; (ii) the *greedy* strategy in which the first time a cell group for a chase step $s$ is requested, we load into the cache all cell groups involved in step $s$ with a SQL query; and (iii) the *single-step* strategy, that caches cell groups for a single step at a time. Similarly to *greedy*, we keep cell groups for chase step $s$ in the cache, but, whenever a cell group for a different step $s'$ is requested, we clean the cache and load all cell groups for $s'$. We will show in our experiments that the last strategy performs best, as the first two tend to keep in memory cell groups that are not immediately reused.

**Equivalence class based chase.** We limit the number of nodes (upgrades) generated by *grouping together different homomorphisms $h$* that make an egd $e : \phi(\bar{x}) \to x_i = x_j$ (or $e : \varphi(\bar{x}) \to x = a$) applicable, as follows. Let $\mathbf{I}^\star_s$ be the labeled instance obtained in step $s$ of the chase. Let $h$ and $h'$ be two different homomorphisms of $\phi(\bar{x})$ into $\mathsf{inst}(\mathbf{I}^\star_s)$ such that $e$ is applicable to $\mathbf{I}^\star_s$ with $h$ and $h'$. We then say that $h$ and $h'$

are *compatible* if $h$ and $h'$ agree on all occurrences of variables $x$ that occur more than once in $\phi(x)$. Intuitively, this implies that the chase steps for $h$ and $h'$ can be combined. The compatibility relation induces an equivalence classes of homomorphisms and we perform a *single* chase step for *each equivalence class* of homomorphisms.

*Example 10* Consider the schema $R(\mathsf{Tid}, A, B, C)$ and the following labeled instance $\mathbf{I}^{\star}$ over $(\mathbf{P}, \preceq_{\mathbf{P}})$:

| Tid | $A$ | $B$ | $C$ |
|---|---|---|---|
| $\{\langle p_{id}, 1\rangle\}$ | $\{\langle p_{11}, 1\rangle\}$ | $\{\langle p_{0.1}, 1\rangle\}$ | $\{\langle p_{13}, 1\rangle\}$ |
| $\{\langle p_{id}, 2\rangle\}$ | $\{\langle p_{21}, 1\rangle\}$ | $\{\langle p_{0.2}, 2\rangle\}$ | $\{\langle p_{23}, 2\rangle\}$ |
| $\{\langle p_{id}, 3\rangle\}$ | $\{\langle p_{31}, 1\rangle\}$ | $\{\langle p_{0.3}, 3\rangle\}$ | $\{\langle p_{33}, 3\rangle\}$ |

such that $p_{0.1} \prec_{\mathbf{P}} p_{0.2} \prec_{\mathbf{P}} p_{0.3}$ and all other preference levels are incomparable. Consider the variable egd $e: R(tid, x, y, z) \wedge R(tid', x, y', z') \rightarrow y = y'$ (expressing the functional dependency $A \rightarrow B$) and homomorphisms $h_1$, $h_2$ and $h_3$ such that $h_1(R(tid, x, y, z)) = t_1$, $h_1(R(tid', x, y', z')) = t_2$, $h_2(R(tid, x, y, z)) = t_1$, $h_2(R(tid', x, y', z')) = t_3$, $h_3(R(tid, x, y, z)) = t_2$ and $h_3(R(tid', x, y', z')) = t_3$. In $e$, only variable $x$ has multiple occurrences and all three homomorphisms map these occurrences to the same value "1". They are then regarded as compatible and the equivalence-based chase will apply the three corresponding forward chase steps simultaneously. The result is the labeled instance in which the cells corresponding to attribute $B$ in all tuples are assigned preference labels $\{\langle p_{0.1}, 1\rangle, \langle p_{0.2}, 2\rangle, \langle p_{0.3}, 3\rangle\}$; the preference labels of all other cells remain the same. $\diamondsuit$

We note that a similar equivalence-class based repairing strategy is used in [10,21]. One can verify that the equivalence-based chase still returns repairs. Of course, some repairs may be missed out because of the coarser granularity with which is chased. Nevertheless, the equivalence-based chase enables some additional ways of guiding the chase when combined with the cost manager, which we describe next.

## 7.2 Cost manager

We have shown before that all leaves in $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^{\circ})$ are repairs. Instead of computing all repairs, in practice, one wants to impose further conditions on these repairs, by e.g., limiting the number of repairs, disallowing backward chase steps, or by disallowing changes to very reliable attributes. In LLUNATIC we allow the user to *control* the behaviour of the chase by incorporating *pruning strategies*. To this aim, we complement the chase procedure in LLUNATIC with a *cost manager*. During the chase, only the nodes (i.e., intermediate results –upgrades– of chase steps) that are accepted by the cost manager are generated.

