Mapping and Cleaning: the LLUNATIC Way

Floris Geerts\textsuperscript{1}, Giansalvatore Mecca\textsuperscript{2}, Paolo Papotti\textsuperscript{3}, Donatello Santoro\textsuperscript{2,4}

\textsuperscript{1}University of Antwerp – Antwerp, Belgium
floris.geerts@ua.ac.be

\textsuperscript{2}Università della Basilicata – Potenza, Italy
giansalvatore.mecca@gmail.com

\textsuperscript{3}Qatar Computing Research Institute (QCRI) – Doha, Qatar
ppapotti@qf.org.qa

\textsuperscript{4}Università Roma Tre – Roma, Italy
donatello.santoro@gmail.com

LLUNATIC Technical Report – University of Basilicata – # 01-2014

July 23, 2014

Abstract

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

1 Introduction

This paper discusses two important problems in database research, namely to integrate and transform data coming from different repositories using \textit{schema mappings}, and to study the quality of the resulting database using \textit{declarative constraints}. Schema mappings are executable transformations that specify how an instance of a source repository should be translated into an instance of a target repository. A rich body of research has investigated mappings [14]. However, it is also well known that data often contain inconsistencies, and that dirty data incurs economic loss and erroneous decisions [15]. The \textit{data-cleaning} (or \textit{data-repairing}) process consists in removing inconsistencies with respect to some set of constraints over the target database.

We may say that both schema-mappings and data repairing are long-standing research issues in the database community. However, so far they have been essentially studied in isolation. On the contrary, we notice that whenever several possibly dirty databases are put together by schema mappings, there is a very high probability that inconsistencies arise due to conflicts and errors in the source data, and therefore there is even more need for cleaning. In fact, bringing together schema mappings and data repairing is considered an open problem [27]. Solving this problem is far for trivial. To illustrate why, we next introduce our motivating example.

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

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

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

**Using Mapping to Exchange Data** Let us first discuss data exchange via mappings. To move data from the sources to the target, users may specify declarative schema mappings. As it is common, the mappings for our example are Let us first discuss data exchange via mappings. To move data from the sources to the target, users may specify declarative schema mappings. As it is common, the mappings for our example are

![Diagram showing data exchange via mappings]

Each tgd states a constraint over the target database. For example, tgd $m_2$ says that for each tuple in the MedTreatments source table, there must be corresponding tuples in the Customers and Treatments target tables. Correspondences are then translated into a set of tuple generating dependencies (tgds) [14].

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

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

   $m_2. \text{MedTreat}(\text{ssn}, \text{name}, \text{phn}, \text{conf}, \text{str}, \text{city}, \text{ins}, \text{treat}, \text{date})$
   $\rightarrow \exists Y_3, Y_4 : \text{Cust}(\text{ssn}, \text{name}, \text{phn}, \text{conf}, \text{str}, \text{city}, Y_3), \text{Treat}(\text{ssn}, Y_4, \text{ins}, \text{treat}, \text{date})$

Each tgd states a constraint over the target database. For example, tgd $m_2$ says that for each tuple in the MedTreatments source table, there must be corresponding tuples in the Customers and Treatments target tables; $Y_i$ are existential variables representing values that are not present in the source database but must be present in the target. The tgds also copy confidence values for source attributes into target tuples.

**Target Constraints** Besides deciding how to populate the target in order to satisfy the s-t tgds, we also need to generate instances that comply with target constraints. Traditionally [1], database architects have specified constraints of two forms: inclusion constraints and functional dependencies. These are expressible under the form of target tgds and equality generating dependencies (egds) [14].

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

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

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

$$e_1. \text{Cust}(ssn, n, p, s, c, cc), \text{Cust}(ssn, n, p', s', c', cc') \rightarrow p = p'$$

$$e_2. \text{Cust}(ssn, n, p, s, c, cc), \text{Cust}(ssn, n, p', s', c, cc') \rightarrow cc = cc'$$

$$e_3. \text{Cust}(ssn, n, p, s, c, cc), \text{Cust}(ssn', n, p', s, c, cc') \rightarrow ssn = ssn'$$

In the target instance shown in Figure 1, the pair of tuples $\{t_5, t_6\}$ violates $d_1$ and $d_2$; the database is thus dirty. It is worth noting that, aside from inclusion constraints and functional dependencies, the recent literature has shown that more advanced forms of constraints are often necessary in data cleaning applications [15]. To model these, new forms of data-quality constraints have been introduced. Here we mention conditional functional dependencies [16], conditional inclusion dependencies [15], and editing rules [17]. In our example, we incorporated the following data-quality constraints.

(4) CFDs: First, we assume two conditional functional dependencies (CFDs). A CFD $d_4 = (\text{Insur}[Abx] \rightarrow \text{Treat}[Dental])$ on table Treatments, expressing that insurance company ‘Abx’ only offers dental treatments (‘Dental’). Tuple $t_8$ violates $d_4$, adding more dirtiness to the target database.

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

Finally, as it is common in corporate information systems [26], an additional master-data table is available in the source database; this table contains highly-curated records whose values have high accuracy and are assumed to be clean. We also assume an additional constraint to clean target tuples based on values in the master-data table:

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

In summary, our example is such that:

- it requires to map different source databases into a given target database;
- it allows the target database to be non-empty, and that both the source and the target instances may contain errors and inconsistencies;
- it comes with a variety of data-quality constraints over the target; these include the common inclusion and functional dependencies but also richer constraints, like conditional dependencies and master-data-based editing rules.

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

First, we notice that data exchange [14] provides an elegant semantics for the execution of the source-to-target mappings (item (1) in our example). It also incorporates techniques to generate target instances that comply with target tgd and target egds (items (2) and (3)). Data exchange concentrates on soft violations. We call a soft violation for an egd any violation of the egd that can be removed by replacing one or more null values, either by a constant or by another null value. Consider for example the Customers table in the target database. Assume we have a constraint that states that $ssn$ is a key for the table; we notice that tuples $t_5, t_6$ violate the constraint. The conflict between street names is a soft violation, since $t_5, \text{Str} = \text{null}, t_6, \text{Str} = ‘Fry Dr.’$. On the contrary, as we discussed above, we assume that the data may also contain hard violations, i.e., violations that may only be repaired by changing one constant into another constant. In our example this happens with credit card numbers, since $t_5, \text{CC#} = 781658, t_6, \text{CC#} = 771859$. In case of hard violations, a data exchange scenario has no solution.

In fact, solving hard violations is the primary goal of data repairing algorithms. Early works on database repairing [5] modeled repairs for data quality constraints as sets of tuple-insertions and tuple-deletions. Since tuple-deletions may result in unnecessary loss of information, recent works have concentrated on cell changes [15], each cell being an attribute of a tuple. In this respect, the recent literature has provided us with a good arsenal of approaches and techniques to repair conflicts. In this paper, we want to capitalize on this wealth of knowledge about the subject, and investigate the following foundational problem: what should a database administrator do when facing a complex data-repairing problem that requires to bring together different data-quality constraints, as discussed above? A second, striking observation is that, despite many studies on the subject, there is currently no way to handle scenarios like the one in our example. The main
problem is the lack of a uniform formalism to handle different data repairing constraints. In fact, although repairing strategies exist for each of the individual classes of constraints discussed at items (2)–(6), there is currently no formal semantics for their combination.

The third important observation is that simply pipelining existing techniques does not work in general. Indeed, we might consider a multi-step process in which:

– we first use the standard semantics of data exchange [14] to generate an instance of the target that is a pre-solution for the tgds at items (1), (2), as shown in Figure 2;

– then, we use a combination of known data repairing algorithms for the constraints at (3), (4), (5), (6), like those in [9] or [8], to repair the target.

<table>
<thead>
<tr>
<th>CUSTOMERS</th>
<th>CUSTOMERS</th>
<th>CUSTOMERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Name</td>
<td>Phone</td>
</tr>
<tr>
<td>t₁</td>
<td>M. White</td>
<td>408-3334</td>
</tr>
<tr>
<td>t₂</td>
<td>L. Lennon</td>
<td>122-1876</td>
</tr>
<tr>
<td>t₃</td>
<td>L. Lennon</td>
<td>000-0000</td>
</tr>
<tr>
<td>t₁₀</td>
<td>W. Smith</td>
<td>324-0000</td>
</tr>
<tr>
<td>t₁₁</td>
<td>W. Smith</td>
<td>324-3456</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TREATMENTS</th>
<th>TREATMENTS</th>
<th>TREATMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Salary</td>
<td>Insur.</td>
</tr>
<tr>
<td>t₁</td>
<td>111</td>
<td>10K</td>
</tr>
<tr>
<td>t₁₁</td>
<td>111</td>
<td>25K</td>
</tr>
<tr>
<td>t₁₂</td>
<td>222</td>
<td>30K</td>
</tr>
<tr>
<td>t₁₃</td>
<td>222</td>
<td>null</td>
</tr>
<tr>
<td>t₁₄</td>
<td>222</td>
<td>null</td>
</tr>
</tbody>
</table>

Figure 2: Pre-solution and solutions.

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

Notice also that data quality constraints interact with each other in nontrivial ways. To see this, consider dependencies \( d_1 \) and \( d_5 \). Suppose we use \( d_1 \) to repair tuples \( t_5, t_6 \) in Figure 1 such that both have phone-number ‘122-1876’; then, since \( t_5 \) and \( t_6 \) agree with the master-data tuple \( t_m \), we can use \( d_5 \) to fix names, streets and cities, to obtain: (222, F. Lennon, 122-1876, Sky Dr., SF, 781658), for \( t_5 \), and (222, F. Lennon, 122-1876, Sky Dr., SF, 784659), for \( t_6 \). If, on the contrary, we apply \( d_5 \) first, only \( t_5 \) can be repaired as before; then, since \( t_5 \) and \( t_6 \) do not share the same name anymore, \( d_1 \) has no violations. We thus get a different result.

Based on these observations, in this paper we tackle the complex problem of defining a single, uniform framework for mapping and cleaning, and present LLUNATIC, the first comprehensive proposal to handle schema mappings and data repairing in a uniform way.

2 Contributions

A Uniform Framework for Mapping and Cleaning We develop a general framework for mapping and cleaning that can be used to generate solutions to complex data transformation scenarios, and to repair conflicts and inconsistencies with respect to a very wide class of constraints.

The framework incorporates most of the features considered in the well-known framework of data exchange, and in existing algorithms for data repairing; at the same time, it considerably extends its reach in both activities; in fact, on the one side it brings a powerful addition to schema mappings, by allowing for sophisticated conflict resolution strategies,
in this kind of problems, we assume that the attribute values should be repaired: ‘122-1876’ or ‘000-0000’, or even a completely different value. As it happens, the problem consists in picking-up a “preferred” value to repair the database. Consider FD $d_1$ in our example. To repair the target database one may want to equate $t_5[\text{Phone}]$ and $t_6[\text{Phone}]$. The FD does not tell, however, to which phone number these attribute values should be repaired: ‘122-1876’ or ‘000-0000’, or even a completely different value. As it happens in this kind of problems, we assume that the Phone attribute values in the Customers table come with a confidence $(Conf.)$ value (‘122-1876’ has confidence 0.9 and the dummy value ‘000-0000’ has confidence 0). Relying on confidence is a typical strategy to select preferred values: if we assume that one prefers values with higher confidence, we can repair $t_5[\text{Phone}]$ by changing it to ‘122-1876’.

There are, however, other possibilities. For example, in the Treatments table, we may use dates of treatments to infer the currency of other attributes. If the target database is required to store the most recent value for the salary by FD $d_2 = (\text{SSN} \rightarrow \text{Salary})$, this may lead us to repair the obsolete salary value ‘10K’ in $t_7$ with the more recent (and preferred) value ‘25K’ in $t_8$. Notice that we do not always have a clear policy to choose preferred values. For example, when repairing $t_5[\text{CC#}]$ and $t_6[\text{CC#}]$ for FD $d_2$, there is no information available to resolve the conflict. Hence, the best we can do is to “mark” the conflict, and then, perhaps, ask for user-interaction to solve it.

Previous algorithms have proposed a variety of strategies to select preferred values, like the ones discussed above. However, these algorithms tend to hard-code the way in which preferred values are used for the purpose of repairing the database. As a consequence, there is no way to incorporate the different strategies in a uniform and principled way.

Our solution to this problem builds on two main concepts. First, we show that seeing repairs simply as cell updates is not sufficient. On the contrary, we introduce the new notion of a cell group, that is essentially a “partial repair with lineage” (Sections 6, 7, 8). In addition, we formalize the process of improving the quality of a database by introducing a very general notion of a partial order over cell groups (Sections 9 and 10): the partial order nicely abstracts most of the typical strategies to decide when a value should be preferred to another, including master data, certainty, accuracy, freshness and currency. In the paper, we show how users can easily plug-in their preference strategies for a given scenario into the semantics (Sections 11, 12). Finally, by introducing a new category of values, called lluns, we are able to complete the lattice of cells induced by the partial order, and to provide a natural hook for incorporating user feedbacks into the process.

The Chase and Scalability We introduce the notion of a minimal solution and develop algorithms to compute minimal solutions, based on a parallel-chase procedure (Section 14). This has the advantage of building on a popular and principled algorithmic approach, but it has a number of subtleties. In fact, our chase procedure is more general than the standard one. To give an example, in the presence of inconsistencies user inputs may be crucial. To this aim, we introduce a nice abstraction of user inputs and show how it can be seamlessly integrated into the chase.

In addition, scalability is a primary concern of this work. We introduce a notion of a cost manager as a plug-in for the chase algorithm that selects which repairs should be kept and which ones should be discarded (Section 15). The cost manager abstracts and generalizes several popular solution-selection strategies, including similarity-based cost, cardinality minimality, certain regions, sampling, among others. In Example 1, our semantics generates minimal solutions like the two solutions in Figure 2, where $L_i$ values represent lluns (confidence values have been omitted); notice that other minimal solutions exist for this example. Cost managers allow users to differentiate between these two solutions, which have different costs in terms of chase computation, and ultimately to fine-tune the tradeoff between quality of solutions and scalability of the repair process.

We show that the chase engine scales to databases with millions of tuples, a considerable advancement in scalability wrt previous main-memory implementations (Section 17). In fact, we have shown in our experiments that the chase engine is significantly faster than existing engines for data exchange [35], and shows superior scalability wrt previous repair algorithms [9, 8] that were designed to run in main memory.

Generality of the Framework We compare our semantics to many previous approaches (Sections 13 and 16). To the best
of our knowledge, this is the first proposal that achieves the level of generality needed to handle three different kinds of problems: traditional mapping problems, traditional data repairing problems, and the new and more articulated category of data translation problems with conflict resolution, as exemplified in Example 1. We believe that these contributions make a significant advancement with respect to the state-of-the-art, and may bring new maturity to both schema mappings and data repairing.

This paper generalizes and extends results that were previously published in conference papers. We make several advancements with respect to the conference versions:

1. we completely rework the semantics of the framework (Sections 7–11), and provide an extensive and completely new treatment of the critical notion of the framework, i.e., cell groups and their partial order. To start, we simplify the definition of a cell group wrt the one given in [20, 21]. On the one side, this allows us to provide an elegant set-theoretic characterization of their partial order in terms of cell containment. On the other side, this suggests a graphical explanation of the working of cell groups, which we use extensively throughout the paper to clarify the technical development (Figures 4, 5, 6). By leveraging these tools, we are able to give an extended treatment of what was presented earlier in a more compact form, and to develop the main notions in a comprehensive and more readable way; given the richness and complexity of the framework, we felt that this was a needed extension; a more detailed comparison of the two formalisms is in Appendix B, proof of Theorem 5;

2. we investigate in depth the chase algorithm, another component at the foundations of the mapping and repairing algorithms (Sections 14, 15, Appendix A); we provide a detailed development of two variants of the chase; the first one is a natural extension of the standard chase to our setting; the second one is a variant that is technically more involved but provides a basis to reason about scalability; we prove that the two variants are both correct and are equivalent;

3. we develop an in-depth comparison of our semantics to previous semantics, both for data exchange and for data repairing (Sections 13 and 16), in order to substantiate our statement that mapping & cleaning unifies many of the previous approaches;

4. we provide an extensive experimental evaluation (Section 18) that covers scenarios of various kinds and proves the good scalability of our approach;

5. we develop full proofs of all theorems (Appendix B).

3  Background

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

**Dependencies** A relational atom over $R$ is a formula of the form $R(\pi)$ with $R \in R$ and $\pi$ is a tuple of (not necessarily distinct) variables. A tuple-generating dependency (tgd) over $R$ is a formula of the form $\forall \pi (\phi(\pi) \rightarrow \exists \gamma \psi(\pi, \gamma))$, where $\phi(\pi)$ and $\psi(\pi, \gamma)$ are conjunctions of relational atoms over $R$. Given two disjoint schemas, $S$ and $T$, a tgd over $\langle S, T \rangle$ is called a source-to-target tgd ($s$-t tgd) if $\phi(\pi)$ only contains atoms over $S$, and $\psi(\pi, \gamma)$ only contains atoms over $T$. Furthermore, a target tgd is a tgd in which both $\phi(\pi)$ and $\psi(\pi, \gamma)$ only contain atoms over $T$. An equality generating dependency (egd) over $T$ is a formula of the form $\forall \pi (\phi(\pi) \rightarrow x_i = x_j)$ where $\phi(\pi)$ is a conjunction of relational atoms over $T$ and $x_i$ and $x_j$ occur in $\pi$.

We assume the standard definition [14] of a mapping scenario $\mathcal{M} = \{S, T, \Sigma_{st}, \Sigma_{t}\}$, where $S$ is a source schema, $T$ is a target schema, $\Sigma_{st}$ is a set of s-t tgds, and $\Sigma_{t}$ is a set of target tgds and egds, and corresponding notions of solution, universal solution, and core solution as the smallest of the universal solutions for a mapping scenario $\mathcal{M}$ over instances $\langle I, J \rangle$ of $\langle S, T \rangle$. We also assume the standard definition of the chase of a mapping scenario $\mathcal{M}$ over $\langle I, J \rangle$. 


4 Extended Dependencies

A first, important step in the definition of our framework for mapping and cleaning consists in the definition of a unified language for mappings and data quality constraints. Traditional embedded dependencies are extensively studied and used when integrating and exchanging data. They fall short, however, when it comes to the constraint formalisms used in the context of data quality. For example, the constraints $d_4$, $d_5$ and $d_6$ described in Example 1 cannot be directly expressed as egds.

To alleviate this problem, we make use of extended tgds and cleaning egds. In particular, we freely mix source and target symbols in the premise. In addition, in a cleaning egd, we also consider equation atoms of the form $t_1 = t_2$, where $t_1, t_2$ are either constants in const or variables.

Egds for our running example are expressed as follows:
\begin{align*}
& e_1. \text{Cust}(ssn, n, p, s, c, cc), \text{Cust}(ssn, n, p', s', c', cc') \rightarrow p = p' \\
& e_2. \text{Cust}(ssn, n, p, s, c, cc), \text{Cust}(ssn, n, p', s', c', cc') \rightarrow cc = cc' \\
& e_3. \text{Cust}(ssn, n, p, s, c, cc), \text{Cust}(ssn', n, p', s', c', cc') \rightarrow ssn = ssn' \\
& e_4. \text{Treat}(ssn, s, \text{ins}, \text{tr}, d), \text{ins} = 'Abx' \rightarrow \text{tr} = 'Dental' \\
& e_5. \text{Cust}(ssn, n, p, s, c, cc), \text{MD}(ssn, n', p, s', c') \rightarrow n = n' \\
& e_6. \text{Cust}(ssn, n, p, s, c, cc), \text{MD}(ssn', n, p, s', c') \rightarrow s = s' \\
& e_7. \text{Cust}(ssn, n, p, s, c, cc), \text{MD}(ssn', n', p, s', c') \rightarrow c = c' \\
& e_8. \text{Cust}(ssn, n, p, s, c, cc), \text{Treat}(ssn, \text{sal}, \text{ins}, \text{tr}, d), \text{ins} = 'Abx' \rightarrow c = 'SF' \\
& e_9. \text{Treat}(ssn, s, \text{ins}, \text{tr}, d), \text{Treat}(ssn', s', \text{ins}', \text{tr}', d') \rightarrow s = s' \\
\end{align*}

We formalize the notion of a cleaning egd as follows:

**Definition 1 [Cleaning EGD]** A cleaning egd over schemas $S$, $T$ is a formula of the form $\forall \pi(\phi(\pi) \rightarrow t_1 = t_2)$ where $\phi(\pi)$ is a conjunction of relational atoms over $(S, T)$, and $t_1 = t_2$ is of the form $x = c$ or $x = x_j$, for some variables $x_i, x_j$ in $\pi$ and constant $c \in \text{CONSTs}$. Furthermore, at most one variable in the conclusion of an egd can appear in the premise as part of a relation atom over $S$.