**Definition 14** A *cost manager* for a labeled instance $\mathbf{I}^{\circ}$ over $\mathbf{P}$ of $\mathbf{I}$, $\Sigma$ and User is a predicate CM over nodes in the chase tree $\mathsf{chase}_{\Sigma,\mathsf{User}}(\mathbf{I}^{\circ})$. For each node $n$ in this tree, it may either accept ($\text{CM}(n) = \text{true}$) or refuse it ($\text{CM}(n) = \text{false}$). $\square$

The standard cost manager is the one that accepts all chase nodes. We note that when the cost manager is too restrictive, it is possible that no repairs are found. Indeed, simply consider the cost manager that rejects all nodes. More practical cost manager are motivated by approaches taken in related work, as follows:

- the *maximum size* cost manager (SN): it accepts new nodes 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 one child of each node, and rejects the rest; as a specific case, the S1 cost manager only generates one path in the chase tree, and ignores other branches;
- the *forward-only* cost manager (FO): it accepts forward nodes (i.e., nodes representing the result of a forward chase step) and rejects backward nodes (i.e., nodes representing the result of a backward chase step);
- the *sampling* cost manager (SPLK): it randomly accepts nodes, until $K$ repairs have been generated (see also [9]);
- The *certain-region* cost manager (CTN): it incorporates the notion of a *certain region* [24], i.e., a set of attributes that are considered "fixed". Values in these attributes are considered reliable, and cannot be changed. Nodes corresponding to results of chase steps in which changes to these reliable attributes are made are rejected; all others are accepted.
- The *frequency* cost manager (FR) (or *similarity to most frequent*): it is inspired by the heuristics originally proposed in [10] and modelled in our semantics as discussed in Section 6. We resort to this manager when preference levels for the cells in a violation for dependency $e$ are incomparable with other partial orders. The idea is to make local decisions for which cells to change by analyzing the violations in one equivalence class for $e$. For a given equivalence class of homomorphisms (same value for the premise cells involved in $e$), the cost manager computes the frequency of values appearing in conclusion cells and a similarity measure across their values (based on the Levenshtein distance for strings). Based on this information, it makes decisions in terms of the next chase step. The manager rejects repair strategies that backward-chase cells with the most frequent conclusion value. The intuition is that these cells are likely to be correct. 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 (i.e., it is likely to be a typo); otherwise, it is backward chased.

Notice that combinations of cost managers are possible, e.g., one can have a FO-S5 or a SPL50-FO cost manager. The FO-S5 strategy, for example, discards backward changes and, in addition, it considers five different ways of chasing in a forward way. We believe that cost managers are an elegant way of controlling the chase.

## 8 Experiments

This section reports our experimental results with LLUNATIC. We consider several cleaning scenarios of different nature and sizes, and study both the quality of the upgrades computed by our system, and the scalability of the chase algorithm. We show that our algorithm produces upgrades of better quality with respect to other systems in the literature, and at the same time scales to large databases. We ran all tests on a server with 40 physical Xeon v4 cores running at 2.4GHz and a 512 GB SSD under Ubuntu v16. All the tools are Java-based, use PostgreSQL as DBMS, and have been executed on a JVM with 16 GB of RAM.

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

**Datasets and Scenarios.** We selected five datasets:
$(a)$ Hospital is based on real data from the US Department of Health & Human Services[5]. It contains a single table with 100K tuples and 17 attributes, over which we specified 7 functional dependencies. In order to test the scalability of the systems, we generated instances of size up to 1M tuples by replicating the original data several times. We called this variant Hospital-Synth.
$(b)$ Bus, is a real-world scenario used by Dallachiesa et al. [17] composed by a single table containing 284K tuples with 25 attributes, and 9 functional dependencies.
$(c)$ IMDB, it contains real data about movies, directors and actors obtained by joining data provided by Internet Movie Database (IMDB)[6]. The resulting single table is composed by 8 attributes and contains 20 million of tuples. We identified 4 functional dependencies for this dataset.
$(d)$ Tax, is a synthetic scenario from Fan et. al. [22] with a single table with 15 attributes and 4 functional dependencies.
$(e)$ Doctors, corresponds to our running example introduced in Section 3. The target database schemas contain 2 tables, plus 1 master data table. We considered 3 editing rules, 4 conditional functional dependencies and 3 functional dependencies. We synthetically generated up to 1M tuples with a proportion of 40% in the Doctors table, and 60% in a Treatments table; the master-data table contains a few hundreds of the tuples present in Doctors. We consider master-data tuples outside the total, as they cannot be modified.

These scenarios represent a wide spectrum of data-repairing problems. The first four scenarios contain functional dependencies only, and therefore are quite standard in terms of constraints. Hospital 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 dependencies interact in various ways, and there is no partial-order information that can be used to ameliorate the cleaning process. The Doctors scenario contains a complex mix of dependencies; this increased complexity of the constraints is compensated by the fact that data are stored as normalized tables, with no redundancy, and preference strategies are given for some of the attributes.

**Errors and Metrics** In order to test our algorithms with different levels of noise, we introduced errors by using BART, an open-source error-generation tool [3]. Differently from ad-hoc strategies, BART allows researchers to inject errors into data in a principled and controlled way. More specifically:

– it guarantees that all errors are *detectable* using the given constraints, i.e., it does not generate errors that are "impossible" to identify using a constraint-based tool;
– it can control the degree of *repairability* of errors; intuitively, this is a measure of how "difficult" to repair errors are;
– finally, it represents a platform for researchers to share their datasets and error-generation configurations, in order to foster repeatability.

We introduced 5% of errors, all detectable by the constraints. In order to test the impact of the errors on the final quality of the process, we used different level of repairability, that we call HIGH REP, MED REP and LOW REP. Datasets, constraints, and BART configurations are available through the project web site[7].

For all scenarios, we measure running times and size of the chase trees. We measure quality as precision and recall in terms of dirty cells that have been restored to the original values. More specifically, for each clean database, we generated the set $\mathscr{C}_p$ of perturbated cells. Then, we run each algorithm to generate a set of repaired cells, $\mathscr{C}_r$, and computed precision ($P$), recall ($R$), and F-measure ($F = 2 \times (P \times R)/(P+R)$) of $\mathscr{C}_r$ wrt $\mathscr{C}_p$. Since several of the algorithms may introduce variables to repair the database – like our lluns – we calculated two different metrics.
- *Metric 0.5.* This is the metrics adopted in [9]: $(i)$ for each cell $c \in \mathscr{C}_r$ repaired to the original value in $\mathscr{C}_p$, the score was 1; $(ii)$ for each cell $c \in \mathscr{C}_r$ changed into a value different

[5] http://www.medicare.gov/hospitalcompare/
[6] https://datasets.imdbws.com/
[7] http://db.unibas.it/projects/llunatic/

from the one in $\mathscr{C}_p$, the score was 0; $(iii)$ for each cell $c \in \mathscr{C}_r$ repaired to a variable value, if the cell was also in $\mathscr{C}_p$, the score was 0.5. In essence, a llun or a variable is counted as a partially correct change. This gives an estimate of precision and recall when variables are considered as a partial match.
- *Metric 1.0.* Since our scenarios may require a consistent number of variables, due to the need for backward updates, and this metric disfavors variables, we also adopt a different metric, which counts all correctly identified cells to repair. In this metric, called *Metric 1.0*, item $(iii)$ above becomes: for each cell $c \in \mathscr{C}_r$ repaired to a variable value, if the cell was also in $\mathscr{C}_p$, the score was 1.

**Algorithms** We ran LLUNATIC with several cost managers and several caching strategies, as discussed in Section 7. 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 single-node approaches, we tested the several repairing algorithms from the literature, implemented as separate systems: $(a)$ HOLISTIC [14]; $(b)$ MIN. COST [10]; $(c)$ VERTEX COVER [34]; $(d)$ SAMPLING [9]; for this, we took 500 samples for each experiment, as done in the original paper. All of these systems support a smaller class of constraints wrt to the ones expressible in our framework, and cannot handle all of the constraints in the Doctors experiment. Therefore, only variants of LLUNATIC were used for the latter.

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

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

**The Quality Experiment** We want first to investigate the quality of the several repairing algorithms, using three different datasets: Hospital, Bus and Tax. For each of them, as discussed, we made three dirty versions introducing 5% errors with different repairability levels. In Table 1 we report the average repairability of errors. An higher repairability configuration involves mostly rules with master data and CFDs (if any) and contains errors for right-hand sides of FDs, while a low repairability one mostly involves left-hand side errors.

Notice that we do not report quality results for Doctors and IMDB since LLUNATIC is the only system capable of handling these scenarios, either due to the variety of dependencies, or to the size of data. Results obtained by LLUNATIC in these scenarios are in line with those discussed below.

We begin with comparing the quality obtained by the different LLUNATIC cost managers (Figure 7.a,e). For this task we choose the Hospital scenario since it contains highly interacting dependencies. As expected the LLUNATIC-FR-S10 cost manager shows better result wrt LLUNATIC-FR-S1, especially for the LOW REP variant. The LLUNATIC-FR-S1-FO cost manager shows good results only whenever the repairing task is easy, while in harder cases the choice of repairing always in a forward way is not appropriate.

| | **Hospital 20k** | **Bus 20k** | **Tax 20k** |
|---|---|---|---|
| **High Rep** | 0.89 | 0.85 | 0.89 |
| **Med Rep** | 0.59 | 0.51 | 0.74 |
| **Low Rep** | 0.12 | 0.33 | 0.49 |

Table 1: Repairability levels for the dirty databases used in Figure 7.a-h

In Figures 7.b-d and 7.f-h we compare LLUNATIC-FR-S1 to the other systems. We notice that LLUNATIC produces repairs of significantly higher quality with respect to those produced by previous algorithms. Quality results for algorithms MIN. COST, SAMPLING, and VERTEX COVER are consistent with those reported in [9], which also conducted a comparison of these three algorithms on scenarios in which forward and backward repairs were necessary.

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

The superior quality achieved by LLUNATIC variants can be explained by first noticing that algorithms capable of repairing both forward and backward obtained better results than those that only perform forward repairs. Besides LLUNATIC, the other algorithms capable of backward repairs are HOLISTIC and SAMPLING. In particular the LLUNATIC's chase algorithm explores the space of solutions in a more systematic way, and this explains its improvements in quality, espe-