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

An immediate observation is that constants and equation atoms in the premise can be avoided altogether, by encoding them in additional tables in the source database. Consider dependency $e_4$ in which two constants appear: ‘Abx’ in attribute Insur and ‘Dental’ in attribute Treat. We extend $S$ with an additional binary source table, denoted by $\text{Cst}_{e_4}$ with attributes Insur and Treat, corresponding to the “constant” attributes in $e_4$. Furthermore, we instantiate $\text{Cst}_{e_4}$ with the single tuple $t_{e_4} : (\text{Abx}, \text{Dental})$. Given this, $e_4$ can be expressed as an egd without constants. Similarly for $e_8$, as follows:
\begin{align*}
& e_4'. \text{Treat}(ssn, s, \text{ins}, \text{tr}, d), \text{Cst}_{e_4}(\text{ins}, \text{tr}') \rightarrow \text{tr} = \text{tr}' \\
& e_8'. \text{Cust}(ssn, n, p, s, c, cc), \text{Treat}(ssn, \text{sal}, \text{ins}, \text{tr}, d), \text{Cst}_{e_8}(\text{ins}, c') \rightarrow c = c' \\
\end{align*}

In general, $S$ can be extended with such constant tables, one for each dependency in which constants appear. This is a natural way to model CFDS by explicitly encoding their pattern tableaux [16], and we shall use it to remove constants from dependencies. Of course, one needs to provide a proper semantics of egds such that whenever such constant tables are present, egds have the same semantics as CFDS. We give such semantics later in the paper.

A similar treatment is also done for tgds. To properly encode data quality constraints, like, for example, conditional inclusion dependencies [27], we need to be able to specify equation atoms in the dependency premise. Suppose, for example, that our target database also contains a DentalProsthesis table, in which we store all prosthesis that have been installed as part of dental treatments. We may want to specify a conditional inclusion dependency of this form:

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

As before, we encode constants as special source relations. In light of this, we define the notion of an extended tgd:

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

Notice that extended tgds generalize both traditional s-t tgds and target tgds. Further extensions of dependencies with, e.g., built-in predicates, matching functions and negated atoms, are needed to encode matching dependencies [6] and constraints for numerical attributes [19]. We leave these to future work.
In the following sections, we formalize the notion of a mapping and cleaning scenario with extended tgds and cleaning egds, and then provide a semantics for it. Before we turn to this, we need to introduce an important result, that motivates the need for a new semantics.

One may wonder why a new semantics is needed after all. Indeed, why can’t we simply rely on the standard semantics for tgds [14], and on known data repairing algorithms, like those in [20], [9] or [8]? As an example, let $\Sigma_t$ be a set of tgds and $\Sigma_e$ be a set of egds, and $I$ and $J$ instances of a source and target schema, $S$ and $T$, respectively. Assume that we simply pipeline the chase of tgds, chase$_{\Sigma_t}$, [14], and a repair algorithm for egds, repair$_{\Sigma_e}$, treated as functional dependencies, as reported in Figure 3.

$$\text{pipeline}_{\Sigma_t \cup \Sigma_e}((I, J))$$

$$(I, J_{\text{tmp}}) := (I, J);$$

while (true)

$$(I, J_{\text{tmp}}) := \text{chase}_{\Sigma_t}((I, J_{\text{tmp}}));$$

$$(I, J_{\text{tmp}}) := \text{repair}_{\Sigma_e}((I, J_{\text{tmp}}));$$

if ($$(I, J_{\text{tmp}}) \models \Sigma_t \cup \Sigma_e$$) return $S_{\text{tmp}} := J_{\text{tmp}};$$

end while

Figure 3: The pipeline algorithm.

Unfortunately, interactions between tgds and egds often prevent that pipelining the two semantics returns a solution, as illustrated by the following proposition (proofs of all theorems are in the Appendix).

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

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

## 5 Mapping and Cleaning Scenarios

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

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

We denote by $\text{LLUNS} = \{L_0, L_1, L_2, \ldots\}$ an infinite set of symbols, called lluns, distinct from consts and nulls. Lluns can be regarded as the opposite of nulls since lluns carry “more information” than constants. In our approach, they play two important roles: (i) they allow us to complete the lattice induced by our partial orders, as it will be discussed in the next section; (ii) they provide a clean way to record inconsistencies in the data that require the intervention of users.

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

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

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

$\square$
Note that User is by definition a partial function, and it may thus be undefined for some sets of cells.

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

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

We say that an attribute A of T has ordered values if its domain DA is a partially ordered set. To specify which values should be preferred during the repair of the database, users may associate with each attribute Ai of T a partially ordered set PAi = ⟨DAi, ≤⟩. The poset PAi associated with attribute Ai may be the empty poset, or its domain DAi, if Ai has ordered values, or the domain of a different attribute DAj, that has ordered values. In the latter case, we call Aj the ordering attribute for Ai. Intuitively, PAi specifies the order of preference for values in the cells of Ai. A partial-order specification is an assignment of ordering attributes to attributes in T, denoted by Π.

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

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

\[
Π = \left\{ \begin{array}{l}
PA_{CUSTOMERS.CONF} = DA_{CUSTOMERS.CONF} \\
PA_{TREATMENTS.DATE} = DA_{TREATMENTS.DATE} \\
PA_{CUSTOMERS.PHONE} = DA_{CUSTOMERS.CONF} \\
PA_{TREATMENTS.SALARY} = DA_{TREATMENTS.DATE} \\
PA_{CUSTOMERS.C#} = ∅
\end{array} \right\}
\]

Notice that a similar treatment can also be done for the source schema S. More specifically, we assume two source schemas, S and Sa. This second schema, Sa, is a set of authoritative tables that provide clean and reliable information as input for the repairing process. This include master-data tables and constant tables introduced to remove constants from dependencies (as discussed in Section 4). Authoritative tables are considered all as equally reliable, since they contain “certified” tuples. On the contrary, a partial order specification can be specified on the attributes of S in order to state preference relations on source values as well. In our example Π also includes:

\[
\left\{ \begin{array}{l}
PA_{PATIENTS.PHONE} = DA_{PATIENTS.CONF} \\
PA_{MEDTREATMENTS.PHONE} = DA_{MEDTREATMENTS.CONF}
\end{array} \right\}
\]

The role of the partial order specification is to induce a partial order for the cells of the initial instance, ⟨I, J⟩. This will be detailed in Section 10.

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

Definition 4 [Mapping & Cleaning Scenario] Given a domain D = Constants ∪ Nulls ∪ LLUNS, a mapping & cleaning scenario over D is a tuple MC = {S, Sa, T, Σt, Σc, Π, User}, where:

1. S ∪ Sa is the source schema; T is the target schema; Sa is the set of authoritative source tables;
2. Σt is a set of extended tgds, as defined in Section 3;
3. Σc is a set of cleaning egds, as defined in Section 3;
4. Π is a partial-order specification for attributes of S and T;
5. User is a partial function as defined in Definition 3.
If the set of tgds, $\Sigma_t$, is empty, $\mathcal{MC}$ is called a cleaning scenario.

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

### 6 The Universe of Cells and Its Partial Order

The main technical tool used in the definition of our semantics are cell groups, introduced in the next section. Before we turn to the formal definition of cell groups, we first formalize the universe of cells associated with a mapping & cleaning scenario $\mathcal{M}$. In addition, we will show how $\mathcal{M}$ induces a partial order over this universe, that we leverage to reason about the notion of “upgrade” to the target database.

As we saw in Section 5, we are given a source database instance, $I$, a subset of which, $I_a$, consists of authoritative tables, and a target instance, $J$. Our universe of cells is composed primarily of the cells of $I$ and $J$. In the following we will refer to the set of non-authoritative tables in $I$ by the symbol $I_{na}$, i.e., $I = I_{na} \cup I_a$. We find it useful to introduce a few additional cells.

(i) **Cells for Constant Symbols in Dependencies**: First, as discussed in Section 4, we simplify the treatment of constants within dependencies by assuming that each constant value $c$ within egd $e$ is assigned a cell of the source database. More specifically, we assume the presence of a table $\text{Cst}_{e,c}$ with a single attribute, Val, within the authoritative source schema, $\mathcal{S}_a$. The table contains a single tuple, $t_{e,c}$, such that cell $t_{e,c}.\text{Val} = c$. As a consequence, from now on these cells are considered as part of $I_a$.

(ii) **New Cells**: We assume the availability of an (infinite) set of cells, denoted by new-cells($J$). Intuitively, new-cells($J$) are used to model possible insertions in $J$ needed to satisfy tgds. We assume that each cell in new-cells($J$) is initialized to a different null value from NULLS.

(iii) **Metacells**: Finally, we introduce a set of metacells. Metacells are special cells that encode specific aspects of the repair process. For the purpose of this paper, we assume a set of meta cells, metaCells, with two elements:

- the invalid cell, $c_x$, with value $\times$; as it will be shown in the next sections, the invalid cell denotes those updates to the target database in which one or more values are considered invalid and need to be replaced by a llun;
- the user cell, $c_T$, with value $\top$; similarly, the user cell denotes those updates whose value comes from the user, the most reliable source of repairs in our approach.

We define the universe of cells of $\mathcal{M}$, $\text{cells}(\mathcal{M}, <I, J>)$, as the set $\text{cells}(I) \cup \text{new-cells}(J) \cup \text{cells}(I_{na}) \cup \text{cells}(I_a) \cup \{c_x, c_T\}$. Recall that $\text{cells}(I_a)$ also contains cells for constants in dependencies. Our purpose it to introduce a partial order within the universe of cells.

The **Partial Order of Cells** based on the partial order specification, $\Pi$, provided by the user as part of a scenario $\mathcal{M}$, we now derive a partial order $\preceq_{\Pi}$ of the cells in $\text{cells}(\mathcal{M}, <I, J>)$. This is crucial to plug-in arbitrary preference strategies during the repair process. Recall that cells in $<I, J>$ may only contain constants and null values, while cells in new-cells($J$) are initially set to null values.

The partial order of cells is based on a number of intuitive rules, as follows:

- first, in analogy to data exchange, we consider cells with a constant value preferable to those with a null value;
- then, cells in $I_{na}$ and $J$ are ordered according to the partial order specification $\Pi$ provided as part of the mapping & cleaning scenario; this may be empty; if it is not empty, it allows to plug into the partial order arbitrary preference strategies, as we discussed in Example 1;
- the invalid cell is incomparable to any other cell in $I_{na}, J$; in fact, it denotes that one or more of these cells carry invalid values, and therefore we cannot rely on their value in the partial order.
• authoritative cells in $I_a$ are better than any target or (non-authoritative) source cell, and also better than the invalid cell;
• finally, the user cell trumps any other cell.

These rules are formalized in the following definition, and sketched in Figure 4, where $c_i$ denotes a constant, and $N_j$ a null value.

**Definition 5 [PARTIAL ORDER OF CELLS]** Given a mapping and cleaning scenario $M$ with partial-order specification $\Pi$, and an instance $(I, J)$, we can define a partial order $\preceq_{\Pi}$ for the set of cells $\text{cells}(M, (I, J)) = \text{cells}(J) \cup \text{new-cells}(J) \cup \text{cells}(I_{na}) \cup \text{cells}(I_a) \cup \text{metaCells}$ as follows. For any pair of cells $c_1, c_2 \in \text{cells}(M, (I, J))$ we say that $c_1 \preceq_{\Pi} c_2$ iff either $c_1 = c_2$ or one of the following holds:

1. $\text{val}(c_1) \in \text{NULLS}$, and $\text{val}(c_2) \in \text{CONSTS}$;
2. $c_1$ and $c_2$ are ordered according to $\Pi$, that is: $c_1 = t_1.A_1$, $c_2 = t_2.A_2$ in $(I, J)$, and both are constants in $\text{CONSTS}$; then, assume the ordering attributes for $A_1$ and $A_2$, called $A'_1$, $A'_2$ have the same poset, i.e., $P_{A'_1} = P_{A'_2}$; call $v'_1, v'_2$ the values of cells $t_1.A'_1$, $t_2.A'_2$. Then, $c_1 \preceq_{\Pi} c_2$ iff $v_1 = v_2$ or $v'_1 < v'_2$ according to $P_{A'_1} = P_{A'_2}$;
3. $c_1 \notin \text{cells}(I_a) \cup \{c_T\}$, while $c_2 \in \text{cells}(I_a)$;
4. $c_2 = c_T$.

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

**Proposition 2** The binary relation $\preceq_{\Pi}$ as specified in Definition 5 is a partial order.

### 7 Cell Groups

Given instance $(I, J)$ of $(S, S_a, T)$, we represent the set of changes made to update the target database $J$ in terms of cell groups encoding the modification. Cell groups are sets of cells. We find it useful to divide the cells of a cell group in three subsets, called occurrences, justifications, and metaCells. These are introduced in the following paragraphs.

**Occurrences:** As the name suggests, cell groups are essentially groups of cells, i.e., locations in a database specified by tuple/attribute pairs $t_{i\text{d}}, A_i$. For example, $t_5.CC#$ and $t_6.CC#$ are two cells in the Customers table. As we have previously seen, to repair inconsistencies, different cells are often updated together, i.e., they are either changed all at the same time...
or not changed at all. For example, $t_5$.CC# and $t_6$.CC# are both modified to the same llun value in solution #1 in Figure 2.

Cell groups capture this by specifying a set of target cells, called *occurrences* of the group, and a value to update them.

**Justifications:** However, cell groups are more sophisticated than that since they also model relationships among target and source values. In some cases the target cells to repair receive their value from tuples in the source database; consider Example 1 and dependency $e_6$. When repairing $t_5$, cell $t_5$.Street gets the value ‘Sky Dr.’ from cell $t_m$.Street in the master-data table. Since these source cells contain highly reliable information, it is important to keep track of the relationships among changes to target cells and values in the source. To do this, a cell group carries provenance information about the repair in terms of associated cells of the source database, called *justifications* for the cell group. Occurrences and justifications need to be kept separate since we can only update target cells, while source cells are immutable. Justifications provide provenance information for changes to the database, cell groups can be seen as partial updates with lineage.

**Metacells:** As we saw, there are two meta-cells, namely, the invalid cell, $c_x$, and the user cell, $c_T$. In our approach, invalid cells are used essentially to mark backward changes to the target instance. We notice that there are two different strategies to remove violations for a dependency. The ones we have discussed so far in our examples are called forward changes, since they amount to changing cells to satisfy the conclusion of an egd. Consider egd $e_2$ that states that customers with equal SSNs must have equal credit-card numbers. The forward change to solve a violation equates the values of the CC# attribute of the conflicting tuples. However, as an alternative, one may introduce backward changes to falsify the premise instead. In our example, this requires to set the value of attribute ssn in either of the conflicting tuples to a llun value to say that the original value is dirty, and should be changed to something else that we currently do not know. To mark these changes in such a way that they are handled appropriately within the partial order, we use cell groups with an invalid cell.

In addition, cell groups also provide an ideal basis to plug-in user-specified changes. In fact, our user function, User, works with cell groups to modify their values when needed. Ultimately, whenever our repairs introduce lluns, i.e., unknown values, within the target to handle conflicting values, users may step in and provide a constant to solve the conflict. In our framework, user-provided values are considered the ones with the highest level of accuracy. Therefore, we mark with the user cell the cell groups that have user-provided values.

These observations are captured by the following definitions. Let $\mathcal{M}$ be a mapping & cleaning scenario, and $\langle I, J \rangle$ be an instance of $\langle S, S_a, T \rangle$.

**Definition 6 [Cell Group]** A cell group $g$ over $\mathcal{M}$ and $\langle I, J \rangle$ is a triple $g = \langle \text{occ}(g), \text{just}(g), \text{metaCells}(g) \rangle$, where:

1. $\text{occ}(g)$ is a finite set of cells in $\text{cells}(J) \cup \text{new-cells}(J)$, called the occurrences of $g$; of these, we call $\text{target-cells}(g)$ the ones that appear in $J$, and $\text{new-cells}(g)$ the new ones from $\text{new-cells}(J)$;
2. $\text{just}(g)$ is a finite set of cells in $\text{cells}(I)$, called the justifications of $g$; of these, we call $\text{auth-cells}(g)$ the ones that belong to authoritative tables in $I_a$;
3. $\text{metaCells}(g)$ is possibly empty set of metacells, i.e., a subset of $\{c_x, c_T\}$.

We denote by $\text{cells}(g)$ the union of $\text{occ}(g)$, $\text{just}(g)$, $\text{metaCells}(g)$.

Each cell group $g$ has a value, denoted by $\text{val}(g)$. In fact, a cell group $g$ can be read as “change (or insert) the cells in $\text{occ}(g)$ to value $\text{val}(g)$, justified by the values in $\text{just}(g)$”. It remains to define what is the value of $g$. Intuitively, the value of $g$ is obtained by computing the least-upper bound of $\text{cells}(g)$ according to the partial order of cells, $\preceq_{\Pi}$, as discussed in the next section.

## 8 The Value of a Cell Group

As we have seen, cell groups are made of target cells (occurrences) and source cells (justifications), with the possible addition of metacells. We expect cell groups to “improve” and “generalize” the values that appear in $\text{cells}(g)$, according to the partial order over cells; in order to do this, we shall define the value of a cell group as the “upper-bound value” of its cells.

**Definition 7 [Value of a Cell Group]** Given a scenario $\mathcal{M}$, and a cell group $g = \langle \text{occ}(g), \text{just}(g), \text{metaCells}(g) \rangle$ over $\mathcal{M}$, we define the value of $C$, denoted by $\text{val}(g)$, as follows:

1. if the user-cell, $c_T \in \text{cells}(g)$, then consider the set of cells $C' = \text{cells}(g) - \{c_T\}$; then $\text{val}(g) = \text{User}(C')$, if this is defined and equal to a constant; if $\text{User}(C')$ is not defined, then $\text{val}(g)$ is a fresh llun value $L_i$;
2. otherwise, consider the set maximal-cells(\text{cells}(g)) of maximal elements in \text{cells}(g) wrt $\preceq_{\Pi}$. If all these maximal cells have exactly the same value $v \in \text{CONS}$, then $\text{val}(g)$ is exactly $v$;
3. otherwise, if all cells in \( \text{cells}(g) \) have a null value, then \( \text{val}(g) \) is a fresh null value \( N_i \);

4. otherwise \( \text{val}(g) \) is a fresh llun value \( L_j \).

In essence, we consider \( \text{cells}(g) \), and look for any maximal elements wrt the partial order of cells. Whenever there is a maximum cell, or all of the maximal elements have the same value \( v \) (up to the renaming of nulls and lluns), we take \( v \) as the value for \( g \). Whenever the maximal elements are incomparable, we use lluns to “complete” the lattice of cells. Special care needs to be taken to handle the user cell, that is always the maximum of \( \text{cells}(g) \) when it is present. In this case, we use the user function, User, to assign a value to \( g \).

Notice that we may also handle “inconsistent” user inputs. Consider for example the following case: at some point of the repair process the user provides a value \( v_1 \) for a group of cells (and therefore a cell group \( g_1 \) with the user cell is generated). Later on, however, we discover that these cells need to be made equal to the ones of a cell group \( g_2 \), that also contains the user-cell, and has value \( v_2 \). There is obviously a contradiction between the two user inputs. The resulting cell group will be the union of the cells of \( g_1, g_2 \), it will have the user-cell, but the user function might not be defined for it. In this case, the value of the new cell group will be a fresh llun.

In the following, we provide several examples of cell groups. For the sake of readability, we shall often write a cell group as:

\[ g = \langle v \rightarrow \text{occ}(g), \text{by just}(g), \text{with metaCells}(g) \rangle \]

(metacells will be omitted when absent). We use superscripts to denote new and authoritative cells. Subscripts are used to report the value of a cell in the original databases, \( I \) or \( J \), so that the modifications specified by cell groups are easier to interpret.