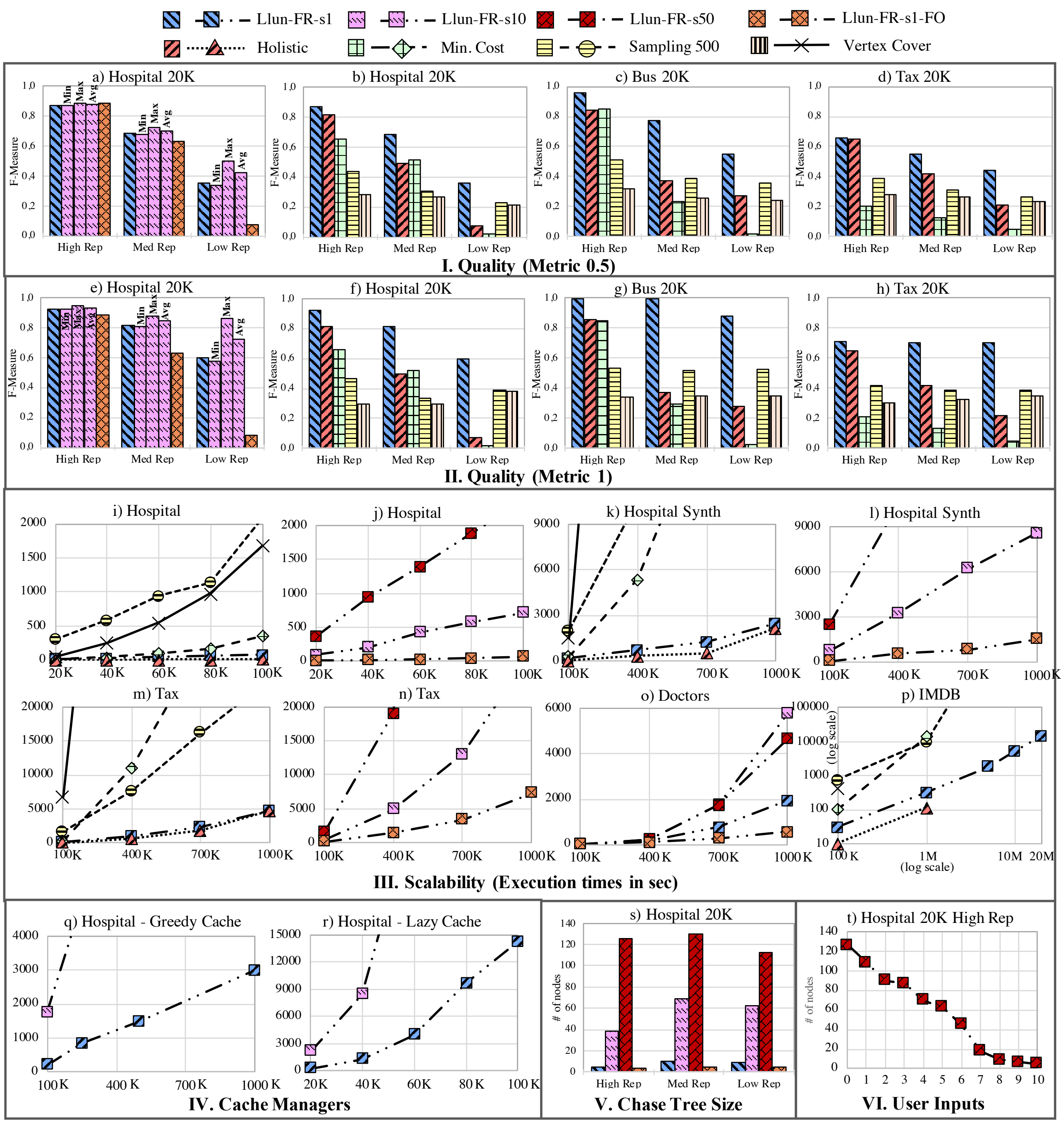


Fig. 7: Experimental Results

cially in harder scenario. In light of this, the superior quality achieved by the LLUNATIC variants, which clearly outperformed the competitors, is a significant improvement.

**The Scalability Experiment** The second set of experiments is aimed at testing scalability. In Figures 7.$i-p$ we compare execution times (in seconds) for the various algorithms on databases with different sizes. We started with a small scenario, Hospital, with data that vary from 20k to 100k tuples, to end up with a very large scenario, IMDB, with 20 millions of tuples (note the logarithmic scale).

To begin, recall that LLUNATIC is the first disk-based implementation of a data repairing algorithm. Therefore, our implementation is somehow disfavored in the comparison or execution times wrt to main-memory tools. More specifically, 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.

On the contrary, our algorithm uses SQL for accessing and repairing data: updating a single cell (a very quick operation, when it is performed in main memory) using the DBMS requires to perform an UPDATE, and therefore a SELECT to locate the right tuple. This difference drastically affects the execution time of a repair. Nevertheless, the LLUNATIC-FR-S1 cost manager scales nicely and has better performance than some of the main-memory implementations, and in general has execution times close to the faster main-memory system, HOLISTIC.

In Figure 7.*j* the other three cost managers of LLUNATIC are compared to each other. We noticed that the LLUNATIC-FR-S1-SO variant has almost the same performance as the LLUNATIC-FR-S1 variant (Figure 7.*i*) but, as discussed, it gives in general worse quality results. In addition the system scales almost linearly with respect to the different number of permutations tested.

Comparing both quality and scalability results we may say that LLUNATIC-FR-S1 represents the best trade-off in terms of quality and scalability for all the considered scenarios. The same trend is also confirmed in the Hospital-Synth and Tax experiments. Other algorithms do not allow to fine tune this trade-off. To see an example, consider the SAMPLING algorithm: we noticed that taking 1000 samples instead of 500 doubles execution times, but it does not produce significant improvements in quality.

Even in scenarios with more complex dependencies like Doctors, our system gives excellent results (Figure 7.*o*). Other systems are not reported here since they were not able to handle the kind of constraints used in this scenario.

Finally, in Figure 7.*p* we show the clear benefits that come with a DBMS implementation wrt main-memory ones, namely the possibility of scaling up to large databases. While previous works have reported results up to a few thousand tuples, we were able to investigate the performance of the system on databases of up to 20 millions of tuples. In these cases, execution times in the order of an hour can be considered as a remarkable result, since no single node system had been able to achieve them before on problems of such exponential complexity. None of the main-memory system was able to execute scenario with more than 1M of tuples. Notice that these results about LLUNATIC were confirmed in a recent study about the scalability of chase engines [5].

**Comparison with ML cleaning** We report the results for HOLOCLEAN [37], a data cleaning system that takes as input declarative rules together with other probabilistic signals, such as cell co-occurence, provenance information, and external dictionaries lookup. To combine such information, it adopts a probabilistic semantics to estimate the value of every noisy cell in the dataset, together with a probability for the given value of being correct. HOLOCLEAN does not compute repairs according to our definition: input constraints are not satisfied by the produced instances. However, it does improve the quality of the instances and it is worth comparing its output to LLUNATIC's results.

| | Hospital 20k | Bus 20k | Tax 20k |
|---|---|---|---|
| **High Rep** | 0.95 | 0.86 | 0.76 |
| **Med Rep** | 0.83 | 0.62 | 0.71 |
| **Low Rep** | 0.79 | 0.38 | 0.64 |

Table 2: HOLOCLEAN F-Measure results with Metric-1

In terms of Metric-1 results, Table 2 shows that HOLOCLEAN has results comparable to LLUNATIC in most cases. Notable exceptions are Hospital in the LOW REP configuration, where HOLOCLEAN does better, and Bus scenarios, where LLUNATIC has better results. To obtain the results of HOLOCLEAN, we manually tuned its configuration parameters. More specifically, we used the lowest not-failing value for "PruningTopK" on our 16GB machine. Notice also that our noisy instances are the ideal input for HOLOCLEAN as all errors are detectable. In terms of execution times, HOLOCLEAN is a main-memory algorithm and could not scale to large input instances with our scenarios and machine configuration: Hospital failed with 60k and Tax with 100k. For the 20k instances reported above, execution times varied between 480 (Hospital–LOW REP) and 1055 seconds (Bus–HIGH REP).

**The Cache Manager Experiment** In this experiment, we investigate the impact of our optimizations on the scalability of the chase (Section 7). While in all previous experiments we used the *single step* caching strategy, we report scalability in Figures 7.*q*–*r* the quality results of the Hospital dataset for the other two caching strategies, the greedy and the lazy cache manager. The charts show that the single-step cache represents the best choice in terms of performance. This is explained by the high degree of locality in our chase algorithm. When chasing node $s$ in the tree to generate its children, only cell groups for step $s$ are needed. Then, after we move from $s$ to its first child, $s'$, cell groups of $s$ will not be needed for a while.

**The Chase Tree Size and User Input Experiments** Execution times achieved by the algorithm can be considered as a remarkable result for problems of this complexity. They are even more surprising if we consider the size of the chase trees that our algorithm computes, which may reach several hundreds of nodes as reported in Figure 7.*s*. Consider also that each node in the tree is a copy of the entire database. It is also worth noting that storing chase trees as delta databases is crucial in order to achieve such a level of scalability. Without such a representation system times would be orders of magnitude higher.

We finish by mentioning Figure 7.*t*, in which we study the impact of user inputs on the chase process. We run the experiment for 20K tuples interactively, and provided random user inputs by alternating the change of a llun value with the rejection of a leaf. It can be seen that small quantities of inputs from the user may significantly prune the size of the chase tree, and therefore speed-up the computation of solutions.

## 9 Related Work

There has been a host of work on data quality management (see [31,38] for recent surveys). It has been shown experimentally with real annotated datasets that methods inspired by different ideas must be all used in practice to achieve high data quality [1]. Among methods based on statistical analysis, such as outlier detection [13], and methods that rely on look up of external dictionaries, such as knowledge bases [15], it stands out that rule (constraint) based methods are a necessary ingredient.

Several classes of constraints have been proposed to characterize and improve the quality of data. Most relevant to our work are the (semi-)automated repairing algorithms for these constraints [9,10,16,24,25,34]. These methods differ in the constraints that they admit, e.g., FDs [9,10], CFDs [16, 34], inclusion dependencies [10], and editing rules [24], and the underlying techniques used to improve their effectiveness and efficiency, e.g., statistical inference [16], measures of the reliability of the data [10,24], and user interaction [16, 45,28].

All of these methods work for a specific class of constraints only, with the exception of [25,29]. A flexible data quality system was recently proposed [17] which allows user-defined procedural code for detection and cleaning. These works explore the interaction among different kinds of dependencies, but they do not have a unified formal semantics and clear definition of a solution, neither the generality of our partial order to model preferences.

Even more importantly, our system is the first disk-based scalable and efficient repair based method. While some of the algorithms above has been rewritten to be executed in a multi-node distributed enviroment, they are still bounded by the size of the memory size. For example, the holistic cleaning algorithm [14] has been adapted to be executed on top of Spark [33] in order to benefit from the bigger memory in the cluster. Interestingly, our system can handle their cleaning scenarios in a single node setting. Table 3 summarizes the features of LLUNATIC wrt some earlier approaches to data repairing. We leave out of the table related data cleaning systems that do not compute repairs [44,37,15].