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

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

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

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

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

If we update the target with \( g_5 - g_6 \), the two new tuples \( t_3, t_4 \) violate the egd. A cell group that enforces the egd over cells \( t_3.B, t_4.B \) is as follows:

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

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

\[ g_8 : \langle L_2 \rightarrow \{ t_3.A^{\text{new}} \}, \text{by } \{ t_1.A[1], \text{with } \{ c_x \} \} \rangle \]

**Example 3**: Here are further examples of cell groups for Example 1:

\[ g_1 : \langle L_0 \rightarrow \{ t_5.CC[78659], t_6.CC[78659] \}, \text{by } \emptyset \rangle \]
\[ g_2 : \langle \text{Treat[Cholel] \rightarrow } \{ t_8.CC[78659] \}, \text{by } \{ t_4.CC[78659], \text{auth} \} \rangle \]
\[ g_3 : \langle \text{Med[Insur] \rightarrow } \{ t_2.CC[78659] \}, \text{by } \{ t_2.CC[78659], \text{auth} \} \rangle \]
\[ g_4 : \langle \text{null \rightarrow } \{ t_1.User \}, \text{by } \emptyset \rangle \]
\[ g_5 : \langle L_3 \rightarrow \{ t_7.User[Abx] \}, \text{by } \emptyset, \text{with } \{ c_x \} \rangle \]

Observe that \( g_1 \) changes two conflicting cells of the target database into a llun value. The justification in \( g_2 \) can be intuitively explained by seeing that the cell is repaired according to egd \( e_4 \), stating that company ‘Abx’ only provides dental treatments. Cell \( t.e_4.Treat[Cholel] \) is the authoritative cell associated with \( e_4 \) and constant \( Dental \). Cell groups \( g_3 \) and \( g_4 \) add new cells to the target, to insert (part of) tuple \( t_12 \). Notice how, in \( g_3 \), the value for the new cell is stored as a justification. Finally, \( g_5 \) is an example of a cell group that backward-changes a cell to satisfy dependency \( e_8 \).
We consider cell groups to be undistinguishable up to the renaming of nulls and lluns. In fact, we say that \emph{g is equal to g'} if \emph{cells(g) = cells(g')}, and: (i) both have the same constant value, i.e., \emph{val(g) = val(g')} \in \textsc{consts}, or (ii) \emph{val(g), val(g')} are both nulls or lluns.

Notice that, throughout the definition of the semantics, we always refer to the original value of a cell in \emph{cells(I) \cup cells(J) \cup new-cells(J)}, regardless of any modification that will be done to the target database to satisfy the constraints. This is a very important feature of our approach: to find solutions we compute upper-bound values with respect to a partial order of cells, \(\leq_f\), that is fixed in advance and cannot change. This guarantees that the repair process does not interfere with the strategy to pick-up preferred values for cell groups, and therefore no termination or confluence problems may arise [10] (further details on this aspect are provided in Section 16).

9 Updates

Given a scenario \(\mathcal{M}\), we define an \textit{update} to an instance \(\langle I, J \rangle\) as a set of cell groups over \(\mathcal{M}, \langle I, J \rangle\). More specifically,

**Definition 8 [Update]** An update \(\text{Upd}\) for \(\mathcal{M}, \langle I, J \rangle\) is a set of cell groups over \(\mathcal{M}, \langle I, J \rangle\) such that there exists a set of tuples \(\Delta J\), distinct from \(J\), for which:

1. for each \(g = \langle \text{occ}(g), \text{just}(g), \text{metaCells}(g) \rangle\) in \(\text{Upd}\), \(\text{occ}(g)\) is not empty, and \(\text{occ}(g) \subseteq \text{cells}(J \cup \Delta J)\). That is, cell groups in \(\text{Upd}\) are restricted to actual updates of cells in \(J \cup \Delta J\);

2. each cell in \(J\) and \(\Delta J\) occurs exactly once in \(\text{Upd}\). That is, two cell groups in \(\text{Upd}\) cannot update the same cell, and the updated instance is completely specified by \(\text{Upd}\);

3. for each pair of tuples \(t_1, t_2 \in \Delta J\) in table \(R\), there exists \(A\) such that \(\text{Upd}\) assigns different values to \(t_1.A, t_2.A\). That is, we never insert two new identical tuples into the target;

4. there are not two cell groups \(g_1, g_2 \in \text{Upd}\) such that \(\text{val}(g_1) = \text{val}(g_2) \in \text{nulls} \cup \text{lluns}\), i.e., each null (resp. llun) value occurs at most once as a value of a cell group in \(\text{Upd}\). That is, null and llun values uniquely identify the cells in which they appear;

5. there is no cell group \(g \in \text{Upd}\) such that \(\text{User}(\text{occ}(g) \cup \text{just}(g)) = \bot\), i.e., none of the cell groups of \(\text{Upd}\) is refused by the user function;

6. for each cell group \(g \in \text{Upd}\) such that \(\text{User}(\text{occ}(g) \cup \text{just}(g))\) is defined, it is the case that \(c_\top \in \text{metaCells}(g)\), i.e., whenever the user function is defined, cell groups take the value specified by the user.

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

Notice that, according to item (2) above, an update needs to specify a value for each cell of the target instance, possibly leaving the value of some cells unchanged. In the following, however, when reporting updates we shall often omit cell groups that do not change the original values of their occurrences.

An update never generates two new identical tuples in \(\Delta J\). This prevents the generation of duplicates when \(J\) is initially empty. Notice, however, that duplicate tuples can still arise by modifying the original cells in \(J\), if \(J\) is not empty. In this case, we assume that duplicate tuples are removed as a final step to generate \(\text{Upd}(J)\).

**Example 4:** Consider our Example 2 above. Following are two updates \(\text{Upd}_1, \text{Upd}_2\) that enforce the constraints:

\[
\Delta J = \{t_3 : T(1, 2, N_1), t_4 : T(1, 3, N_2)\}
\]

\[
\text{Upd}_1 = \{g_1, g_3, g_4, g_6, g_7\} \quad \text{Upd}_2 = \{g_2, g_3, g_4, g_5, g_6, g_8\}
\]

**Example 5:** Consider the table \text{Treatments} from Example 1 and corresponding table in solution \#1 shown in Figure 2. It is easy to see that the repaired instance can be seen as an update. For example, it can be regarded as \(\text{Upd}_1(\text{Treatments})\).
Following are some cell groups from $Upd_1$:

$$
g_1 : \langle Dental \to \{t_8.\text{Treat}_{\text{Cholest}}\}, \text{by} \ t_{e_4}.\text{Treat}_{\text{Dental}}^{\text{new}} \rangle
$$

$$
g_2 : \langle 23k \to \{t_7.\text{Salary}^{[10k]}, t_8.\text{Salary}^{[25k]}\}, \emptyset \rangle
$$

$$
g_3 : \langle L_1 \to \{t_{10}.\text{SSN}^{\text{new}}, t_{11}.\text{SSN}^{\text{new}}, t_{12}.\text{SSN}^{\text{new}}\}, \text{by} \ \{t_1.\text{SSN}^{[22]}, t_{2}.\text{SSN}^{[22]}, t_{3}.\text{SSN}^{[24]}\} \rangle
$$

$$
g_4 : \langle \text{null} \to \{t_{12}.\text{Salary}^{\text{new}}, t_{13}.\text{Salary}^{\text{new}}\}, \emptyset \rangle
$$

$$
g_5 : \langle \text{Med} \to \{t_{12}.\text{Insur}^{\text{new}}\}, \text{by} \ \{t_2.\text{Insur}_{\text{Med}}\} \rangle
$$

$$
g_6 : \langle \text{Eye surg.} \to \{t_{12}.\text{Treat}^{\text{new}}\}, \text{by} \ \{t_2.\text{Treat}_{\text{Eye surg}}\} \rangle
$$

$$
g_7 : \langle 12/01/2013 \to \{t_{12}.\text{Date}^{\text{new}}\}, \text{by} \ \{t_2.\text{Date}_{[12/01/2013]}\} \rangle
$$

$$
g_8 : \langle \text{Med} \to \{t_{13}.\text{Insur}^{\text{new}}\}, \text{by} \ \{t_3.\text{Insur}_{\text{Med}}\} \rangle
$$

$$
g_9 : \langle \text{Lapar.} \to \{t_{13}.\text{Treat}^{\text{new}}\}, \text{by} \ \{t_3.\text{Treat}_{\text{Lapar}}\} \rangle
$$

$$
g_{10} : \langle 03/11/2013 \to \{t_{13}.\text{Date}^{\text{new}}\}, \text{by} \ \{t_3.\text{Date}_{[03/11/2013]}\} \rangle
$$

Clearly, other updates are possible. For example, to resolve $c_4$ one may consider changing the value of the cell $t_8.\text{INSURANCE}$ into a new llun value $L_3$, i.e., an unknown value that improves ‘Abx’. The following update, $Upd_2$, follows the same approach to satisfy all dependencies:

$$
g_1' : \langle L_2 \to \{t_{5}.\text{SSN}_{[22]}\}, \emptyset, \text{with} \ c_x \rangle
$$

$$
g_2' : \langle L_3 \to \{t_7.\text{Insurance}_{\text{Abx}}\}, \emptyset, \text{with} \ c_x \rangle
$$

$$
g_3' : \langle L_4 \to \{t_8.\text{Insurance}_{\text{Abx}}\}, \emptyset, \text{with} \ c_x \rangle
$$

$$
g_4' : \langle L_5 \to \{t_{10}.\text{Name}^{\text{new}}\}, \text{by} \ \{t_1.\text{Name}_{W. Smith}\}, \text{with} \ c_x \rangle
$$

It yields the solution #2 shown in Figure 2.

We say that two updates coincide if their cell groups are identical, up to the renaming of nulls and lluns. From now on, we blur the distinction between an update $Upd$ and the instance $Upd(J)$ obtained by applying $Upd$ to $J$. Observe that the initial target instance $J$ can be seen as $Upd_0(J)$ where $Upd_0$ denotes the trivial update, i.e., no modifications are made.

10  The Partial Order of Cell Groups and the Notion of Upgrades

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

The Partial Order of Cell Groups Solutions to mapping & cleaning scenarios (formalized in Section 11) are updates made of cell groups that represent “upgrades” to the original target instance. To formalize the notion of an upgrade, we now lift the partial order over cells, $\leq_{\Pi}$, to a partial order over cell groups and updates. This is based on the following intuition: it is natural to say that a cell group $g'$ is an improvement wrt a cell group $g$ if its cells “carry more information” according to the partial order of cells and values. This happens when the set of cells $g'$ contains those of $g$: in this case, we know that the value of $g'$ to be “at least as high” as the one of $g$ in the partial order of cells.

Therefore we state that $g'$ is preferred over $g$ if this containment property among their cells is satisfied. When the containment property is not satisfied, these cell groups represent incomparable ways to modify a target instance, and therefore cannot be ordered. Figure 5 exemplifies some typical cases, to show how the containment property works. Additional examples are given in Example 6 and Figure 6.

**Definition 9 [PARTIAL ORDER FOR CELL GROUPS]** Given two cell groups $g$ and $g'$, we say that $g \preceq_{\Pi,User} g'$ iff $\text{cells}(g) \subseteq \text{cells}(g')$.

This set-theoretic formalization of the partial order of cell groups has several advantages. First, it is straightforward to prove that $\preceq_{\Pi,User}$ is in fact a partial order:

**Proposition 3** Relation $\preceq_{\Pi,User}$ among cell-groups over $(I, J)$ as specified in Definition 9 is a partial order.

In addition, whenever we find two conflicting cell groups $g_1, g_2$ during the repair process, the partial order suggests a straightforward strategy to generalize them by a new cell group $g$ that is the least upper-bound of the original ones: $g$ is obtained by taking the union of their occurrences, justifications, and metacells. This notion will be formalized in Section...
<table>
<thead>
<tr>
<th>$g_1$</th>
<th>$g_2$</th>
<th>$g_1 \preceq_{\text{user}} g_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${N_1 \rightarrow {t_7 A^\text{new}, t_8 A^\text{new}}}$, new cells only</td>
<td>${N_2 \rightarrow {t_7 A^\text{new}, t_8 A^\text{new}}}$, new cells only</td>
<td></td>
</tr>
<tr>
<td>${N_1 \rightarrow {t_7 A^\text{new}, t_8 A^\text{new}}}$, new cells only</td>
<td>${c_1 \rightarrow {t_4 A^\text{new}, t_8 A^\text{new}}}$, constant cell</td>
<td></td>
</tr>
<tr>
<td>${c_3 \rightarrow {t_1 A^\text{new}, t_2 A^\text{new}, t_3 A^\text{new}}}$, constant cell</td>
<td>${L_1 \rightarrow {t_1 A^\text{new}, t_2 A^\text{new}, t_3 A^\text{new}}}$, incomparable constant cells</td>
<td></td>
</tr>
<tr>
<td>${c_4 \rightarrow {t_1 A^\text{new}, t_2 A^\text{new}, t_3 A^\text{new}}}$,</td>
<td>${L_1 \rightarrow {t_1 A^\text{new}, t_2 A^\text{new}, t_3 A^\text{new}}}$, invalid cell</td>
<td></td>
</tr>
<tr>
<td>${c_6 \rightarrow {t_1 A^\text{new}, t_2 A^\text{new}, t_3 A^\text{new}}}$,</td>
<td>${c_6 \rightarrow {t_1 A^\text{new}, t_2 A^\text{new}, t_3 A^\text{new}}}$,</td>
<td></td>
</tr>
<tr>
<td>authoritative cell(s)</td>
<td>authoritative cell(s)</td>
<td></td>
</tr>
<tr>
<td>${c_9 \rightarrow {t_4 A^\text{new}, t_5 A^\text{new}}}$, by ${t_6 B}$, user cell</td>
<td>${c_9 \rightarrow {t_4 A^\text{new}, t_5 A^\text{new}}}$, by ${t_6 B}$, with ${c_7}$, user cell</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Examples of Ordering Relationships between Cell Groups.

14, when we introduce our chase procedure. For now, we notice that the partial order over cell groups can be easily lifted into a partial order over updates, and provides the basis to formalize the crucial notion of an upgrade.

**Example 6:** Consider table $R(A, B)$ with three dependencies:

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

Assume $R$ contains two tuples: $t_1 : R(a, 1), t_2 : R(a, 2)$. Here, $c1, c2$ are authoritative source tables with a single tuple $t_{c1}, t_{c2}$ each, encoding the patterns in the CFDs: $t_{c1} : (A : a, B : v_1), t_{c2} : (A : a, B : v_2)$. Since the two CFDs contradict each other, previous approaches [15] would fail to give a solution to this example. Nevertheless, later on we will provide a semantics for this case.
Following is a set of cell groups, with an indication of their partial order. Here we assume that the partial order specification $\Pi$ states that cells of $A$ with higher values are to be preferred over ones with smaller ones, and $User(cells(g_4)) = k$.

The same cell groups are also described in Figure 6.

$$g_1 = \langle 1 \rightarrow \{t_1.B_{[1]}\}, by \{\emptyset\} \rangle \quad \preceq_{\Pi,User} \quad g_2 = \langle 2 \rightarrow \{t_1.B_{[1]}, t_2.B_{[2]}\}, by \{\emptyset\} \rangle \quad \preceq_{\Pi,User}$$

$$g_3 = \langle v_1 \rightarrow \{t_1.B_{[1]}, t_2.B_{[2]}\}, by \{t_{c_1}, B_{[v_{1}], meta}^{auth}\} \rangle \quad \preceq_{\Pi,User}$$

$$g_4 = \langle L \rightarrow \{t_1.B_{[1]}, t_2.B_{[2]}\}, by \{t_{c_1}, B_{[v_{1}], meta}^{auth}, t_{c_2}, B_{[v_{2}], meta}^{auth}\} \rangle \quad \preceq_{\Pi,User}$$

$$g_5 = \langle k \rightarrow \{t_1.B_{[1]}, t_2.B_{[2]}\}, by \{t_{c_1}, B_{[v_{1}], meta}^{auth}, t_{c_2}, B_{[v_{2}], meta}^{auth}\}, with \{ct\} \rangle \rangle$$

Figure 6: A Graphical Representation of the Cell Groups for Example 6.

In the case of $g_5$, we look for the maximal cell in $occ(g_5) \cup just(g_5) \cup metaCells(g_5)$. Since the user function is defined, the maximal cell consists of the user cell $ct$, and the value of $g_5$ is exactly $User(occ(g_5) \cup just(g_5))$.

**What are llus, in the End?** The role and the importance of llus should now be apparent. While llus are nothing more than symbols from a distinguished set, like constants and nulls, their use in conjunction with cell groups makes them a powerful addition to the semantics. Not only they allow us to complete the lattice of cell-groups and updates, but, when appearing inside cell-groups, they also provide important lineage information to support users in the delicate task of resolving conflicts. Consider again Example 6. The cell group $\langle L \rightarrow \{t_1.B_{[1]}, t_2.B_{[2]}\}, by \{t_{c_1}, B_{[v_{1}], meta}^{auth}, t_{c_2}, B_{[v_{2}], meta}^{auth}\} \rangle$ is a clear indication that it was not possible to fully resolve the conflicts, and therefore user interventions are needed to complete the repair. In addition, the cell-group provides complete information about the conflict, both in terms of which target cells – and therefore which original values – were involved, and also in terms of source values that justify the change.

**Upgrades** We are now ready to formalize the notion of an upgrade. Intuitively, an update $Upd$ is an upgrade of $Upd'$ if the cell groups of $Upd$ are higher in the partial order wrt the ones in $Upd'$. To formalize this notion, we must be able to compare cell groups in the two updates with each other.

In the case of cleaning scenarios, i.e., scenarios with no tgds, the space of cells of the target database is fixed, and we can easily test containment between the cells of a cell group $g \in Upd$ and those of a cell group $g' \in Upd'$.

However, as soon as we have tgds this is not true any longer. Tgds may introduce tuples made of new cells into the target. It is possible that two updates introduce tuples with different ids. In order to be able to compare their cell groups, we need to reduce them to a uniform set of tuple ids. We therefore introduce one final tool, called *id mappings*. This is a way to map updates – which by tgds may add new tuples with completely new ids to the target – to one another.

**Definition 10 [ID MAPPING]** Let $Upd$ and $Upd'$ be two updates over $\langle I, J \rangle$. An id mapping $h_{id}$ from $Upd$ to $Upd'$ maps tuple ids appearing in $Upd(J)$ into those appearing in $Upd'(J)$. We assume that $h_{id}$ also maps each tuple id that appears in both $Upd(J), Upd'(J)$ to itself.

Id mappings can be extended to cells in a straightforward way. Given a cell group $g = \langle occ(g), just(g), metaCells(g) \rangle$ we denote its image according to $h_{id}$ by $h_{id}(g) = \langle h_{id}(occ(g)), h_{id}(just(g)), h_{id}(metaCells(g)) \rangle$. With this in mind, we introduce the concept of an upgrade:

**Definition 11 [UPGRADE]** Given two updates $Upd$ and $Upd'$ over $\langle I, J \rangle$ and a partial order $\preceq_{\Pi,User}$ on cell groups, we say that $Upd'$ upgrades $Upd$, denoted by $Upd \preceq_{\Pi,User} Upd'$, if there exists an id mapping $h_{id}$ from the tuple ids in $Upd$ to the tuple ids in $Upd'$ such that for each cell group $g \in Upd$ there exists a cell group $g' \in Upd'$ and $h_{id}(g) \preceq_{\Pi,User} g'$. 

17
In the rest of the paper, we consider that an update $\text{Upd}'$ is preferable to $\text{Upd}$ whenever $\text{Upd} \preceq_{\Pi, \text{User}} \text{Upd}'$. Notice that $\preceq_{\Pi, \text{User}}$ is a preorder for updates, and not a partial order. It is not, in fact, antisymmetric: there may exist updates $\text{Upd}$ and $\text{Upd}'$ such that $\text{Upd} \preceq_{\Pi, \text{User}} \text{Upd}'$, $\text{Upd}' \preceq_{\Pi, \text{User}} \text{Upd}$, and $\text{Upd} \neq \text{Upd}'$ (examples are in the next section). It is possible to show that, in the case of cleaning scenarios where no tgd is present, $\preceq_{\Pi, \text{User}}$ is also a partial order.

11 Semantics

In this section, we formalize the semantics of a mapping and cleaning scenario. The key challenge here is to develop a new semantics for mappings and cleaning constraints that is an extension of the semantics for mappings in [14], and of those of data repairing in [15]. We address this challenge by leveraging the notions of cell groups, partial order and upgrades.

Solutions and Satisfaction after Upgrades

Intuitively, a solution of a mapping and cleaning scenario is a set of repair upgrades.

Example 7: To see this, consider first this simple example, in which we have a source table $S(A, B)$, and a target table $T(A, B)$, with a s-t tgd and an egd:

$$\begin{align*}
m & : S(x, y) \rightarrow T(x, y) \\
e & : T(x, y), T(x, y') \rightarrow y = y'
\end{align*}$$

Suppose we are given two source tuples, $t_{s_1} : R(a, b), t_{s_2} : R(a, c)$. To satisfy the tgd, we need to insert two new tuples, $t_1 : T(a, b), t_2 : T(a, c)$ into the target. These, however, violate egd $e$. Therefore, we update the database using the following update $\text{Upd}$ that equates the two $B$-values by using a llun, $L$:

$$\begin{align*}
g_1 & : (a \rightarrow \{t_{s_1}, A^{new}\}, \{t_{s_1}, A_{[a]}\}) \\
g_2 & : (a \rightarrow \{t_{s_2}, A^{new}\}, \{t_{s_2}, A_{[a]}\}) \\
g_3 & : (L \rightarrow \{t_{1}, B^{new}, t_{2}, B^{new}\}, \{t_{s_1}, B_{[b]}, t_{s_2}, B_{[c]}\})
\end{align*}$$

Notice that $\text{Upd}(J)$ satisfies the egd, but it does not satisfy tgd $m$ in the standard sense. However, we still want to consider $\text{Upd}$ as a solution, since it is the result of an “improvement” of values that originally satisfied $m$, but were dirty according to $e$. Notice also that trying to satisfy $m$ in the standard sense would yield an infinite loop (we would need to insert back into the target tuples $T(a, b), T(a, c)$, then equate the $B$-values, and so on).