Some of the ingredients of our scenarios are inspired by from, features of other repairing approaches: repairing based on both premise and conclusion of constraints [16,9, 34], cells [9,34,10], groups of cells [10], partial orders [23] and its incorporation in the chase [8]. We discuss these aspects in detail next.

We do allow for forward and backward chasing. Similarly, [16,34,9] 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 [34,9] 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 fact, lluns and cell-groups can be seen as a novel representation system [32] for solutions, that stands in between the naive tables of data exchange and the more expressive c-tables, trying to strike a balance between complexity and expressibility.

An approach similar to ours has been proposed in [8], 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. Furthermore, update-based database repairing has been considered in [43]. In that work, a pre-order on tableaux is defined and so-called up-repairs are introduced as a way of to compute consistent query answers in the presence of updates.

In this work we discussed how our system can compute repairs with a smaller number of chase step by exploiting user interaction, a popular way to involve the domain experts in improving data repairing [16,45,28].

## 10 Other Applications of the Partial Order

The framework developed in this paper has already been used as a basis for a number of extensions in data repairing. The semantics has been extended to proposes an end-to-end solution to deal with schema-mappings problems in presence of inconsistencies [27]. It has also been used as a baseline for developing an interactive approach to data repairing [28]. In this respect, we believe that this work may provide the basis for further investigation on the subject of both data transformation and data repairing.

However, a crucial contribution of this article consists in developing a new semantics to incorporate preference strategies in data repairing. We showed that our partial order allows users to model master-data, value confidence, and even currency rules. In this section, we discuss how our partial order is related to other different notions of preference that

| | Dependency Language | | | | Repair Strategy | | Value Preference | | | Solution Selection | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| System | FDs | CFDs | ERs | DCs | RHS | LHS | Confid. | Currency | Master | Cost | Certain | Card. Min | Sampling |
| [10] | √ | | | | √ | | √ | √ | | √ | | | |
| [16] | √ | √ | | | √ | √ | √ | √ | | √ | | | |
| [34] | √ | √ | | | √ | √ | | | | √ | | | |
| [24] | | | √ | | √ | | | | √ | | √ | | |
| [9] | √ | | | | √ | √ | | | | √ | | √ | √ |
| [17] | √ | √ | √ | | √ | √ | √ | | √ | | | √ | |
| [14] | √ | √ | √ | √ | √ | √ | √ | | √ | √ | | √ | |
| **This article** | √ | √ | √ | √ (eq. only) | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| | equality-generating dependencies | | | | chase proced. | | partial order | | | cost manager | | | |

Table 3: Feature Comparison for Data Repairing Systems.

have been studied in connection with data quality constraints. We believe those are valid directions of research enabled by our work.

### 10.1 Consistent Query Answering

We start by considering *prioritized repairs for consistent query answering* [40]. This research is inspired by works on preferred models for logic programs, and is different in spirit from our semantics. While we focus primarily on materializing preferred solutions by means of a general chase procedure, their focus is on consistent query answering in an inconsistent database. In this setting, a consistent answer is the one obtained in every repair. A prioritized repair is one that makes use of the use provided preferences about how to solve conflicts (e.g., "removing the tuples with the smaller salary"), similarly to what we allow with our partial order.

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

Nevertheless, our partial order has points of contact with their notion of a prioritized repair. Prioritized repairs rely on preference orders that are specified over tuples, and lift them to sets of tuples. On the contrary, we specify preference orders over cells, and lift them to cell groups, i.e., sets of cell modifications. This finer granularity of our approach makes our notion of an upgrade more general than their notion of *global optimal repair*. To see this, consider the simple example in which we have a single table $R(A,B,C)$ with a functional dependency $A \rightarrow B,C$, and a dirty instance $J = \{t_1 : R(a,1,4), t_2 = R(a,2,3)\}$. Suppose our partial order specification states that, for any attribute, cells with higher values should be preferred to the ones with lower values; this gives us the following minimal solution: $J' = \{R(a,2,4)\}$. First, we notice that $J'$ is not a repair in their setting, since any of their repairs must either correspond to $t_1$ or $t_2$ only (depending on the preference relation on tuples, and therefore on the tuple that is deleted to satisfy the FD). Second, by changing our partial order specification, we could simulate their semantics. Suppose we say that for attribute $B$ we prefer cells with lower values, while for attribute $C$ the ones with higher values; then we have a minimal solution: $J'' = \{t_1 = R(a,1,4)\}$ that coincides with their globally optimal repair. There are also other restrictions associated with prioritized repairing that we do not need to impose with our partial order, namely the acyclicity restriction on preference relations, and the notion of Pareto optimality.

### 10.2 Merge in Entity Resolution

Le us consider the *merge problem* within the reach of *entity resolution* [6] rather than constraint-based data repairing. It is formulated as follows: we are given a set of records $I_e$ with the same schema, that correspond to a description of a single real world entity $e$. These records may have conflicting values, and the goal is to derive a single entity tuple $t_e$, with the *most accurate values* for all attributes [11]. Master-data tuples may be used during the process. In the entity resolution field, this task is also called the *golden record* problem [18].