Example 8: As a second example, assume now we have a target table $T(A, B)$ and two source tables, $S_1(A, B), S_2(A, B)$. Assume also $S_2$ is authoritative, while $S_1$ is not. We are given the following egds:

$$\begin{align*}
e_1 & : S_1(x, y), T(x, z) \rightarrow z = y \\
e_2 & : S_2(x, y), T(x, z) \rightarrow z = y
\end{align*}$$

We start with a target instance made of a single tuple $t_1 : T(a, b)$, and two source tuples $t_{s_1} : S_1(a, c), t_{s_2} : S_2(a, d)$. It can be seen that the egds are contradicting each other, since the $e_1$ requires that the $B$-value of $t_1$ is equal to $c$, while $e_2$ requires that it is equal to $d$. However, we know that $S_2$ is authoritative, and therefore $d$ “wins” over other values. As a consequence, we want to repair the target by the following update $\text{Upd}$:

$$g : (d \rightarrow \{t_{1}, B_{[d]}, t_{s_1}, B_{[c]}, t_{s_2}, B^{auth}_{[d]}\})$$

Again, $\text{Upd}$ is a natural solution in our setting, since it clearly improves the quality of the target. We accept the fact that it does not satisfy egd $e_1$ in the standard sense, since satisfying the egd would result in a “decrease” of quality of the target. A similar issue arises also in Example 6 above, where we have two contradicting CFDs that force us to introduce a llun.

In essence, any dependency – egd or tgd – that has source symbols in its premise, relates target values to values in the source database. Later on during the repair process, we may need to change these target values to fix further violations. However, we are not allowed to change the source values they are related to, since the source database is immutable. To handle this, we shall adopt a revised notion of satisfaction for dependencies, called satisfaction after upgrades. In fact, as it will be clear at the end of this section, for dependencies that only contain target symbols, the notion of satisfaction after repairs coincides with the standard notion of satisfaction for egds and tgds.

Based on these ideas, we can now give the formal definition of a solution. The notion of satisfaction after upgrades is formalized in Section 12.

Definition 12 [Solution] Given a mapping&cleaning scenario $\mathcal{MC} = \{S, S_0, T, \Sigma_t, \Sigma_e, \Pi, \text{User}\}$, and an input instance $(I, J)$, a solution for $\mathcal{MC}$ over $(I, J)$ is an update $\text{Upd}$ s.t.:
1. $J \subseteq_{\text{Update}} \text{User} \cup \text{Upd}$, i.e., Upd upgrades the initial target instance;

2. $(I, \text{Upd}(J))$ satisfies after upgrades $\Sigma_t \cup \Sigma_u$ under $\subseteq_{\text{Update}}$.

Minimal Solutions We are interested in solutions that are minimal, i.e., they do not contain unneeded target tuples and upgrade the initial target instance as little as possible. To quantify minimality, we leverage $\subseteq_{\text{Update}}$ to decide when one update strictly upgrades another.

**Definition 13 [Strict Upgrade]** We say that update $\text{Update}'$ strictly upgrades update $\text{Upd}$, denoted by $\text{Upd} \preceq_{\text{User}} \text{Update}'$ if:

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

![Figure 7: A diagram of solutions for Example 2.](image)

**Definition 14 [Minimal Solutions]** A minimal solution for a mapping and cleaning scenario is a solution $\text{Upd}$ such that there exists no solution $\text{Upd}'$ such that $\text{Upd} \preceq_{\text{User}} \text{Upd}'$.

Consider Example 2. Figure 7 reports some of the updates and solutions for this example, along with a diagram of the “upgrades” relationship among them. This example clearly shows that $\subseteq_{\text{Update}}$ is a preorder for updates, and not a partial order (see, for example, $\text{Upd}_3, \text{Upd}_7$).
Notice also the top solution, i.e., the update that changes all cells of \( J \), and justifies it by all cells in \( I \), with the invalid cell, i.e., \( \text{Upd}_{\text{top}} = \langle \text{cells}(J), \text{cells}(I), \{c_x\} \rangle \).

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

## 12 Satisfaction After Upgrades

To complete the formalization of the semantics, it remains to define the notion of satisfaction after upgrades, first for egds and then for tgd.

Let \( \text{Upd} \) be an update over \( \langle I, J \rangle \). Clearly, if \( \langle I, \text{Upd}(J) \rangle \) satisfies an edg or tgd in the standard semantics, nothing needs to be done. Otherwise, we need to revise the semantics for egds and tgd in such a way that they remain satisfied as long as we upgrade the database.

### Satisfaction After Upgrades for Egds

Consider our Example 8 above, and our solution \( \text{Upd} \). We notice that there exists an homomorphism \( h \) of the premise of \( e_1 \) into \( \langle I, \text{Upd}(J) \rangle \) such that \( h(y) \neq h(z) \), and therefore \( e_1 \) is not satisfied in the standard sense. Intuitively, we consider that \( \text{Upd} \) satisfies \( e_1 \) under \( h \) since the cell group in \( \text{Upd} \) associated with \( z \) is “better” than the one associated with \( y \). To formalize this intuition, given an egds, variables \( x, x' \), and a homomorphism \( h \) of the premise into \( \langle I, \text{Upd}(J) \rangle \), we need to be able to compare the cell groups associated by \( h \) with \( x, x' \), to check whether one value, say \( h(x) \), is an upgrade for \( h(x') \), or vice versa.

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

#### Definition 15 (Cell Group for Variable)

Given a dependency \( d : \phi(x, z) \rightarrow \exists y : \psi(x, y) \), and a homomorphism \( h \) of \( \phi(x) \) into \( \langle I, \text{Upd}(J) \rangle \), for each variable \( x \in \bar{x} \) we define the cell group associated by \( h \) with \( x \) as the cell group \( g_h(x) = \langle \text{occ}, \text{just}, \text{metaCells} \rangle \), where:

- \( \text{occ} \) (resp. \( \text{just} \), resp. \( \text{metaCells} \)) is the union of all occurrences (resp. justifications, resp. metacells) of the cell groups in \( \text{Upd} \) for cells in \( \text{cells}_h(x) \);

- in addition, \( \text{just} \) contains all cells in \( \text{cells}_h(x) \) that belong to the source tables in \( I \).

Consider the sample scenario in example 2. Given the initial instance, the s-t tgd \( m : S(x, y) \rightarrow \exists z : T(x, y, z) \), and the homomorphism \( h \) that maps \( x \) into constant 1 and \( y \) into 2, the cell groups for these variables are \( g_h(x) = \langle 1 \rightarrow \emptyset, \text{by} \{t_1, A[I]_1\} \rangle \), and \( g_h(y) = \langle 2 \rightarrow \emptyset, \text{by} \{t_2, B[I]_1\} \rangle \). Consider now the target instance \( \Delta J \) and the egd \( e : T(x, y, z), T(x, y', z') \rightarrow y = y' \). In this case, consider homomorphism \( h \) that maps, among others, variable \( x \) into constant 1; the cell group associated by \( h \) with \( x \) is \( g_h(x) = \langle 1 \rightarrow \{t_3, A^\text{new}, t_4, A^\text{new}\}, \text{by} \{t_1, A[I]_1|t_2, A[I]_1\} \rangle \).

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

#### Definition 16 (Satisfaction After Upgrades - Egds)

Given an egd \( e : \forall \exists \phi(x) \rightarrow x = x' \), an instance \( \langle I, J \rangle \), and an update \( \text{Upd} \), we say that \( \langle I, \text{Upd}(J) \rangle \) satisfies after upgrades \( e \) wrt the partial order \( \preceq_{\text{User}} \) if, whenever there is a homomorphism \( h \) of \( \phi(x) \) into \( \langle I, \text{Upd}(J) \rangle \), then \( i \) either the value of \( h(x) \) and \( h(x') \) are equal (standard semantics), or \( ii \) it is the case that the cell groups associated by \( h \) with \( x, x' \) are comparable wrt \( \preceq_{\text{User}} \), i.e., either \( g_h(x) \preceq_{\text{User}} g_h(x') \) or \( g_h(x') \preceq_{\text{User}} g_h(x) \).

Let us now reconsider Example 8. Consider egd \( e_1 : e_1 : S_1(x, y), T(x, z) \rightarrow z = y \), and update \( \text{Upd} \) discussed above: \( g : \langle d \rightarrow \{t_1, B[I]_1\}, \text{by} \{t_{s_1}, B[c], t_{s_2}, B[d] \} \rangle \). Given \( h \) that maps \( x \) to \( t_{s_1}, A[I]_1 \), \( t_1, A[I]_1 \); \( y \) to \( t_{s_1}, B[c] \), and \( z \) to \( t_1, B \) whose value is \( d \) in \( \text{Upd}(J) \). We know that \( e_1 \) is not satisfied in the standard sense, since \( h(y) = c \neq h(z) = d \). Let us consider the cell groups that \( h \) associates with \( y, z, \) as follows:

\[
g_h(y) : \langle c \rightarrow \emptyset, \text{by} \{t_{s_1}, B[c]\} \rangle \quad g_h(z) : \langle d \rightarrow \{t_1, B[I]_1\}, \text{by} \{t_{s_1}, B[c], t_{s_2}, B[d] \} \rangle
\]

Notice that \( g_h(y) \) has no occurrences – it only holds the justification coming from \( t_{s_1} \), while \( g_h(z) \) coincides with \( g \), the cell group for cell \( t_1.B \). It is easy to see that \( g_h(y) \not\preceq_{\text{User}} g_h(z) = g \), and therefore \( e \) is satisfied after upgrades.

### Satisfaction After Upgrades for Tgds

In the case of tgd, the intuition is that we want to regard a tgd \( m \) as being satisfied after upgrades by an update \( \text{Upd} \) whenever \( \text{Upd}(J) \) is an upgrade of the canonical update for \( m \) and \( h \). Intuitively, the canonical update represents the “standard way” to satisfy the tgd, defined as follows:
Definition 17 [Canonical Update] Given an update, Upd, consider a tgd $m : \forall \exists \phi(\pi) \rightarrow \exists \psi(\pi, \eta))$ that is not satisfied by $\{I, \text{Upd}(J)\}$. Let $h$ be a homomorphism of $\phi(\pi, \eta)$ into $\{I, \text{Upd}(J)\}$ that cannot be extended to a homomorphism $h'$ of $\psi(\pi, \eta)$ into $\{I, \text{Upd}(J)\}$.

Let $h_{\text{can}}$ be the canonical homomorphism that extends $h$ by injectively assigning a fresh labeled null with each existential variable. Consider the updated instance $h_{\text{can}}$ of $\psi(\pi, \eta)$, obtained by adding to $\text{Upd}(J)$ the set of tuples in $h_{\text{can}}(\psi(\pi, \eta))$, each with a fresh tuple id. Then, the canonical update associated with $\text{Upd}$, $m$ and $h$, denoted by $\text{Upd}_{m}^{h_{\text{can}}}$, is such that:

1. $\text{Upd}_{m}^{h_{\text{can}}}(J) = \text{Upd}(J) \cup h_{\text{can}}(\psi(\pi, \eta))$;
2. $\text{Upd}_{m}^{h_{\text{can}}}$ coincides with $\text{Upd}$ when restricted to cells$(\text{Upd}(J))$;
3. it contains a cell group $g_{h_{\text{can}}}(x)$ over $\{I, J\}$ for each variable $x \in \pi$, where $g_{h_{\text{can}}}(x)$ denotes the cell group associated by $h_{\text{can}}$ with variable $x$;
4. it contains a cell group $g_{h_{\text{can}}}(y)$ over $\{I, J\}$ for each variable $y \in \eta$, with value $h_{\text{can}}(y)$, occurrences equal to cells$h_{\text{can}}(y)$, all new cells, empty justifications and empty metacells. $\square$

Definition 18 [Satisfaction After Upgrades - Tgds] Given a tgd $m : \forall \exists \phi(\pi) \rightarrow \exists \psi(\pi, \eta))$, and an update $\text{Upd}$, we say that $\{I, \text{Upd}(J)\}$ satisfies after upgrades $m$ under partial order $\leq_{\text{User}}$ if, whenever there is a homomorphism $h$ of $\phi(\pi)$ into $\{I, \text{Upd}(J)\}$, then (i) either $m$ is satisfied by $\{I, \text{Upd}(J)\}$ in the standard sense, or (ii) $\text{Upd}_{m}^{h_{\text{can}}} \leq_{\text{User}} \text{Upd}$.

Consider again Example 7, tgd $m : S(x, y) \rightarrow T(x, y)$, and homomorphism $h$ that maps $S(x, y)$ to $t_{s_{1}} : S(a, b)$. The canonical update $\text{Upd}_{m}^{h_{\text{can}}}$ in this case is as follows:

$$g_{1}^{\text{can}} : \{a \rightarrow \{t_{1}, A^{\text{new}}\}, \text{by } \{t_{s_{1}}, A_{b[a]}\}\}$$

$$g_{2}^{\text{can}} : \{b \rightarrow \{t_{1}, B^{\text{new}}\}, \text{by } \{t_{s_{1}}, B_{b[b]}\}\}$$

Consider the update $\text{Upd}$ in Example 7 we obtain after satisfying the egds. Recall that it contains a cell group $g_{3} : \{L \rightarrow \{t_{1}, B^{\text{new}}, t_{2}, B^{\text{new}}\}, \text{by } \{t_{s_{1}}, B_{b[b]}, t_{s_{2}}, B_{b[b]}\}\}$. It can be seen that $\text{Upd}_{m}^{h_{\text{can}}} \leq_{\text{User}} \text{Upd}$. Similarly for the other homomorphism $h'$ that maps $S(x, y)$ to $t_{s_{2}} : S(a, c)$. Therefore, $\text{Upd}$ satisfies $m$ after upgrades.

This concludes the formalization of the notion of a solution as given in Definition 12. As we mentioned, the notion of satisfaction after repairs is only needed for dependencies in which source symbols appear, as stated by the following lemma:

Lemma 4 Any cleaning egd or extended tgd in which only target symbol appear is satisfied after repairs if and only if it is satisfied in the standard sense.

13 Restrictions and Relationship to Other Semantics

We stated several times that our semantics generalizes the one previously introduced for mapping scenarios [14], and incorporates many of the features of the semantics proposed for data repairing [9, 27, 8]. Our goal in this section is to introduce two restrictions of our setting, the notion of a mapping scenario, and the notion of a cleaning scenario. We will investigate the relationship between mapping & cleaning scenarios and existing repair strategies later on, in Section 16, after we have introduced our operational chase-based semantics.

Mapping Scenarios and Data Exchange To start, we compare our semantics to the one of data exchange. Recall that traditionally [14] a data-exchange scenario has been defined as a quadruple $\mathcal{M}_{de} = \{S, T, \Sigma_{de}, \Sigma_{de}^{e}\}$, where $S$ and $T$ are the source and target schemas, $\Sigma_{de}$ a set of self tgds, and $\Sigma_{de}^{e}$ a set of target constraints that includes target tgds and target egds. Note how we use superscripts to emphasize that $\mathcal{M}_{de}$ only contains standard dependencies, as defined in Section 3.

We see mapping scenarios as a restriction of our framework. Recall the general definition of a mapping & cleaning scenario as a tuple $\mathcal{M} = \{S, S_{a}, T, \Sigma_{t}, \Sigma_{c}, \Pi, \text{User}\}$, where dependencies are the extended tgds and cleaning egds defined in Section 4. Given a data-exchange scenario $\mathcal{M}_{de}$, we now introduce its associated mapping scenario $\mathcal{M}_{de}^{\text{map}}$ as a mapping & cleaning scenario over the same source and target schema, with the following restrictions:

- $S_{a}$, the authoritative schema is empty;
- $\Sigma_{t}$ is the set of standard s-t tgds in $\Sigma_{de}$ and the set of standard target tgds in $\Sigma_{de}^{e}$.
• $\Sigma_e$ is the set of standard egds in $\Sigma^{de}$;

• $\Pi$ and User are empty;

• finally, the set of lluns, $\text{LLUNS}$, is also empty, i.e., we only allow for constants and labeled nulls.

When restricted to this notion of a mapping scenario, some of the notions in our semantics are simplified. First, a cell group becomes simply a set of occurrences and justifications (no metacells are used). Second, the partial order over the cells of $\langle I, J \rangle$ only states that constants are preferable to labeled nulls.

The relationship of our semantics to the one of data exchange is stated by the following theorem.

**Theorem 5:** Every core solution $J_{\text{core}}$ of a data exchange scenario corresponds to a minimal solution $\text{Upd}_{\text{min}}$ of its associated mapping scenario. Every minimal solution $\text{Upd}_{\text{min}}$ corresponds to a solution $J_{\text{min}}$ of the data exchange scenario. If $J_{\text{min}}$ is universal, then it is also a core solution for the data exchange scenario.

**Remark** Previously publisher papers [21] have reported a more general result, stating that mapping scenarios capture any solution to the data exchange setting, and every minimal solution to the mapping scenario corresponds to a core solution. This is not true in this paper. In fact, as it will shown in Lemma 15, within this framework mapping scenarios only capture domain-based solutions to the corresponding data-exchange settings, i.e., solutions in which only constants from the active domain of $I$ may appear, and minimal solutions do not necessarily correspond to universal solutions. It remains true, however, that any solution that is generated by the chase is universal.

These two results are not in contradiction with each other. In fact, the previous theorem [21] used a different formalization of cell groups, based on non-strict cell groups. A non-strict cell group is a cell group that may either have a value that is the least-upper bound value of its cells, as in this paper, or a generalization thereof. Here, on the contrary, we concentrate on strict cell groups. This prevents us from capturing non-universal solutions in which a constant outside of the active domain of $I$ is arbitrarily written into a cell of the target. The relationship between solutions and cell groups is depicted in Figure 8.

Despite this, in this paper we prefer strict cell groups over the more general but significantly more involved non-strict ones, since the former strict cell groups can be formalized in a more elegant and readable way. A complete treatment of the framework based on non-strict cell groups can be found in [34].

**Cleaning Scenarios** Let us turn our attention to scenarios with no tgds. We call a cleaning scenario a mapping and cleaning scenario in which the set of tgds $\Sigma_t$ is empty. We discuss them in detail in Section 16. For now, let us summarize some of the properties of cleaning scenarios. First, we notice that, even in the absence of tgds, cleaning egds may still maintain a link between the source and the target database, e.g., to leverage authoritative sources. Second, cleaning scenarios always admit solutions:

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

Indeed, there is always a solution corresponding to the top update that changes all cells of $J$ together, and justifies it by all cells in $I$, using both metacells, i.e., $\text{Upd}_{\text{top}} = \langle \text{cells}(J), \text{cells}(I), \{c_x, c_T\} \rangle$. We make the straightforward assumption that this solution is never refused by the user function.
Theorem 7: Given two solutions \( \text{Upd}, \text{Upd}' \) for a scenario \( \text{CS} \) over instance \( \langle I, J \rangle \), one can check if \( \text{Upd} \preceq_{\Pi} \text{Upd}' \) in \( O(n + k n \log(m)) \) time, where \( n \) is the number of cells in \( J \), \( k \) is the maximum number of cell groups in \( \text{Upd}, \text{Upd}' \), and \( m \) is the maximum size of a cell group in \( \text{Upd}, \text{Upd}' \).

14 The Chase

In order to generate solutions for mapping & cleaning scenarios, we revise and extend the traditional chase procedure for egds and tgds [14] with the following goals:

- first, we want that the chase correctly computes solutions for our semantics; a key property, in this respect, is that chase steps preserve cell groups and generate actual upgrades of the target database; as a consequence, our chase works with cell groups and the partial order;
- second, we consider important that the chase explores all possible strategies to satisfy the dependencies. This implies that an egd may be chased both forward, to satisfy its conclusion, and backward, to falsify its premise; this, in turn, means that we need to consider a disjunctive chase, which generates a tree of alternative solutions;
- finally, we want that the chase provides a basis to implement an engine that scales nicely to large databases. Given this requirement and the expressibility one above, we need to find the appropriate balance between generality and scalability.

To simplify the presentation and ease the reading, we shall proceed in two steps. In this section, we introduce a first, simplified version of the chase that we show to be correct. Then, we revise the definition in the next section to provide a basis for the introduction of scalability techniques.

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

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

**Chase Step for Tgds** The definition of a chase step for an extended tgd is a natural generalization of the standard notion of a chase step for a standard tgd [14], the main difference being the need to properly update cell groups.

**Definition 19 [Chase Step for TGDs]** Given an extended tgd \( m : \forall \pi (\phi(\bar{x}, \bar{z}) \Rightarrow \exists y : (\psi(\bar{x}, \bar{y})) \in \Sigma_t \) applicable to \( \langle I, \text{Upd}(J) \rangle \) with homomorphism \( h \), by chasing \( m \) on \( \langle I, \text{Upd}(J) \rangle \) with \( h \) we generate a new update \( \text{Upd}' \) obtained from \( \text{Upd} \) by:

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

(ii) adding the new cell groups in the canonical update \( \text{Upd}_{\text{can}} \).

In symbols we write \( \text{Upd} \rightarrow_{m,h} \text{Upd}' \), where \( \text{Upd}' = \text{Upd}_{\text{can}} \).

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

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

**Chase Step for Egds** In order to introduce the notion of a chase step for egds we need a few additional definitions. We first introduce the notions of witness and witness variable for an egd. Intuitively, the witness variables are those variables upon which the satisfiability of the dependency premise depends; these are all variables that have more than one occurrence in the premise, i.e., they are involved in a join or in a selection.
Definition 20 [Witness] Let \( e : \forall x (\phi(x) \rightarrow x = x') \). A witness variable for \( e \) is a variable \( x \in \bar{x} \) that has multiple occurrences in \( \phi(\bar{x}) \). For a homomorphism \( h \) of \( \phi(\bar{x}) \) into \( \langle I, \text{Upd}(J) \rangle \), we call a witness, \( w_h \) for \( e \) and \( h \), the vector of values \( h(\bar{x}_w) \) for the witness variables \( \bar{x}_w \) of \( e \).

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

In the following, we denote by \( \text{Union}(g, g') \) the union of two cell groups \( g, g' \), i.e., the new cell group obtained by taking the union of occurrences, justifications, and metacells of the original cell groups. As usual, the value of \( \text{Union}(g, g') \) is defined according to Definition 7.

Definition 21 [Chase Step for EGDS] Given a cleaning edge \( e : \forall x (\phi(x) \rightarrow x = x') \) in \( \Sigma_e \) applicable to \( \langle I, \text{Upd}(J) \rangle \) with homomorphism \( h \), by forward chasing \( e \) on \( \langle I, \text{Upd}(J) \rangle \) with \( h \) we generate a new update \( \text{Upd}_f \) obtained from \( \text{Upd} \) by:

1. removing \( g_h(x) \) and \( g_h(x') \);
2. adding the new cell group \( g_f = \text{Union}(g_h(x), g_h(x')) \).

In symbols \( \text{Upd}_f = \text{Upd} - \{ g_h(x), g_h(x') \} \cup g_f \).

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

1. \( \text{val}(g_{ij}) \in \text{CONSTS} \), i.e., the cell has a constant value, and
2. \( \text{auth-cells}(g_{ij}) = \emptyset \), i.e., \( g_{ij} \) has no authoritative justifications,

then, we generate a new update \( \text{Upd}_{bij} \) obtained from \( \text{Upd} \) by changing \( g_{ij} \) to another cell group \( g'_{ij} \) that has same occurrences and justifications, and the invalid cell. In symbols \( \text{Upd}_{bij} = \text{Upd} - \{ g_j \} \cup \{ g'_j \} \).

We consider all these chase steps in parallel, and write \( \text{Upd} \rightarrow_{c,h} \text{Upd}_f, \text{Upd}_{d0}, \text{Upd}_{d1}, \ldots, \text{Upd}_{dn} \), where \( \text{Upd}_f \) and \( \text{Upd}_{d0}, \text{Upd}_{d1}, \ldots, \text{Upd}_{dn} \) are the updates generated by the forward and backward chase step, respectively.

We do not backward-chase cells in two cases: (i) they contain nulls or lluns, which are essentially placeholders; in fact, replacing a placeholder by another one does not represent an upgrade; (ii) they have a authoritative justification from the source; in fact, we consider it unacceptable to disrupt an authoritative value coming from the source in favor of a llun.

To see an example, tuples \( t_7 \) and \( t_8 \) in the pre-solution in Figure 2 violate edge \( e_9 \), which states that \( \text{SSN} \) implies \( \text{Salary} \) on the table Treatments. Here the witness variable is \( \text{ssn} \), and the \( \text{cells}_h(\text{ssn}) \) are \( t_7.\text{SSN} \) and \( t_8.\text{SSN} \), with cell groups \( g_{\text{Upd}}(t_7.\text{SSN}) : \{111 \rightarrow \{ t_7.\text{SSN}_{[111]} \}, \text{by 0} \} \) and \( g_{\text{Upd}}(t_8.\text{SSN}) : \{111 \rightarrow \{ t_8.\text{SSN}_{[111]} \}, \text{by 0} \} \). Moreover, the cell groups for variables \( x \) and \( x' \) are \( g_h(x) : \{10K \rightarrow \{ t_7.\text{Salary}_{[10K]} \}, \text{by 0} \} \) and \( g_h(x') : \{25K \rightarrow \{ t_8.\text{Salary}_{[25K]} \}, \text{by 0} \} \). Notice that the two cell groups are incomparable, and therefore we have a violation.

The chase procedure generates three different updates for the violation:

(a) \( \text{Upd}_f \) in the place of \( g_h(x) \) and \( g_h(x') \) contains the new cell group \( g_f : \{25K \rightarrow \{ t_7.\text{Salary}_{[10K]} , t_8.\text{Salary}_{[25K]} \}, \text{by 0} \} \)

(b) \( \text{Upd}_{bij} \) that in the place of \( g_{\text{Upd}}(t_7.\text{SSN}) \) contains its backward cell group \( g_{ij} : \{ L_6 \rightarrow \{ t_7.\text{SSN}_{[111]} \}, \text{by 0, with } \{ c_s \} \} \);

(c) \( \text{Upd}_{bij} \) that in the place of \( g_{\text{Upd}}(t_8.\text{SSN}) \) contains its backward cell group \( g_{ij} : \{ L_7 \rightarrow \{ t_8.\text{SSN}_{[111]} \}, \text{by 0, with } \{ c_s \} \} \).

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

Definition 22 [Chase Step for User Inputs] Given an update \( \text{Upd} \) of \( J \), we say that User applies to a group \( g \in \text{Upd} \) if User(occ\( g \) \cup just\( g \)) is defined, and \( g \) does not contain the user cell.

We say that User refuses \( \text{Upd} \) if User(cells\( g \)) = \( \perp \). If User refuses \( \text{Upd} \), we mark \( \text{Upd} \) as invalid. Otherwise, we denote by User(\( \text{Upd} \)) the update obtained from \( \text{Upd} \) by replacing any cell group \( g \in \text{Upd} \) such that User applies to \( g \), by a new cell group \( g_{\text{User}} \) obtained from \( g \) by adding the user cell, \( c_T \), to its metacells.

Chasing \( \langle I, \text{Upd}(J) \rangle \) with User either marks \( \langle I, \text{Upd}(J) \rangle \) as invalid, or generates a new update \( \text{Upd}' \) in which cell groups have been changed according to User. In the latter case, we write \( \text{Upd} \rightarrow_U \text{Upd}' \).
Consider again the Upd in Example 5. If User(\{t_1.SSN, t_2.SSN, t_3.SSN\}) = 123, chasing \(\{I, Upd_i(J)\}\) with User generates a new update that contains, in the place of \(g_3\), the new cell group \(g_{User} : \langle I23 \rightarrow \{t_{10}.SSN^{new}, t_{11}.SSN^{new}, t_{12}.SSN^{new}, t_{13}.SSN^{new}\}\}\ by \(\{t_{1}.SSN[I23], t_{2}.SSN[I23], t_{3}.SSN[I23]\}\) with \(\{e_T\}\)

**Chase Tree** Given sets of tgd and egds \(\Sigma_t, \Sigma_e\), a chase of \(\{I, J\}\) with \(\Sigma_t, \Sigma_e\) and User, denoted by \(\text{chase}_{\Sigma_t, \Sigma_e, \text{User}}(\{I, J\})\) is a tree whose root is \(\langle I, J\rangle\), i.e., the empty update, and for each valid node Upd, the children of Upd are the updates Upd_0, Upd_1, ..., Upd_n such that, for some \(d \in \Sigma_t, \Sigma_e\) and some \(h\), it is the case that Upd \(\rightarrow_{de} Upd_0, Upd_1,..., Upd_n\), or Upd \(\rightarrow_{User} Upd_0\). The leaves are updates Upd_i such that there is no dependency applicable to \(\{I, Upd_i(J)\}\) with some homomorphism \(h\). Any leaf in the chase tree is called a result of the chase of \(\{I, J\}\) with \(\Sigma_t, \Sigma_e\).

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

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

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

We can prove that, as soon as the tgd are non-recursive, then the chase terminates. This result is far from trivial, since, as we discussed, egds interact quite heavily with tgd by updating values in the database. We conjecture that this result can be extended to more sophisticated termination conditions for tgd [22].

**Theorem 9:** Given a mapping & cleaning \(\mathcal{MC} = \{S, S_a, T, \Sigma_t, \Sigma_e, \Pi, \text{User}\}\) instances \(I \cap S \cup S_a\) and \(J \cap T\), and oracle User, if \(\Sigma_t\) is a set of weakly-acyclic tgd, then the chase of \(\{I, J\}\) with \(\Sigma_t, \Sigma_e\), User terminates after a finite number of steps, and each leaf in the chase tree is a solution for \(\mathcal{MC}\).

In the case of mapping scenarios, the chase always generates universal solutions:

**Theorem 10:** Given a data exchange scenario \(\mathcal{M}_{de}\), the corresponding cleaning scenario \(\mathcal{M}_{de}^{map}\) is such that any solution generated by the chase of \(\{I, J\}\) is a universal solution to \(\mathcal{M}_{de}\) and \(\{I, J\}\).

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

**Theorem 11:** Given a cleaning scenario \(\mathcal{CS} = \{S, S_a, T, \emptyset, \Sigma_e, \Pi, \text{User}\}\) and an instance \(\{I, J\}\), the chase of \(\{I, J\}\) with \(\Sigma_e\) (i) terminates; (ii) it generates a finite set of results, each of which is a solution for \(\mathcal{CS}\) over \(\{I, J\}\).

It is well-known [9] that a database can have an exponential number of solutions, even for a cleaning scenario with a single FD and when no backward chase steps are allowed.

**Theorem 12:** Given a cleaning scenario \(\mathcal{CS} = \{S, S_a, T, \emptyset, \Sigma_e, \Pi, \text{User}\}\) and an instance \(\{I, J\}\), \(\mathcal{CS}\) may have at most an exponential number of solutions over \(\{I, J\}\), and each solution is computed in a number of steps that is polynomial in the size of \(\{I, J\}\).

We discuss techniques to handle this high complexity in the next section.

### 15 A Revised Chase

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

**Example 9:** To explain scalability issues, consider a simple functional dependency \(A \rightarrow B\) over relation \(R(A, B, C)\), expressed as the following egd: \(e. R(a, b, c), R(a, b', c') \rightarrow b = b'\). Assume we are given tuples \(t_1 = R(1, 2, x)\), \(t_2 = R(1, 2, y)\), \(t_3 = R(1, 4, z)\), \(t_4 = R(2, 5, w)\), \(t_5 = R(2, 6, v)\). In our definition of a chase step so far, violations to dependencies are analyzed in a tuple-oriented fashion, i.e., by considering two tuples at a time. However, this is highly inefficient, in some cases. Assume we want to enforce the FD by forward changes. It can be seen that eventually the \(B\)-value of \(t_1, t_2, t_3\) will all become equal. If we analyze violations in a tuple-oriented fashion, it will take several chase steps to realize this. Our goal, on the contrary, is to group and fix all violations together. In the literature [9, 15] a similar notion has been used under the name of equivalence classes.

We now rework our definition of a chase step in such a way that:
• it considers groups of tuples that may be repaired together within a single step, to reduce the number of steps needed to reach a solution and speed-up the chase;
• it generates the same solutions that the chase in Section 14 would do, i.e., it is correct and equally expressive;
• at the same time, it allows scenario designers to plug optimization strategies into the chase, to specify which branches of the chase tree should be explored and which ones should be ignored, typically for performance reasons, in order to prune the tree.

To achieve this goal, our approach is the following: (a) first, for each violation we identify all of the tuples involved by considering equivalence classes made more than one homomorphism at a time; (ii) then, we enumerate all of the possible ways of solving the violation by means of forward and backward changes; each one is called a repair strategy for the violation; (iii) finally, we introduce the notion of a cost manager as a predicate that accepts or refuses repair strategies. In this way, by adopting the standard cost manager – the one that accepts all possible repair strategies – we have the same power of the chase procedure in Section 14. On the contrary, by adopting more aggressive cost managers we may effectively prune the chase tree and improve the scalability of the chase engine.

For ease of readability, these notions are introduced informally in this section. Their formal treatment is reported in Appendix A.

The Chase of Egds Reworked Recall the definition of a witness and witness variable given in Section 14. In our example above, the witness variable for egd e is a, and we have several homomorphisms, some with witness 1 and some others with witness 2. We design the chase so that each chase step does not consider a single homomorphism at a time, but rather groups of homomorphisms.

To do this, we introduce the notion of an homomorphism class as a set of homomorphisms of a formula into an instance that have equal witness values. Notice that homomorphism classes induce classes of tuples in a natural way. In our example, the tuples are partitioned into two classes, as follows: \( ec_1 = \{t_1, t_2, t_3\} \) (with witness 1) and \( ec_2 = \{t_4, t_5\} \) (with witness 2). Consider the homomorphism class \( ec_1 \) (witness 1), composed of the three tuples \( \{t_1, t_2, t_3\} \): we identify a violation since these tuples have two different values for the \( B \)-attribute, 2 and 4, respectively. Our intuition is that the chase can be made more efficient by repairing the cells of all of these tuples at once.

The actual definition of the chase is made more complex by the fact that our updates are made of cell groups. As a consequence, for each tuple to repair we need to identify a cell group. We call the conclusion groups \( css \), composed of the three tuples \( \{t_1, B_{[2]}\} \), by \( g_1 : (z \rightarrow \{t_1, B_{[2]}\}, by 0) \) and \( g_2 : (z \rightarrow \{t_2, B_{[2]}\}, by 0) \) and \( g_3 : (z \rightarrow \{t_3, B_{[4]}\}, by 0) \).

The first very important notion of this reworked chase procedure is the one of chase-step strategy. In essence, a chase-step strategy tells which of the conclusion groups for an homomorphism class \( H \) should be forward chased, which ones should be backward chased (by changing cells matched to premise variables), and which ones should be left untouched. Therefore, we see a chase-step strategy as a mapping from the set of conclusion groups for \( H \), into the set \( \{f, b, u\} \) (where \( f \) stands for “forward”, \( b \) for “backward” and \( u \) for “unaffected”).

In our example, possible chase-step strategies are: (i) \( css_1 = \{f, f, f\} \), where all conclusion groups are repaired in a forward way; (ii) \( css_2 = \{f, b, f\} \), where the second conclusion group is repaired in a backward way; (iii) \( css_3 = \{u, f, f\} \), where the first conclusion group remains unaffected.

Notice that for a given update \( Upd \) and egd \( e \), several different chase step strategies may exist: one for each different chase step strategy of any homomorphism class \( H \) that generates a violation for \( Upd \) and \( e \). We denote by \( css_{Upd} \) the set of all possible chase-step strategies for \( Upd \).

In our example, chase step strategies \( css_1, css_2 \) and \( css_3 \) over \( \langle I, Upd_0(J) \rangle \) generates the following updates:

1. \( Upd_{css_1} \), contains, in the place of conclusion groups \( g_1, g_2 \) and \( g_3 \), the new cell group \( Union(g_1,g_2,g_3) = (L_1 \rightarrow \{t_1, B_{[2]}, t_2, B_{[2]}, t_3, B_{[4]}\}, by 0) \);
2. in order to generate \( Upd_{css_2} \) we need to forward chase \( g_1 \) and \( g_3 \), and change in a backward way the cell group associated to the cell \( t_2A \) in \( Upd_{J} \). The result is an update with two cell groups: \( Union(g_1,g_3) = (L_2 \rightarrow \{t_1, B_{[2]}, t_3, B_{[4]}\}, by 0, with \{c_x\}) \).
3. in \( \text{Upd}_{\text{css}} \), we leave untouched \( g_1 \), and we merge \( g_2 \) and \( g_3 \) in the new cell group \( \text{Union}(g_2, g_3) = (L_2 \rightarrow \{t_2, B_2\}, t_3, B_3\}, \) by \( \emptyset \).

**Introducing the Cost Manager** The notion of a chase-step strategy provides a basis to introduce a key component of our chase procedure, called the *cost manager*. Cost managers represent an elegant and principled way to incorporate pruning strategies into the chase, and, as it will be discussed in the following Sections, allow us to capture many of the practical optimizations that have been used in the literature to tame the complexity of the repair process. These include various notions of minimality of the updates [5] [9] [8], certain regions [17], and sampling [8].

**Definition 23 (Cost Manager)** A *cost manager* for a scenario \( \mathcal{M} \) and instance \( \langle I, J \rangle \) is a predicate \( \text{CM} \) over chase-step strategies. For each chase-step strategy \( \text{css} \) that is considered during the chase, it may either accept \((\text{CM}(\text{css}) = \text{true})\) or refuse it \((\text{CM}(\text{css}) = \text{false})\).

In essence, we complement each chase procedure by an appropriate cost manager, and only generate nodes in the chase tree that correspond to repair strategies accepted by the cost manager. The standard cost manager is the one that accepts all chase step strategies, and may be used for very small scenarios. Alternative cost managers are the following:

- a *maximum size* cost manager \((\text{SN})\): it accepts repair strategies as long as the number of leaves in the chase tree (i.e., the updates produced so far) are less than \( N \); as soon as the size of the chase tree exceeds \( N \), it accepts only the first one of them, and rejects the rest; as a specific case, the \( \text{SN} \) cost manager only generates one path in the chase tree, and ignores other branches;
- a *forward-only* cost manager \((\text{FO})\): it accepts forward-only repair strategies, and rejects backward updates;
- a *sampling* cost manager \((\text{SPL}k)\): it randomly accepts repair strategies, until \( k \) solutions have been generated;
- a *certain-region* cost manager \((\text{CTN})\): it incorporates the notion of a *certain region* [17] in the target, i.e., a set of attributes \( A \) that are considered “fixed”, i.e., reliable, and cannot be changed; it refuses chase steps with changes to attributes in \( A \).

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

## 16 Comparison to Data Repairing Semantics

We have already shown in Section 13 that our general notion of a mapping and cleaning scenario can be restricted to mapping scenarios on one side, and cleaning scenarios, on the other side, and discussed the relationship to the semantics of data exchange.

In this section, we want to develop the comparison of mapping and cleaning scenarios to some of the semantics that have been proposed for data repairing [9, 11, 27, 8]. Since the partial order stands at the core of our approach, we also consider a few other recent proposals [36, 10] that have dealt with notions of preference in connection with data quality constraints.

### 16.1 The Minimum Cost Algorithm

Let us start our discussion with early works on repairing by cell-modifications [9], and algorithms for conditional dependencies [11, 27].

We find it useful to compare our framework to the algorithm by Bohannon and others [9] by using a simple data repairing problem where we are given a database table with schema \( T \), a set of functional dependencies \( F \) over it, and an instance \( J \) of \( T \) that is dirty wrt \( F \). We want to repair \( J \) by using the semantics given in [9], which we call the *minimum cost semantics*. The main ideas behind this semantics are the following:

(i) a *repair* is any instance of \( J' \) that is the result of updating \( J \) and satisfies the constraints in \( F \);

(ii) we seek repairs that *minimally differ* from \( J \); the notion of minimality is based on the cost of updating \( J \) to obtain a repair \( J' \), according to an appropriate cost function;

(iii) the actual repair algorithm is organized in two phases; in the first one the goal is to build equivalence classes (as introduced in Section 15), i.e., groups of cells that ultimately will become equal according to the dependencies, without
actually modifying them; the decision about the value to update the cell is deferred to phase two, when its equivalence class has been determined, and more informed decisions can be taken.