We focus our discussion on a recent paper that tries to come up with the golden record assuming the presence of a partial order [11]. While their algorithms do not aim at repairing an arbitrary database instance that is dirty wrt a set of constraints, there are some points in common with our approach. The authors develop a language of *accuracy rules* that have two goals. First, they can be used to specify a partial order among values. They can express that the value in cell $t'.A_1$ is more accurate than the one in cell $t.A_1$; this may happen, for example, because they know that more recent values are higher. Similarly to our ordering attributes, accuracy rules can be used to infer accuracy relationships among attributes, such as the value for attribute $A_2$ is more accurate in those tuples that have a more accurate attribute $A_1$. Second, the rules can be used to correct the entity tuple $t_e$ based on master data tuples, similarly to editing rules.

The authors develop algorithms to dynamically handle the construction of the entity tuple while at the same time deriving the partial order of accuracy among attribute values. The main concern here is about the termination and confluence of the process, i.e., whether the algorithm terminates, and whether it returns the same identical tuple regardless of the order in which accuracy rules are fired. This cannot be guaranteed in all cases. While this is not a general-purpose data repairing algorithm, since it does not contemplate constraints and makes the strong assumption that all tuples represent a single entity, we discuss how our partial order can be used. Our approach to the partial order is immune from termination and confluence problems. In fact, the partial-order specification of a cleaning scenario fixes a partial order of the cells of the initial instance, $J$; this partial order never changes during the chase. In other terms, our algorithm clearly separates the definition of the partial order for cells, that is done once and for all over $J$ before the repair process starts, and the generation of the actual updates using cell groups. This separation, along with the monotonicity property of cell groups, guarantees that our chase procedure for cleaning scenarios always terminates and gives deterministic results.

## 11 Conclusions

This paper develops a framework to handle cleaning tasks with complex dependencies and preference rules within a new semantics. The main tools upon which the framework is based are: $(a)$ a uniform, logic-based language to express dependencies; $(b)$ the adoption of a flexible data structure, called cell groups, to specify updates to the dirty database; $(c)$ a clear notion of an upgrade to a database, based on a partial order of cell groups and labels.

Our framework promotes a new approach to data repairing, based on the idea that a database should be updated in presence of conflicts only as long as these updates represent "certified improvements" to its quality. Ultimately, it provides a basis to involve users within the process, a much needed feature in data quality applications. In addition to the semantics, we have developed a number of techniques to implement the generation of solutions to cleaning scenarios within a disk-based scalable chase engine. This is the only tool able to scale data repairing over millions of tuples in a single node environment.

At the core of our solution there is the chase algorithm and, while there are several chase engines available, there are no distributed implementations. An important direction for new research is the development of a distributed version of our proposal. Another direction for future work is the deployment of our partial order in other applications, such as consistent query answering and entity resolution. Finally, an iterative repair process leads to a better understanding of the data for the user. It is natural to expect the original user-defined dependencies to change over time while examining the errors. Existing proposals studied how to decide between repairing data or constraints when users do data changes [12,30,41], but our system has specific opportunities that can be exploited. Specifically, new dependencies can trigger the reuse of previous user interactions, such as resolution of Llun values, and the backtracking of previous (heuristic) decisions taken in the chase tree. New algorithms are needed to optimize this iterative loop.

## References

1. Abedjan, Z., Chu, X., Deng, D., Fernandez, R.C., Ilyas, I.F., Ouzzani, M., Papotti, P., Stonebraker, M., Tang, N.: Detecting data errors: Where are we and what needs to be done? PVLDB **9**(12), 993–1004 (2016)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Arocena, P.C., Glavic, B., Mecca, G., Miller, R.J., Papotti, P., Santoro, D.: Messing up with BART: error generation for evaluating data-cleaning algorithms. PVLDB **9**(2), 36–47 (2015)
4. Beeri, C., Vardi, M.: A Proof Procedure for Data Dependencies. J. of the ACM **31**(4), 718–741 (1984)
5. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: PODS, pp. 37–52 (2017)
6. Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., Whang, S.E., Widom, J.: Swoosh: a Generic Approach to Entity Resolution. VLDB J. **18**(1), 255–276 (2009)
7. Bertossi, L.: Database Repairing and Consistent Query Answering. Morgan & Claypool (2011)
8. Bertossi, L., Kolahi, S., Lakshmanan, L.: Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. In: ICDT, pp. 268–279 (2011)
9. Beskales, G., Ilyas, I.F., Golab, L.: Sampling the repairs of functional dependency violations under hard constraints. PVLDB **3**, 197–207 (2010)
10. Bohannon, P., Flaster, M., Fan, W., Rastogi, R.: A cost-based model and effective heuristic for repairing constraints by value modification. In: SIGMOD, pp. 143–154 (2005)
11. Cao, Y., Fan, W., Yu, W.: Determining the relative accuracy of attributes. In: SIGMOD, pp. 565–576 (2013)
12. Chiang, F., Miller, R.J.: A unified model for data and constraint repair. In: ICDE (2011)
13. Chu, X., Ilyas, I.F., Krishnan, S., Wang, J.: Data cleaning: Overview and emerging challenges. In: SIGMOD, pp. 2201–2206 (2016)
14. Chu, X., Ilyas, I.F., Papotti, P.: Holistic Data Cleaning: Putting Violations into Context. In: ICDE, pp. 458–469 (2013)
15. Chu, X., Morcos, J., Ilyas, I.F., Ouzzani, M., Papotti, P., Tang, N., Ye, Y.: KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In: SIGMOD, pp. 1247–1261 (2015)
16. Cong, G., Fan, W., Geerts, F., Jia, X., Ma, S.: Improving data quality: Consistency and accuracy. In: VLDB, pp. 315–326 (2007)
17. Dallachiesa, M., Ebaid, A., Eldawy, A., Elmagarmid, A.K., Ilyas, I., Ouzzani, M., Tang, N.: Nadeef: a commodity data cleaning system. In: SIGMOD, pp. 541–552 (2013)
18. Deng, D., Tao, W., Abedjan, Z., Elmagarmid, A.K., Ilyas, I.F., Madden, S., Ouzzani, M., Stonebraker, M., Tang, N.: Entity consolidation: The golden record problem. CoRR **abs/1709.10436** (2017)