Consider table \( R(A,B,C) \) in Figure 9, with a simple functional dependency \( d : A \rightarrow B \). The minimum cost algorithm would first build equivalence classes of cells (in our example \( c_1 \) for value 1, \( c_2 \) for value 2 of the \( A \) attribute), and then figure out a way to repair them. This is done by minimizing an elaborate cost function [9] that mixes together various features, like string similarity and confidence values. In its simplest form, however, all updates have the same unitary cost. Thus, to minimize the cost the algorithm repairs each equivalence class by the value with the highest frequency.

Two minimum cost repairs for our example are \( R_1, R_2 \) in Figure 9. Notice how we do not have a clear policy to pick up a preferred value for the second equivalence class, since both values have equal frequency. Repair \( R_3 \), on the contrary, is not minimal: by picking 4 as a preferred value for the first equivalence class, it generates an higher number of updates. It is also worth noting that, according to the semantics in [9], it is perfectly acceptable – although not minimal – to update the cells in \( c_2 \) to some arbitrary value, 7, that is different from both 5 and 6.

We now introduce a cleaning scenario that mimics this semantics. We have an empty source database, a target database that coincides with \( R \), and a single egd that encodes the functional dependency \( d \). We assume that no backward changes are allowed, i.e., we use a forward-only cost manager, and the user function is empty. The crux is to properly construct the partial order specification. Suppose we are given a dirty instance \( R \) that we want to repair. Our goal is to associate an ordering attribute to attribute \( B \) in \( R \). From the theoretical viewpoint, this can be done by building a table \( R' \) that is obtained from \( R \) by adding a new attribute \( fr \). For each tuple \( t \in R \), the value of \( t.fr \) is the frequency of the value of \( t.B \), as shown in Figure 9. We then specify \( fr \) with the natural order of integer numbers as the ordering attribute for \( B \). In practice, of course, this is not necessary. We may consider \( fr \) as a virtual attribute, and compute value frequencies on the fly. It is easy to see that our semantics applied to \( R' \) yields the minimal solution \( R_4 \) in Figure 9. There are a few important things to notice:

(i) there is a relationship (as we also discussed in Section 15) between equivalence classes and cell groups; in fact, in this example two cell groups are generated in the end, the first having occurrences that coincide with \( c_1 \), the second with \( c_2 \); cell groups, however, are more sophisticated than equivalence classes, since their semantics is such that we do not need to separate the class-construction step from the value-selection one. This is due to the monotonicity property of cell groups: they may only increase in size, and at every step they keep upgrading the quality of the target database;

(ii) our semantics is based on the assumption that arbitrary choices are to be avoided, because they correspond to unjustified ways of updating the target. To start, \( R_3 \) is not even a solution in our approach: a cell group that updates cells \( \{t_4,B,t_5,B\} \) cannot have value 7, since the constant is not motivated either by some occurrence, justification or user-input. In addition, our chase wouldn’t even choose between 5 or 6 for that cell, because the two values are incomparable according to the partial order, and introduce a llun instead, that needs to be resolved later on by the user;

(iii) if we wanted to make an explicit choice between \( R_1 \) and \( R_2 \) without resorting to users, then we need to refine our partial order in such a way that all values for the \( fr \) attribute are different. One way to do this is to say that we use both frequency and value, and whenever two values are equally frequent, we pick up the higher one. We did this in table \( R'' \) in Figure 9. A minimal solution for \( R'' \) would be \( R_1 \); \( R_4 \) is a non-minimal solution for \( R'' \), while \( R_2 \) is not even a solution in this case.

Notice that the algorithms in [9, 11, 27] deal with a few more features. We quickly discuss them here.

(a) more sophisticated metrics and confidence values: it is easy to generalize the frequency approach we have introduced here to incorporate string similarity or some other forms of value distance. In a similar way it is possible to handle confidence values;

(b) conditional functional dependencies and backward chasing [11]: CFDs are considered as a source of authoritative
values in our approach; they nicely blend with the partial order and need no ad hoc treatment. Similarly for backward chasing:

(c) inclusion dependencies and conditional inclusion dependencies [9, 27]: these papers develop hoc algorithms to handle the interaction of functional and inclusion dependencies. On the contrary, in our approach their interaction is nicely handled by our chase over cell groups.

16.2 The Sampling Algorithm

To further emphasize the flexibility of our approach, let us consider a second repair algorithm from the literature, that is different in spirit from the minimum-cost algorithm discussed above. Beskales and others [8] combine together forward and backward chasing, a random strategy to repair cells, and sampling to reduce complexity. We call this the sampling algorithm. In essence, whenever a violation for an FD is detected, this algorithm may randomly decide to forward or backward repair it. It also nondeterministically chooses whether to introduce a variable to repair the conflict, or a random value from the active domain of the dirty table. The space of repairs is sampled in order to generate \( k \) repairs that have a minimality property.

We use two different techniques to reproduce this semantics in our approach. The challenge is to design a partial order that “simulates” the random selection of values. We proceed along the lines of what we did with the frequency attribute. In this case, we start with the dirty database, and associate with each attribute \( A \) a (virtual) additional attribute \( rndA \); for each cell of \( A \), we initialize the corresponding cell of \( rndA \) with a randomly assigned value. This guarantees that values will be preferred to each other in a completely random way. Then, we adopt a sampling cost manager, that randomly decides whether to accept or refuse a chase step, until \( k \) solutions have been generated.

16.3 Prioritized Repairing

We now compare our partial order with different notions of preference that have been recently introduced in connection with data quality constraints. We start by considering prioritized repairing [36]. This research is inspired by works on preferred models for logic programs, and is somehow different in spirit from our semantics. While we focus primarily on obtaining preferred solutions by means of a general chase procedure, their focus is on the complexity of repair checking, and on consistent query answering.

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

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

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

First, we notice that \( J \) is not a repair in their setting, since any of their repairs must either correspond 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 can easily simulate their semantics. Suppose, in fact, we say that for attribute \( B \) we prefer cells with lower values, while for attribute \( C \) the ones with higher values; then we have a minimal solution: \( J' = \{ t_1 = R(a, 1, 4) \} \) that coincides with their globally optimal repair.

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

16.4 Relative Accuracy

We conclude by discussing a recent work by Cao and others [10]. This paper studies a rather specific problem, called by the authors the accuracy problem which falls within the reach of entity resolution rather than constraint-based data repairing. It is formulated as follows: we are given a set of records \( I_e \) with the same schema, that correspond to a
description of a single real world entity \( e \). These records may have conflicting values, and the goal is to derive a single tuple \( t_e \), which we call the *entity tuple*, with the *most accurate values* for all attributes. Master-data tuples may be used during the process.

One example of this problem is shown in Figure 10. Consider for now only tuples \( t_1 \) to \( t_4 \), that refer to Michael Jordan. The goal is to unify them within a single tuple that reflects the most accurate values for season 1994-95.

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

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

\[
    a_1 : \forall t_1, t_2 \in \text{Stats} : (t_1 \cdot \text{leag} = t_2 \cdot \text{leag}, t_1 \cdot \text{rnds} < t_2 \cdot \text{rnds} \rightarrow t_1 \prec_{\text{rnds}} t_2)
\]

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

\[
    a_2 : \forall t_1, t_2 \in \text{Stats} : (t_1 \prec_{\text{rnds}} t_2 \rightarrow t_1 \prec_{\text{TotalPts}} t_2)
\]

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

(ii) on the other side, they may be used to correct the entity tuple \( t_e \) based on master data tuples, like the ones shown in Figure 10 within table NBA.

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

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

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

On the contrary, the relative accuracy algorithm adopts a dynamic strategy to derive the partial order by interleaving the firing of accuracy rules and master–data rules. Despite the fact that our definition of the partial order is static, we believe that our semantics guarantees most of the benefits of accuracy rules, without the associated shortcomings. The intuition behind this is that accuracy rules in essence do two things: (i) fix a natural order for the values of an attribute, as discussed above for rule \( a_1 \); (ii) they propagate this ordering to other attributes, as in rule \( a_2 \) above. But this is exactly what our partial order specification does.
To show this, in Figure 10 we rephrase the Michael Jordan example as a cleaning scenario. Recall that the original problem was not a data repairing problem. Therefore we needed to make some changes, to adapt it to our setting. First, the target table Stats now may freely contain data about different players (we added tuples for Larry Bird as well). Second, we needed to specify a set of data quality constraints for this example to state that ID is a key, in order to trigger the repair of tuples for each player, and an editing rule to correct tuples using master-data. In Figure 10 we also report our partial order specification for this example, and the minimal solution returned by our algorithm (duplicate tuples have been removed for the sake of readability), which is in line with the expected results reported by Cao and others.

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

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

17 Scalability and Optimizations

In this section we introduce a number of additional techniques that were necessary in order to improve the scalability of our chase engine. These consist of an ad-hoc representation system for chase trees called delta databases, and several optimizations in the implementation of chase steps.

17.1 Delta Databases

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

Delta relations rely on an attribute-level storage system, inspired by U-relations [3], modified to efficiently store cell groups and chase sequences. More specifically, (i) each column in the original database is stored in a separate delta relation, to be able to record cell-level changes; (ii) chase steps are identified by a function with a prefix property, such that the id of the father of \( n \) is a prefix of the encoding of \( n \); this allows to quickly reconstruct the state of the database at any given step, using fast SQL queries; (iii) additional tables are used to store cell groups, i.e., occurrences and justifications.

More formally, we introduce a function \( \text{stepId}(\cdot) \) that associates a string id with each chase step, i.e., with each node in the chase tree, and has the prefix property such that for each \( n \), \( \text{stepId}(\text{father}(n)) \) is a prefix of \( \text{stepId}(n) \). For this, we use the function that assigns the id \( r \) to the root, \( r0, r1, \ldots, r.n \) to its children, and so on.

Given a target database schema \( \mathcal{R} = \{R_1, \ldots, R_k\} \), a delta database for \( \mathcal{R} \) contains the following tables: (i) a delta table \( R_i.A_j \) with attributes \( (t.id, \text{stepId}, value) \), for each \( R_i \) and each attribute \( A_j \) of \( R_i \); (ii) a table occurrences, with schema \( (\text{stepId}, value, t.id, table, attr) \); (iii) a table justifications, with schema \( (\text{stepId}, value, t.id, table, attr) \).

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

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

17.2 Caching Cell Groups

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

\[
\text{insert into } R_3 \text{ select } x, \text{Sk}(x) \text{ from } R_1, R_2 \text{ where } R_1.x = R_2.x
\]

We call this a batch chase execution. In fact, chasing s-t tgds, or even the more general FO-rules [30] is extremely fast. On the contrary, the chase becomes slow whenever it needs to be executed in a violation-by-violation fashion. Unfortunately, our chase procedure does not allow for easy batch-mode executions: during the chase, we need to keep track of cell groups, and properly maintain them. Repairing a violation for either a tgd or an egd changes the set of cell groups, and therefore may influence other violations.

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

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

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

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

To find the best compromise between storage-efficiency and performance, we noticed that our chase algorithm has a high degree of locality. In fact, when chasing node $s$ in the tree to generate its children, only cell groups 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. We therefore designed a single-step, greedy caching strategy, that caches cell groups for a single step at a time. In essence, we keep track of the step $s$ currently in the cache; whenever a cell group for a different step $s'$ is requested, we clean the cache and load all cell groups for $s'$. Our experiments show that this brings excellent improvements in terms of running times.

Additional optimizations to the chase have been discussed in [21].

## 18 Experimental Results

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

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

### Datasets and Scenarios

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

(a) Hospital-Norm is the normalized version of the hospital data, of which we considered 3 tables with 2 foreign keys, a total of 20 attributes, and approximately 150K tuples.
(b) Hospital-Den is a highly denormalized version of the same data, with 100K tuples and 19 attributes in one table, over which we specified 9 functional dependencies. This second version has traditionally been used in data cleaning experiments to test algorithms that were restricted to single-table databases. For both Hospital datasets, in our scalability tests we generated instances of size up to 1M tuples by replicating the original data several times.

(c) Customers, corresponds to our running example in Figure 1. The source database schemas contain 3 tables, plus 1 master data table and 2 additional tables encoding constants in CFDs. The target database schema contains 2 tables. We synthetically generated up to 1M tuples for the 4 source tables, with a proportion of 40% in MedTreatments, 40% in Surgeries, and 20% in Patients; the master-data table contains a few hundreds of the tuples present in MedTreatments and in Patients. We consider master-data tuples outside of the total, as they cannot be modified. For the target, we generated up to 1M tuples, with a proportion of 40% in the Customers table, and 60% in Treatments.

Based on these datasets, we defined 5 scenarios of different kinds. For each scenario we also fixed an expected solution, called \( DB_{exp} \), as follows:

(i) a data exchange scenario Customers-DE based on a version of the Customers dataset for which there are no conflicts among the sources and an empty target database; the expected instance is the core universal solution \( C \) for the set of tgds and egds in Section 1;

(ii) a cleaning scenario Hospital-Den-CL based on the Hospital-Den dataset, with 1 table, 9 functional dependencies, and the standard partial order specification; the expected instance is the original table;

(iii) a cleaning scenario Customers-CL based on the Customers dataset, with the 2 target tables Customers and Treatments, 3 source tables (1 master data table and 2 additional tables encoding constants in CFDs), the 9 extended egds reported in Section 4, and the partial order discussed in Section 5; the expected instance is the original table;

(iv) a mapping & cleaning scenario Hospital-Norm-MC based on the Hospital-Norm dataset, with 3 tables, 2 tgds and 12 egds, and the standard partial order specification; the expected instance corresponds to the original tables;

(v) a mapping & cleaning scenario Customers-MC based on the Customers dataset, with the set of tgds and egds in Section 1 (a total of 6 source tables, 2 target tables, 3 tgds, and 9 egds), the partial order in Section 5, and a non-empty target database; since we are integrating several sources, fixing the expected instance in this case is not obvious. We consider the clean and consistent versions of the source tables used for scenario Customers-DE, and the core universal solution, \( C \), of the mapping scenario. Then, we introduced random noise and inconsistencies in the sources, and fed them to the mapping and cleaning scenario. We intend to measure to which extent our algorithm is capable of generating a consistent and minimal repair of the target database. To do this, we adopt as an expected solution the core universal solution \( C \) above.

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

Four additional synthetic scenarios were used to test the scalability of the algorithm wrt larger number of relations and dependencies. Using the scenario generator developed for STBenchmark [2], we generated four relational scenarios \((s_{25}, s_{50}, s_{75}, s_{100})\) containing 20/50/75/100 tables, with an average join path length of 3, variance 1. To generate complex schemas we used a composition of basic cases, as follows: Vertical Partitioning (3/6/11/15 repetitions), Denormalization (3/6/12/15), and Copy (1 repetition). With such settings we got schemas varying between 11 relations with 3 joins and 52 relations with 29 joins. The number of tgds varies between 22 and 93. We tested each scenario with a number of egds varying from 0 to 25.

**Error Induction** In order to test our algorithms with different levels of noise, we introduced errors in the datasets. Part of these errors were generated by a random-noise generator. However, to be as close as possible to real scenarios, in the Hospital datasets we also used a different source of noise. We asked workers from Mechanical Turk (MT) (https://www.mturk.com/mturk/) to perform data entry for a random sample of tuples from the original database. Workers were shown the original tuple under the form of a jpeg image, and needed to manually copy values into a form. We used different groups of workers with different approval rates; approval rates measure the quality of a worker in terms of the percentage of previous jobs positively evaluated within MT. Approval rates varied between 50% and 99%; for these, we observed a percentage of wrong values between 5% and 1%. These errors were then complemented with those generated by the random noise generator. Errors have been added to the other datasets with the same procedure.
In this case we report a repair rate defined as approximately 90% similar to the clean one, and therefore all repairs will also have high similarity to the clean instance.

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

**Metric 0.5.** The first one is the one adopted in [8], which we call Metric 0.5: (i) for each cell $c \in C_r$ repaired to the original value in $C_p$, the score was 1; (ii) for each cell $c \in C_r$ changed into a value different from the one in $C_p$, the score was 0; (iii) for each cell $c \in C_r$ repaired to a variable value, if the cell was also in $C_p$, the score was 0.5. In essence, a llun or a variable is counted as a partially correct change. This gives an estimate of precision and recall when variables are considered as a partial match.

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

In mapping & cleaning scenarios, on the contrary, we may have different tables, referential integrity constraints, and the addition of new cells to the target. The presence of new cells makes it impossible to reuse the traditional metrics. Given a clean target database, we need for each repair a general algorithm to measure the similarity of the whole, multi-repairable database to the expected target database. A general and efficient algorithm to measure the similarity of two complex databases by taking into account foreign keys, different cell ids, and placeholders, like labeled nulls or lluns has been recently developed in [31], and we adopt it for this metric. Based on this algorithm, we report two different quality measures.

**Metric Sim.** The first one is the similarity, $sim(Upd, DB_{exp})$, measured by the algorithm in [31]. In the comparison, lluns are considered as partial matches, and counted as 0.5 each.

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

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

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

**Algorithms** We ran LLUNATIC with several cost managers and several caching strategies, as discussed in Sections 15, 17. In particular we used a new kind of cost manager, called frequency cost manager (FR), that adopts the following rules in order to repair an homomorphism class $\mathcal{H}$ for dependency $e$: it relies on the frequency of values appearing in conclusion cells, and on a similarity measure for values (based on the Levenshtein distance for strings); then: (i) it rejects repair strategies that backward-chase cells with the most frequent conclusion value; (ii) for every other conclusion cell, if its value is similar (distance below a fixed threshold) to the most frequent one, the cell is forward-chased; otherwise, it is backward-chased; this is typically used with a frequency rule in the partial order of cell-groups.

We chose variants of the LLUN-FR-SN cost manager – the frequency cost-manager that generates up to $N$ solutions – with $N = 1, 10, 50$, and the LLUN-FR-S1-FO, the forward-only variant of LLUN-FR-S1. We do not report results obtained by the standard cost manager, as it only can be used with small instances due to its high computing times.

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

The DEMO system [32], as the state of the art chase engine for mappings.

Three repair systems:

(a) **Minimum Cost** [9] (MIN. COST);
(b) **Vertex Cover** [25] (VERTEX COVER);
(c) **Repair Sampling** [8] (SAMPLING), for which, for each experiment, we took 500 samples, as done in the original paper.
Finally, we used \textsc{Min. Cost} for repair scenarios with FDs and IDs (in which IDs are repaired only by tuple insertions, and not by deletions or modifications), and an implementation of the \textsc{Pipeline} algorithm in Section 4 for mapping and cleaning scenarios. The latter is obtained by coupling a standard chase engine for tgds, and the \textsc{Sampling} algorithm for FDs in [8]; here, for each experiment, we took 100 samples.

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

\textbf{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 \textsc{Sampling}, which is based on a sampling of the possible repairs and has no guarantee that the best repair is computed first.

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

\textbf{The \textsc{Customers-DE} Experiment} We start by showing the scalability of our chase engine in Figure 11.d. We compare the performance of \textsc{LLUNatic} to the data exchange chase engine \textsc{Demo} on scenario \textsc{Customers-DE}. It can be seen that our implementation is orders of magnitude faster than \textsc{Demo}.

\textbf{The \textsc{Hospital-Den-CL} Experiment} We report in Figures 11.a–c scalability results for some of our cost managers and the caching strategies discussed in Section 17 (single step, greedy, lazy). The charts confirm that, due to the locality of the chase algorithm, the single-step cache represents the best choice in terms of performance. Further experiments were performed with a single-step cache manager.

Figures 11.a–d clearly show the benefits that come with a DBMS implementation wrt main-memory ones, namely the possibility of scaling up to very large databases. While previous works [9, 8] have reported results up to a few thousand tuples, we were able to investigate the performance of the system on databases of millions of tuples. The figures show that \textsc{LLUNatic} scales in both data-exchange and cleaning scenarios to large databases. For \textsc{Hospital-Den} we replicated the original dataset ten times with 1% errors. In these cases, execution times in the order of an hour for millions of tuples can be considered as a remarkable result, since no system had been able to achieve them before on problems of such exponential complexity.

The trade-offs between quality and scalability are shown in Figures 11.e–g. We start by showing that \textsc{LLUNatic} produces repairs of significantly higher quality wrt those produced by previous algorithms. We ran the chase with the cost managers listed above, and the three competing algorithms on samples of the \textsc{Hospital} dataset with increasing size (5k to 25k tuples) and increasing percentage of errors (1% to 5%).

The maximum F-measure for Metric 1 is in Figure 11.e; for the two algorithms that return more than one solution, the minimum and average F-measures are reported in Figure 11.f. The maximum F-measure for Metric 0.5 is in Figure 11.g. Quality results for algorithms \textsc{Min. Cost}, \textsc{Vertex Cover}, and \textsc{Sampling} are consistent with those reported in [8], which also conducted a comparison of these three algorithms on scenarios in which forward and backward repairs were necessary.

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

The superior quality achieved by \textsc{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 ours, there are other algorithm capable of backward repairs. However, our chase algorithm explores the space of solutions in a more systematic way, and this explains its improvements in quality. In light of this, the superior quality achieved by the \textsc{LLUNatic} variants, which clearly outperformed the competitors, is a significant improvement.

Figure 11.h compares execution times for the various algorithms on \textsc{Hospital-Den} dataset up to 100K tuples, with 1% perturbation. Recall that \textsc{LLUNatic} is the first DBMS-based implementation of a data repairing algorithm. Therefore, our implementation is somehow disfavored in this comparison. To see this, consider that, when producing repairs, main-memory algorithms may aggressively use hash-based data structures to speed-up the computation of repairs, at the cost of using more memory. Using the DBMS, our algorithm is constrained to use SQL for accessing and repairing data; to see how this changes the cost of a repair, consider that even updating a single cell (a very quick operation when performed in
main memory) when using the DBMS requires to perform an UPDATE, and therefore a SELECT to locate the right tuple.

Nevertheless, the LLUN-FR-S1 cost manager scales nicely and has better performances than some of the main memory implementations. We may therefore say that graphs e–h in Figure 11 give us a concrete perception of the trade-offs between complexity and accuracy, and allow us to say that the LLUN-FR-S1 is the best compromise for the HOSPITAL scenario. Other algorithms do not allow to fine tune this trade-off. To see an example, consider the SAMPLING algorithm: we noticed that taking 1000 samples instead of 500 doubles execution times, but it does not produce significant improvements in quality.

**The Customers-CL Experiment** Figures 11.7 reports quality results for the Customers-CL scenario. Recall that LLUNATIC is the first system that is able to handle such kind of scenarios with complex constraints. We notice that quality results are better than those on Hospital-Den-CL; this is a consequence of the clear user-specified preference rules.

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

**The Customers-MC Experiment** The overall scalability of the chase is confirmed on scenario Customers-MC in Figure 11.8. In fact, the normalized nature of the data guarantees performance results that are significantly better than those reported for the denormalized scenario in Hospital-Den-CL, even though in this case we are chasing tgds and egds together.

The execution times achieved by the algorithm can be considered as a remarkable result for problems of this complexity. They are even more surprising if we consider the size of the chase trees that our algorithm computes, which may reach several hundreds of nodes as reported in Figure 11.9. Consider also that each node in the tree is a copy of the entire database. It is also worth noting that storing chase trees as delta databases is crucial in order to achieve such a level of scalability. Without such a representation system times would be orders of magnitude higher.

Figures 11.10–11 report the quality achieved by the various cost managers, in terms of the similarity to the core instance, sim(Upd, DBexp). No other system is capable of handling scenarios of this complexity, and therefore no baseline is available. Notice that achieving 100% quality is in some cases impossible, since the sources have been made dirty in a random way, and some conflicts are not even detected by the dependencies. However, quality of the solutions is very high. This is a consequence of the rich preference rules that come with this scenario.

**The Hospital-Norm-MC Experiment** Figure 11.12 confirms the excellent scalability of chasing tgds and egds on normalized databases, even with chase trees of large size (Figure 11.13). We do not report computation times for the PIPELINE and MIN.COST algorithms since they were designed to run in main memory and do not scale to large databases.

In terms of quality, we notice that finding the right repairs for Hospital-Norm-MC is quite hard, since here we have no preference relations, and there is very little redundancy in the tables. In Figure 11.14 we report metric rep-rate(Upd, DBexp) for the three algorithms that we ran on this scenario. Two things are apparent: LLUNATIC was able to partially repair the dirty database, but the overall quality was lower than the maximum one achieved in scenario Customers-MC.

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

**Scalability with Large Numbers of Dependencies** Figure 11.15 reports executions time for the chase over the four synthetic scenarios $s_{25}, s_{50}, s_{75}, s_{100}$, each run with an increasing number of egds (0, 5, 10, 25) over a source instance of 100K tuples. Execution times are all within a few minutes and show a linear trend w.r.t. the number of constraints.

**Impact of User Inputs** We finish by mentioning Figure 11.16, in which we study the impact of user inputs on the chase process. We run the experiment for 25K tuples interactively, and provided random user inputs by alternating the change of a llun value with the rejection of a leaf. It can be seen that small quantities of inputs from the user may significantly prune the size of the chase tree, and therefore speed-up the computation of solutions.

## 19 Related Works

There has been a host of work on both data exchange and data quality management (see [4] and [15, 33] for recent surveys, respectively).

We have discussed the relationship of mapping & cleaning with data exchange and several data repairing proposals in the previous sections. Here we discuss other approaches.
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 [8, 9, 11, 17, 18, 25]. These methods differ in the constraints that they admit, e.g., FDs [8, 9], CFDs [11, 25], inclusion dependencies [9], and editing rules [17], and the underlying techniques used to improve their effectiveness and efficiency, e.g., statistical inference [11], measures of the reliability of the data [9, 17], and user interaction [11].

All of these methods work for a specific class of constraints only, with the exception of [18, 23]. A flexible data quality system was recently proposed [12] which allows user-defined procedural code for detection and cleaning. These works explore the interaction among different kinds of dependencies, but they do not have a unified formal semantics with a definition of solution, neither the generality of our partial order to model preferences. Table 1 summarizes the features of LLUNATIC wrt some earlier approaches to data repairing.

More importantly, none of the above algorithms allow tgds between schemas, i.e., the mappings. However, some of the ingredients of our scenarios are inspired by, but different from, features of other repairing approaches: repairing based on both premise and conclusion of constraints [11, 8, 25], cells [8, 25, 9], groups of cells [9], partial orders and its incorporation in the chase [7]. We discuss these aspects in detail next.

We do allow for forward and backward chasing. Similarly, [11, 25, 8] 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 [25, 8] 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 [24] 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 [7], 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 [37]. In that work, a pre-order on tableau is defined and so-called up-repairs are introduced as a way of to compute consistent query answers in the presence of updates.

A crucial contribution of this paper consists in developing a new semantics to incorporate preference strategies in data repairing, as reported in Table 1. We discussed the comparison of our work to the ones on prioritized repairing [36] and accuracy rules [10] in Section 16.

This work is also related to prior work on truth discovery from data sources [13]. In fact, these methods first discover dependencies on sources, such as copy relationships, to identify reliable sources; then, they employ these statistics in a probabilistic vote counting to estimate the accuracy of tuples with inconsistent values.

Studies to guarantee scalability for data exchange scenarios were undertaken in [30, 29, 28], but they were based on the technique of rewriting dependencies to remove the need to chase egds.

### 20 Conclusions

This paper develops a framework to handle mapping & cleaning tasks within a single 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 target database; (c) a clear notion of an upgrade to the target database, based on a partial order of cell groups and updates.

In addition to the semantics, we have developed a number of techniques to implement the generation of solutions to mapping & cleaning scenarios within a scalable chase engine.

We believe that this work may provide the basis for further investigation on the subject of both data transformation and data repairing. On the one side, it proposes an end-to-end solution to deal with data integration problems in presence of inconsistencies. On the other side, it proposes a new conservative 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.

References


Figure 11: Experimental results.
A Formalization of the Revised Chase

We start by formalizing the revised chase procedure that was informally introduced in Section 15. Recall the definition of a witness and witness variable given in Section 14. Our goal is to group homomorphisms with equal witnesses together.

Definition 24 [HOMOMORPHISM CLASS] Given an update Upd, and an egd \( e : \forall \varphi(x) \rightarrow x = x' \), let \( \bar{x}_w \subseteq \bar{x} \) be the witness variables of \( e \). An homomorphism class for Upd and \( e \), \( \mathcal{H} \), is a set of homomorphisms of \( \phi(\bar{x}) \) into \( \langle I, \text{Upd}(J) \rangle \) such that all \( h_i \in \mathcal{H} \) have equal witness values \( h_i(\bar{x}_w) \).

We introduce the notion of witness groups and conclusion groups for an homomorphism class \( \mathcal{H} \):

Definition 25 [WITNESS GROUPS, CONCLUSION GROUPS] Given an homomorphism class \( \mathcal{H} \) for Upd and egd \( e \): (i) we call witness groups, \( w\text{-groups}_{\mathcal{H}} \), the set of cell groups associated by homomorphisms in \( \mathcal{H} \) with the witness variables, \( \bar{x}_w \); (ii) we call conclusion groups, \( c\text{-groups}_{\mathcal{H}} \), the set of cell groups associated by homomorphisms in \( \mathcal{H} \) with the conclusion variables, \( x, x' \), of \( e \):

\[
\begin{align*}
    w\text{-groups}_{\mathcal{H}} &= \{ g_h(x_w) \mid h \in \mathcal{H}, x_w \in \bar{x}_w \} \\
    c\text{-groups}_{\mathcal{H}} &= \{ g_h(x) \mid h \in \mathcal{H} \} \cup \{ g_h(x') \mid h \in \mathcal{H} \}
\end{align*}
\]

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

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

Definition 26 [REPAIR STRATEGY] A repair strategy \( rs \mathcal{H} \) for an homomorphism class \( \mathcal{H} \) is a mapping from the set of conclusion groups, \( c\text{-groups}_{\mathcal{H}} \), into the set \( \{ f, b, u \} \) (where \( f \) stands for “forward”, \( b \) for “backward” and \( u \) for “unaffected”). The forward groups, \( \text{forw-}g_{rs\mathcal{H}} \), are those groups \( g_i \) such that \( rs_{\mathcal{H}}(g_i) = f \), the backward groups, \( \text{back-}g_{rs\mathcal{H}} \), are those such that \( rs_{\mathcal{H}}(g_i) = b \) and the unaffected groups, \( \text{equi-}g_{rs\mathcal{H}} \), are those such that \( rs_{\mathcal{H}}(g_i) = u \), i.e., those that we don’t want to repair in that particular chase step.

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

1. there exists an homomorphism \( h \in \mathcal{H} \) that covers both \( c_i \) and \( w\text{-cell}_{rs\mathcal{H}}(c_i) \), i.e., it maps one of the conclusion variables of \( e \) into \( c_i \), and one of the witness variables into \( w\text{-cell}_{rs\mathcal{H}}(c_i) \);
2. the corresponding cell group \( g_i \) according to Upd has a constant value, i.e., \( \text{val}(g_i) \in \text{CONSTS} \);
3. the corresponding cell group \( g_i \) has empty justifications, i.e., \( \text{just}(g_i) = \emptyset \).

Repair strategies for homomorphism classes are the main building block to generate chase steps for egds. More specifically, the chase tree is built up by selecting appropriate chase-step strategies:

Definition 27 [CHASE STEP STRATEGY] Given \( \mathcal{M}C = \{ S, S_o, T, \Sigma_t, \Sigma_e, \Pi, \text{User} \} \), and an update Upd of \( J \), a chase step strategy \( css \) is a triple \( \{ e, \mathcal{H}, rs_{\mathcal{H}} \} \), where \( e \) is an egd applicable to \( \langle I, \text{Upd}(J) \rangle \) with homomorphism class \( \mathcal{H} \), and \( rs_{\mathcal{H}} \) is a repair strategy for \( \mathcal{H} \).

Notice that for a given update Upd and egd \( e \), several different chase step strategies may exist: once for each different repair strategy \( rs_{\mathcal{H}} \) of any homomorphism class \( \mathcal{H} \) that generates a violation for Upd and \( e \). We denote by \( css_{\text{Upd}} \) the set of all possible chase-step strategies for Upd. Based on chase-step strategies, we are now ready to rework the definition of a chase step for egds (chase steps for tgd and user-inputs remain unchanged).

Definition 28 [CHASE STEP FOR EGDs] Given a mapping & cleaning \( \mathcal{M}C = \{ S, S_o, T, \Sigma_t, \Sigma_e, \Pi, \text{User} \} \), and an update Upd of \( J \). For each chase step strategy \( css = \{ e, \mathcal{H}, rs_{\mathcal{H}} \} \in css_{\text{Upd}} \), a chase step generates a new update \( \text{Upd}_{\text{css}} \) defined as follows:

1. to start, we initialize \( \text{Upd}_{\text{css}} = \text{Upd} \)
2. then, we replace all forward groups in \( rs_{\mathcal{H}} \) by their least upper bound:

\[
\text{Upd}_{\text{css}} = \text{Upd}_{\text{css}} - \text{forw-}g_{rs_{\mathcal{H}}} \cup \text{Union(forw-}g_{rs_{\mathcal{H}}})
\]

41
3. finally, we add the backward changes, i.e., for each backward group \( g \in \text{back-} \text{gsn} \), and cell \( c_i \in \text{occ}(g) \), we replace \( g_i = g_{\text{upd}}(w\text{-cell}_{\text{gsn}}(c_i)) \) by the cell group \( g'_i = \{ L_i \rightarrow \text{occ}(g_i), \text{by just}(g_i), \text{with } \{ c_x \} \} \) (where \( L_i \) is a new \text{LLUN}), as follows:

\[
\text{Upd}_{\text{css}} = \text{Upd}_{\text{css}} - \{ g_i \} \cup \{ g'_i \}
\]

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

Given \( \text{Upd} \), each valid chase step strategy \( \text{css}^i \in \text{css}_{\text{Upd}} \) generates a different step, \( \text{Upd}_{\text{css}^i} \). As discussed in the previous section, we simultaneously consider all these chase steps in parallel to build the chase tree.

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

**Theorem 13:** Consider the chase tree \( \text{chase}_{\Sigma_t, \Sigma_e, \text{User}}((I, J)) \), generated by the chase of \( \mathcal{MC} \) over \( (I, J) \) as defined in Section 14. If the chase of \( (I, J) \) with \( \Sigma_t, \Sigma_e, \text{User} \) terminates, then the revised chase of \( (I, J) \) with \( \Sigma_t, \Sigma_e, \text{User} \) also terminates. In this case, the revised chase procedure generates a chase tree \( \text{revised-chase}_{\Sigma_t, \Sigma_e, \text{User}}((I, J)) \) such that for any node in \( \text{chase}_{\Sigma_t, \Sigma_e, \text{User}}((I, J)) \), there is an identical node in \( \text{revised-chase}_{\Sigma_t, \Sigma_e, \text{User}}((I, J)) \).

### B Proofs

**Proposition 1** There exist sets \( \Sigma_t \) of non-recursive tgds, \( \Sigma_e \) of cleaning egds, and instances \( (I, J) \) such that procedure \( \text{pipeline}_{\Sigma_t, \Sigma_e}((I, J)) \) does not return solutions.

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

**Proposition 2** The binary relation \( \preceq \) as specified in Definition 5 is a partial order.

**Proof:** Given \( \mathcal{M} \) and \( (I, J) \), we consider the set of cells \( \text{cells}(\mathcal{M}, (I, J)) = \text{cells}(J) \cup \text{new-cells}(J) \cup \text{cells}(I_{\text{na}}) \cup \text{cells}(I_0) \cup \text{metaCells} \). We need to show that \( \langle \text{cells}(\mathcal{M}, (I, J)), \preceq \rangle \) is a partially-ordered set. This amounts to show that \( \preceq \) is reflexive, antisymmetric, and transitive.

We denote by \( \text{const-cells}(J), \text{null-cells}(J) \) the set of cells of \( J \) that has a value in \( \text{CONSTs}, \text{NULLs} \), respectively. We regard the binary relation \( \preceq \) as a directed graph \( G_\preceq \) with cells in \( \text{cells}(\mathcal{M}, (I, J)) \) as vertices and a directed edge between them as specified by \( \preceq \). This graph can be represented in a compact way by the following adjacency matrix, which we call \( A \), in which cells are grouped in blocks, numbered 1–5 (recall that cells in \( \text{new-cells}(J) \) have different null values, and therefore are not ordered):

\[
\begin{array}{cccccc}
A & 1. c_T & 2. \text{cells}(I_a) & 3. c_x & 4. \text{const-cells}(J) \cup \text{cells}(I_{\text{na}}) & 5. \text{null-cells}(J) & 6. \text{new-cells}(J) \\
1. c_T & 1 & 0 & 0 & 0 & 0 & 0 \\
2. \text{cells}(I_a) & 1 & 1(=) & 0 & 0 & 0 & 0 \\
3. c_x & 1 & 1 & 1 & 0 & 0 & 0 \\
4. \text{const-cells}(J) \cup \text{cells}(I_{\text{na}}) & 1 & 1 & 0 & 1(=) & 0 & 0 \\
5. \text{null-cells}(J) & 1 & 1 & 0 & 1 & 1(=) & 0 \\
6. \text{new-cells}(J) & 1 & 1 & 0 & 1 & 0 & 1(=) \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
A & \text{cols 1} & \text{cols 2} & \text{cols 3} & \text{cols 4} & \text{cols 5} & \text{cols 6} \\
\text{row 1.} & 1 & 0 & 0 & 0 & 0 & 0 \\
\text{rows 2.} & 1 & I & 0 & 0 & 0 & 0 \\
\text{row 3.} & 1 & 1 & 1 & 0 & 0 & 0 \\
\text{rows 4.} & 1 & 1 & 0 & P & 0 & 0 \\
\text{row 5.} & 1 & 1 & 0 & 1 & I & 0 \\
\text{rows 6.} & 1 & 1 & 0 & 1 & 0 & I \\
\end{array}
\]

where: (i) \( I \) is the identity matrix; (ii) \( I \) is a matrix made of ones only; (iii) 0 is a block made of zeros only; (iv) \( P \) is the adjacency matrix of the partial order induced by the partial-order specification, \( \Pi \), over the cells of \( (I_{\text{na}}, J) \).
It can be seen that \( \preceq_{\Pi} \) is reflexive (\( \forall c : c \preceq_{\Pi} c \), and so all elements on the diagonal of the matrix are equal to 1). Since the matrix is lower triangular, \( \preceq_{\Pi} \) is also anti-symmetric.

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>row 1.</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 2.</td>
<td>1 + I</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 3.</td>
<td>3</td>
<td>1 + I</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 4.</td>
<td>2 + P</td>
<td>I + P</td>
<td>0</td>
<td>( P^2 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 5.</td>
<td>3 + I</td>
<td>1 + 2I</td>
<td>0</td>
<td>( P + I )</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td>row 6.</td>
<td>3 + I</td>
<td>2 + I</td>
<td>0</td>
<td>( P + I )</td>
<td>0</td>
<td>I</td>
</tr>
</tbody>
</table>

where 1, 2, 3 denote matrices made all of 1, 2, 3, respectively.

We need to show that any element of \( A^2 \) with a non-null value denotes a path of length 2. We now show that any of these elements is such that also the corresponding element in \( A \) is not null. This would guarantee that the transitivity property is satisfied.

Let us analyze the single blocks one by one. First, we notice that the property is surely guaranteed for elements of the first column, since it only contains non-null elements in \( A \). Similarly for cols. 2 (all elements but the first one are not null in \( A \)). Column 3 is identical in the two matrices.

Let us now consider cols. 4. The main issue is to prove that any non null element in \( P^2 \) is also non null in \( P \). Recall that these elements encode ordering relationships associated with the partial order specification, \( \Pi \). These correspond to a partial order by definition, and therefore they are also transitive. Therefore, the property is satisfied.

Finally, we notice that cols. 5., 6. are identical in the two matrices. This concludes the proof. □

**Proposition 3** Relation \( \preceq_{\Pi, \text{User}} \) among valid cell-groups over \( \langle I, J \rangle \) as specified in Definition 9 is a partial order.

**Proof:** It is sufficient to note that \( \preceq_{\Pi, \text{User}} \) coincides with set-containment, a well known partial order.

**Lemma 4** Any cleaning egd or extended tgd in which only target symbol appear is satisfied after repairs if and only if it is satisfied in the standard sense.

**Proof:** The if part is straightforward: any dependency that is satisfied in the standard sense is also satisfied after upgrades by definition.

Let us first prove the only if part for egds, and then for tgd. Consider egd \( \phi : \phi(x) \rightarrow x = x' \), an instance \( \langle I, J \rangle \), and repair Upd. We know that \( e \) contains only target symbols, so we may ignore \( I \) in the following. Given homomorphism \( h \) of \( \phi(x) \) into Upd(\( J \)), assume \( e \) is satisfied after upgrades, i.e., \( g_{h}(x) \preceq_{\Pi, \text{User}} g_{h}(x') \). We need to show that \( h(x) = h(x') \), i.e., \( e \) is satisfied in the standard sense for \( h \).