19. Fagin, R., Kolaitis, P., Miller, R., Popa, L.: Data Exchange: Semantics and Query Answering. TCS **336**(1), 89–124 (2005)
20. Fan, W., Gao, H., Jia, X., Li, J., Ma, S.: Dynamic constraints for record matching. VLDB J. **20**(4), 495–520 (2011)
21. Fan, W., Geerts, F.: Foundations of Data Quality Management. Morgan & Claypool (2012)
22. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional Functional Dependencies for Capturing Data Inconsistencies. ACM TODS **33** (2008)
23. Fan, W., Geerts, F., Wijsen, J.: Determining the Currency of Data. In: PODS, pp. 71–82 (2011)
24. Fan, W., Li, J., Ma, S., Tang, N., Yu, W.: Towards certain fixes with editing rules and master data. PVLDB **3**(1), 173–184 (2010)
25. Fan, W., Li, J., Ma, S., Tang, N., Yu, W.: Interaction Between Record Matching and Data Repairing. In: SIGMOD, pp. 469–480 (2011)
26. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: The LLUNATIC Data-Cleaning Framework. PVLDB **6**(9), 625–636 (2013)
27. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: Mapping and Cleaning. In: ICDE, pp. 232–243 (2014)
28. He, J., Veltri, E., Santoro, D., Li, G., Mecca, G., Papotti, P., Tang, N.: Interactive and deterministic data cleaning. In: SIGMOD, pp. 893–907 (2016)
29. Hernández, M., Koutrika, G., Krishnamurthy, R., Popa, L., Wisnesky, R.: Hil: a high-level scripting language for entity integration. In: EDBT, pp. 549–560 (2013)
30. Ilyas, I.F.: Effective data cleaning with continuous evaluation. IEEE Data Eng. Bull. **39**(2), 38–46 (2016)
31. Ilyas, I.F., Chu, X.: Trends in cleaning relational data: Consistency and deduplication. Foundations and Trends in Databases **5**(4), 281–393 (2015)
32. Imieliński, T., Lipski, W.: Incomplete Information in Relational Databases. J. of the ACM **31**(4), 761–791 (1984)
33. Khayyat, Z., Ilyas, I.F., Jindal, A., Madden, S., Ouzzani, M., Papotti, P., Quiané-Ruiz, J., Tang, N., Yin, S.: Bigdansing: A system for big data cleansing. In: SIGMOD, pp. 1215–1230 (2015)
34. Kolahi, S., Lakshmanan, L.V.S.: On Approximating Optimum Repairs for Functional Dependency Violations. In: ICDT, pp. 53–62 (2009)
35. Koudas, N., Saha, A., Srivastava, D., Venkatasubramanian, S.: Metric functional dependencies. In: ICDE, pp. 1275–1278 (2009)
36. Loshin, D.: Master Data Management. Knowl. Integrity, Inc. (2009)
37. Rekatsinas, T., Chu, X., Ilyas, I.F., Ré, C.: Holoclean: Holistic data repairs with probabilistic inference. PVLDB **10**(11), 1190–1201 (2017)
38. Saha, B., Srivastava, D.: Data Quality: The Other Face of Big Data. In: ICDE, pp. 1294–1297 (2014)
39. Song, S., Chen, L.: Differential dependencies: Reasoning and discovery. ACM Trans. Database Syst. **36**(3) (2011)
40. Staworko, S., Chomicki, J., Marcinkowski, J.: Prioritized Repairing and Consistent Query Answering in Relational Databases. Ann. Math. Artif. Intell. **64**(2-3), 209–246 (2012)
41. Volkovs, M., Chiang, F., Szlichta, J., Miller, R.J.: Continuous data cleaning. In: ICDE (2014)
42. Wang, J., Tang, N.: Towards dependable data repairing with fixing rules. In: SIGMOD (2014)
43. Wijsen, J.: Database repairing using updates. ACM Trans. Database Syst. **30**(3), 722–768 (2005)
44. Yakout, M., Berti-Équille, L., Elmagarmid, A.K.: Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In: SIGMOD, pp. 553–564 (2013)
45. Yakout, M., Elmagarmid, A.K., Neville, J., Ouzzani, M., Ilyas, I.F.: Guided data repair. PVLDB **4**(5), 279–289 (2011)