Consider the two cell groups \( g_{h}(x), g_{h}(x') \). Recall that they are defined as the union of occurrences and justifications of all cell groups associated by Upd with the cells of variable \( x, x' \), respectively. Let us denote by \( g_{\text{Upd}}(e) \) the cell group of cell \( e \) in Upd, then:

\[
g_{h}(x) = \langle h(x) \rightarrow \bigcup_{c \in \text{cell}_{S_{h}}(x)} \text{occ}(g_{\text{Upd}}(c)) \text{ by } e \in \text{cell}_{S_{h}}(x) \rangle \bigcup_{e \in \text{cell}_{S_{h}}(x)} \text{just}(g_{\text{Upd}}(c)), \ldots
\]

\[
g_{h}(x') = \langle h(x') \rightarrow \bigcup_{c \in \text{cell}_{S_{h}}(x')} \text{occ}(g_{\text{Upd}}(c)) \text{ by } e \in \text{cell}_{S_{h}}(x') \rangle \bigcup_{e \in \text{cell}_{S_{h}}(x')} \text{just}(g_{\text{Upd}}(c)), \ldots
\]

Since we know that \( g_{h}(x) \preceq_{\Pi, \text{User}} g_{h}(x') \), we also know that \( \text{occ}(g_{h}(x)) \subseteq \text{occ}(g_{h}(x')) \). Notice also that, since \( e \) only contains target symbols, all cells for \( x, x' \) according to \( h \) are target cells, and therefore both set of occurrences are not empty.

Notice also that all cells \( c \in \text{cell}_{S_{h}}(x) \) have the same value in Upd(\( J \)) (they are mapped by \( h \) to the same variable), and therefore all cell groups \( g_{\text{Upd}}(c) \) in Upd for these cells have the same value. Similarly for \( c' \in \text{cell}_{S_{h}}(x') \). Therefore, Upd(\( J \)) is such that all cells in \( \text{occ}(g_{h}(x)) \) have value \( h(x) \), and all cells in \( \text{occ}(g_{h}(x')) \) have value \( h(x') \). However, since \( \text{occ}(g_{h}(x)) \subseteq \text{occ}(g_{h}(x')) \), this may only happen if \( h(x) = h(x') \), and therefore \( e \) is satisfied in the standard sense for \( h \).
Let us now consider a tgd $m : \phi(x, z) \rightarrow \exists y : \psi(x, \bar{y})$. Given $J$ and Upd, assume $h$ maps $\phi(x, z)$ into Upd($J$) and $m$ is satisfied after upgrades for $h$. This means that Upd is an upgrade of the canonical repair for $m$ and $h$, $\text{Upd}_h^{\text{can}}$, i.e., $\text{Upd}_h^{\text{can}} \subseteq \text{UserUpd}$. We need to show that there is an extension $h'$ of $h$ that maps $\psi(x, \bar{y})$ into Upd($J$), i.e., $m$ is satisfied in the standard sense for $h$.

Since $\text{Upd}_h^{\text{can}} \subseteq \text{UserUpd}$, we know that there exists an id mapping $h_{\text{id}}$ from tuple ids in $\text{Upd}_h^{\text{can}}(J)$ to tuple ids in $\text{Upd}(J)$ such that for each cell group $g \in \text{Upd}_h^{\text{can}}$, there exists a cell group $g' \in \text{Upd}$ such that $h_{\text{id}}(g) \subseteq \text{User}(g')$. Recall that $\text{Upd}_h^{\text{can}}(J)$ is obtained by taking the set of tuples in $\text{Upd}(J)$, and adding the new tuples in $h_{\text{can}}(\psi(x, \bar{y}))$ to it. Therefore, by definition of an id mapping, $h_{\text{id}}$ is the identity on the tuples in $\text{Upd}(J)$.

To show that $m$ is satisfied in the standard sense, we need to prove that $\text{Upd}(J)$ contains a set of tuples of the form $\psi(h(x), \bar{b})$, for some vector of values $\bar{b}$. In this case, in fact, we can extend $h$ by mapping $\bar{y}$ to $\bar{b}$, and have an homomorphism of $\psi(x, \bar{y})$ into $\text{Upd}(J)$.

Consider the set of tuples inserted into $\text{Upd}(J)$ by $\text{Upd}_h^{\text{can}}$, i.e., $\Delta_h^{\text{can}} = \psi(h_{\text{can}}(x), h_{\text{can}}(y))$, and recall that $h_{\text{can}}(x) = h(x)$ by construction. Let us consider the set of tuples:

$$h_{\text{id}}(\Delta_h^{\text{can}}) = h_{\text{id}}(\psi(h(x), h_{\text{can}}(\bar{y})))$$

that is the image of $\Delta_h^{\text{can}}$ in $\text{Upd}(J)$. We know that this image exists by definition of $\subseteq \text{User}$. We will now show that this set of tuples is of the form $\psi(h(x), \bar{b})$, and therefore $m$ is satisfied in the standard sense by $\text{Upd}$ for $h$.

This amounts to show the following:

1. for each $x_i \in \bar{x}$, and for each cell $c \in \text{cells}_{h_{\text{can}}(x_i)}$, the value of the cell is preserved by the id mapping, i.e., $\text{val}(h_{\text{id}}(c)) = \text{val}(c) = h(x_i)$;

2. for each $y_i \in \bar{y}$, there exist a value $b_i \in \text{NULLS} \cup \text{CONSTS} \cup \text{LLUNS}$ such that, for each cell $c \in \text{cells}_{h_{\text{can}}(y_i)}$, the value of $\text{val}(h_{\text{id}}(c))$ is equal exactly to $b_i$.

Let us first consider item 1. Consider $x_i \in \bar{x}$, and the corresponding cell group $g_{h_{\text{can}}(x_i)}$ in $\text{Upd}_h^{\text{can}}$. We know that it has the following form:

$$g_h(x_i) = (h(x_i) \rightarrow \bigcup_{c \in \text{cells}_{h_{\text{can}}(x_i)}} \text{occ}(g_{h_{\text{can}}}(c)) \text{ by } \bigcup_{c \in \text{cells}_{h_{\text{can}}(x_i)}} \text{just}(g_{h_{\text{can}}}(c)), \ldots)$$

Consider a cell $c \in \text{cells}_{h_{\text{can}}(x_i)}$, and the corresponding cell group, $g_c = g_{h_{\text{can}}}(c)$. By definition of the partial order over cell groups, we know that there exists a cell group $g'_c$ in $\text{Upd}$ such that $h_{\text{id}}(g_c) \subseteq \text{User}(g'_c)$. Consider now the occurrences of $g_c$. Since $m$ contains by hypothesis only target symbols, these occurrences are cells that appear in part in $\text{Upd}(J)$, in part in $\Delta_h^{\text{can}}$. In fact, the cells in $\text{Upd}(J)$ are those that atoms in the premise of $m$, $\phi(x)$, are mapped to, while those in $\Delta_h^{\text{can}}$ are those that belong to the new tuples added to $\text{Upd}(J)$ to satisfy the tgd in the canonical way.

Let us now consider the cell group $g'_c$ such that $h_{\text{id}}(g_c) \subseteq \text{User}(g'_c)$. We know that occurrences of $g'_c$ are such that they contain the set of cells $h_{\text{id}}(\text{occ}(g_c))$. Among these cells, those in $\text{Upd}(J)$ are mapped to themselves by definition of id mapping. Since, as discussed above, this set is not empty, $g_c$ and $g'_c$ have at least one occurrence in common. We know that all occurrences in $\text{occ}(g_c)$ that come from $\text{Upd}(J)$ have the same value in $\text{Upd}(J)$. Similarly for occurrences of $g'_c$. Since the two sets have an element in common, it must be the case that $\text{val}(g'_c) = h(x_i)$. Since this is true for each $c \in \text{cells}_{h_{\text{can}}(x_i)}$, and each $x_i \in \bar{x}$, then item 1 holds.

With a similar argument it is possible to show that also item 2 holds. In fact, the containment property for cell groups can be used to show that all cells $c$ for a variable $y_i$ have the same value $b_i$.

This concludes the proof. \hfill $\Box$

**Theorem 5** Every core solution $J_{\text{core}}$ of a data exchange scenario corresponds to a minimal solution $\text{Upd}_{\min}$ of its associated mapping scenario. Every minimal solution $\text{Upd}_{\min}$ corresponds to a solution $J_{\min}$ of the data exchange scenario. If $J_{\min}$ is universal, then it is also a core solution for the data exchange scenario.

**Proof:** We first recall that a data exchange scenario is specified by $\mathcal{M}_d = \{S, T, \Sigma_{st}^d, \Sigma_t^d\}$, where $S$ and $T$ are the source and target schemas, $\Sigma_{st}^d$ is a set of standard s-t tgds, and $\Sigma_t^d$ is a set of standard target constraints that includes target tgds and target egds. For a data exchange scenario $\mathcal{M}_d$, we denote by $\mathcal{M}_d^{\text{map}}$ its associated mapping scenario specified by $\{S, S_a, T, \Sigma_t, \Sigma_e, \Pi, \text{User}\}$, where $\mathcal{S}_a = \emptyset$;
• \( \Sigma_t \) is the set of standard s-t tgds in \( \Sigma^d_{st} \) and the set of standard target tgds in \( \Sigma^d_t \);
• \( \Sigma_e \) is the set of standard egds in \( \Sigma^d_t \);
• \( \Pi \) and User are empty;
• the set of lluns, LLUNS, is also empty, i.e., we only allow for constants and labeled nulls in instances.

We write such mapping scenario’s as \( \{ S, T, \Sigma_t, \Sigma_e \} \) for short.

Conversely, given a mapping scenario \( M_{map} = \{ S, T, \Sigma_t, \Sigma_e \} \) satisfying the conditions above for some sets \( \Sigma^d_{st} \) and \( \Sigma^d_e \), we define its associated data exchange scenario as \( M^d_{map} = \{ S, T, \Sigma^d_{st}, \Sigma^d_e \} \).

Let \( I \) be an instance of \( S \) and \( J_{de} \) be an instance of \( T \) such that \( \langle I, J_{de} \rangle \) is a core solution for \( M_{de} \). We will show that \( J_{de} \) corresponds to a minimal solution \( Upd^d_{map} \) for \( M^d_{map} \) over \( \langle I, \emptyset \rangle \). Conversely, we show that a minimal solution \( Upd^d_{map} \) for \( M_{map} \) over \( \langle I, \emptyset \rangle \) corresponds to a core solution \( \langle I, J^d_{map} \rangle \) for \( M^d_{map} \).

In order to prove the claim, we will leverage two preliminary lemmas.

**Lemma 14** In the absence of authoritative sources, LLUNS, user input and partial order specification \( \Pi \), \( \text{Upd} \preceq \Pi, \text{User} \) \( \text{Upd}^d \) implies that there exists a homomorphism from \( \text{Upd}(J) \) to \( \text{Upd}^d(J) \).

**Proof of Lemma 14** Assume that \( \text{Upd} \preceq \Pi, \text{User} \text{Upd}^d \). This implies that there exists an id mapping \( h_{id} \) from tuple ids in \( \text{Upd} \) to tuples ids in \( \text{Upd}^d \) such that for each cell group \( g \in \text{Upd} \) there exists a cell group \( g' \in \text{Upd}^d \) such that \( h_{id}(g) \preceq \Pi, \text{User} g' \). We define a homomorphism \( h \) from \( \text{Upd}(J) \) to \( \text{Upd}^d(J) \), as follows. For constants \( c \in \text{dom} \text{(Upd}(J)) \) we let \( h(c) = c \). For labeled nulls \( N \in \text{dom} \text{(Upd}(J)) \) we define \( h(N) \) as the value of the cell group \( g' = \langle v \rightarrow \text{occ}', \text{just}' \rangle \) in \( \text{Upd}^d \) such that \( h_{id}(g) \preceq \Pi, \text{User} g' \). Observe that \( h \) is well-defined as \( g \) has a unique cell group \( g' \) such that \( h_{id}(g) \preceq \Pi, \text{User} g' \).

We show that \( h \) is indeed a homomorphism. Let \( t_i : R(a_1, ..., a_k, N_1, ..., N_l) \) be a tuple in \( \text{Upd}(J) \). We need to show that \( R(a_1, ..., a_k, h(N_1), ..., h(N_l)) \) is in \( \text{Upd}^d(J) \). We claim that the tuple \( h_{id}(t_i) \) is such a tuple.

First, we show that \( h_{id}(t_i) \) is a tuple containing the same constants as \( R(a_1, ..., a_k, N_1, ..., N_l) \). Indeed, this follows from the fact that the cell groups corresponding to the cells holding constant values in \( R(a_1, ..., a_k, N_1, ..., N_l) \) have counterparts in \( \text{Upd}^d \) that cover the corresponding cells in \( \text{Upd}(J) \) under \( h_{id} \). In this simplified setting, according to Definition 7, the value of these cell groups must be the same constant, since distinct constants are incomparable, and there are no lluns that can be used to generalize constants.

Let us now consider cells with a labeled null \( N_i \) in \( h_{id}(t_i) \). We now show that the values of these cells are exactly \( h(N_i), i = 1 \ldots l \). In fact, the original cells in \( t_i \) had value \( N_i \) and by construction, the corresponding cell groups in \( \text{Upd}^d \) have value \( h(N_i) \).

Based on this, we know that, given a tuple \( t_i : R(a_1, ..., a_k, N_1, ..., N_l) \) in \( \text{Upd}(J) \), its image according to \( h, \ R(a_1, ..., a_k, h(N_1), ..., h(N_l)) \), is a tuple \( h_{id}(t_i) \) in \( \text{Upd}^d(J) \) and therefore \( h \) is an homomorphism of \( \text{Upd}(J) \) into \( \text{Upd}^d(J) \). This proves the lemma.

The second lemma shows the relationship among solutions to a mapping & cleaning scenario and solutions to the corresponding data exchange scenario. Given a solution \( J \) for \( M_{de} \) and \( I \), we say that \( J \) is domain-based if any constant in \( \text{dom}(J) \) also appears in \( \text{dom}(I) \). In essence, domain-based solutions can only use constants from \( I \) and are an important subset of the solutions for \( M_{de} \) and \( I \), since they subsume the set of universal solutions.

Indeed, it is easy to see that any universal solution \( J_u \) for \( M_{de} \) and \( I \) must be domain-based. In fact, \( J_u \) needs to have homomorphisms into any other solution.

We can prove the following lemma:

**Lemma 15** Every solution of a mapping scenario corresponds to a domain-based solution of its associated data-exchange scenario, and vice versa.

**Proof of Lemma 15** We first show how domain-based solutions for \( M_{de} \) relate to solutions for \( M^d_{map} \), and then the converse.

(Part a.) Data Exchange Solution to Mapping & Cleaning Solution Given a domain-based solution \( J_{de} \) we need to construct an upgrade \( Upd^d_{de} \) of the initial target instance \( J \) (which is empty) and such that \( \langle I, Upd^d_{de}(J) \rangle \) satisfies after upgrades the cell groups of \( M^d_{de} \). We observe that in the absence of LLUNS no backward changes can be present in cell groups. Similarly, since User is empty no user input can be present in cell groups. As a consequence, all cell groups have empty meta-cells. We can thus safely represent cell groups in this setting simply by their occurrences and justifications.
We define \( \text{Upd}_{de}^{map} \) as follows:

(i) for each null value \( N \in \text{dom}(J_{de}) \), \( \text{Upd}_{de}^{map} \) contains the cell group \( g_N = \langle N \rightarrow \text{occ}_N, \emptyset \rangle \), where \( \text{occ}_N \) consists of all cells in \( J_{de} \) that have value \( N \);

(ii) furthermore, for each occurrence \( c = t_i.A_j \) of a constant value \( v \) in \( J_{de} \), we define \( g_{c,v} = \langle v \rightarrow \{ t_i.A_j \} \rangle \), by \( \{ t_k.A_l \} \), where \( t_k.A_l \) is a cell of the source instance, \( I \), such that \( \text{val}(t_k.A_l) = v \). Notice that cell \( t_k.A_l \) must exists, since we assume that \( J_{de} \) is domain-based. In case there are several cells with this property, we pick one randomly.

Observe that this update is valid since all cells in a cell groups carry the same value (either a constant or null). An update that only consists of these two types of cell groups will be referred to as an update in singleton form.

Recall that \( J = \emptyset \) and we can represent \( J \) by the trivial modification \( \text{Upd}_J \), which in this case is empty. The condition \( \text{Upd}_0 \leq \text{Upd}_{de}^{map} \leq \text{Upd}_{de} \) is then vacuously satisfied and it remains to be shown that \( \langle I, \text{Upd}_{de}^{map}(J) \rangle \) satisfies after upgrades \( \Sigma_{st} \) and \( \Sigma_e \) under \( \preceq \text{user} \). For this, it suffices to observe that the definition of satisfaction after upgrades incorporates the standard notion of satisfaction. Recall that \( \langle I, J_{de} \rangle \) is a solution for \( \mathcal{M}_{de} \) and thus \( \langle I, J_{de} \rangle \) satisfies \( \Sigma_t \) and \( \Sigma_e \) in the standard semantics. Furthermore, \( J_{de} = \text{Upd}_{de}^{map}(J) \) by construction. We may thus conclude that \( \langle I, \text{Upd}_{de}^{map}(J) \rangle \) also satisfies \( \Sigma_t \) and \( \Sigma_e \) in the standard sense, and thus also satisfies these after upgrades.

**Part b.) Mapping & Cleaning Solution to Data Exchange Solution** We next show that a solution \( \text{Upd}_{map} \) for \( \mathcal{M}_{map} \) over \( \langle I, J = \emptyset \rangle \) corresponds to a domain-based solution \( \langle I, J_{de}^{map} \rangle \) of \( \mathcal{M}_{de}^{map} \).

Let \( J_{map} = \text{Upd}_{map}(J) \), i.e., \( J_{map} \) is the set of tuples which are inserted in the initial empty target instance, as specified by \( \text{Upd}_{map} \). We claim that \( J_{map} \) is a solution for \( \mathcal{M}_{de}^{map} \). In other words, \( \langle I, J_{map} \rangle \) satisfies \( \Sigma_t \) and \( \Sigma_e \) under the standard semantics of first-order logic. In addition, we notice that \( J_{de} = \text{Upd}_{de}^{map}(J) \) by construction. We may thus conclude that \( \langle I, \text{Upd}_{de}^{map}(J) \rangle \) also satisfies \( \Sigma_t \) and \( \Sigma_e \) in the standard sense, and thus also satisfies these after upgrades.

**(i)** occurrences are exclusively new cells, i.e., cells with a value that was originally null;

**(ii)** justifications are sets of cells from \( I \).

As a consequence, if \( \text{val}(g) \) is a constant, this comes from one or more cells within the source instance \( I \). As a consequence, we claim that \( J_{map} \) is a domain-based solution.

Let us now prove that \( \langle I, J_{map} \rangle \) satisfies \( \Sigma_t \) and \( \Sigma_e \) under the standard semantics of first-order logic. By Lemma 4, we know that this is true for standard egds (that only contain target symbols) and target tgd. It remains to show that satisfaction after upgrades coincides with the standard notion of satisfaction for s-t tgd.

Consider an s-t tgd \( m : \forall \pi, \theta(\phi(\pi, \theta) \rightarrow \exists \varphi \psi(\pi, \varphi)) \in \Sigma_t \) and suppose that it is satisfied after upgrades. Clearly, if \( \langle I, J_{map} \rangle \) satisfies \( m \) under the standard semantics then nothing needs to be shown. Assume for the sake of contradiction \( m \) is not satisfied under the standard semantics, i.e., there exists a homomorphism \( h \) of \( \phi \) into \( \langle I, J_{map} \rangle \) that cannot be extended to a homomorphism \( h' \) of \( \psi \) into \( \langle I, J_{map} \rangle \). This implies, since \( m \) is satisfied after upgrades, that \( \text{Upd}_{can}^{h_{de}} \preceq \text{user} \) \( \text{Upd}_{map} \) holds. Here \( \text{Upd}_{can}^{h_{de}} \) refers to the canonical update associated with \( \text{Upd}_{de}^{map} \), \( m \) and \( h \). We show that in the context of the mapping scenario’s considered here, this still implies that \( m \) is satisfied under the standard semantics.

By Lemma 14, \( \text{Upd}_{can}^{h_{de}} \preceq \text{Upd}_{map} \) implies that there exists a homomorphism \( h'' : \text{Upd}_{can}^{h_{de}}(J) \rightarrow \text{Upd}_{map}(J) \). Consider the homomorphism \( h \) which maps \( \phi(\pi, \theta) \) into \( I \). We show that \( h \) can be extended to a homomorphism \( h' \) which maps \( \psi(\pi, \varphi) \) into \( J_{map} \). By definition, \( h_{can} \) extends \( h \) and maps \( \psi(\pi, \varphi) \) into \( \text{Upd}_{can}^{h_{de}}(J) \). Consider the mapping \( h'(x) = h(x) \), \( h'(z) = h(z) \) and \( h'(y) = h''(h_{can}(y)) \) for \( x \in \pi, y \in \theta \) and \( z \in \pi \).

Clearly, \( h'((\phi(\pi, \theta))) = h'((\phi(\pi, \theta))) \) and thus \( h' \) is an extension of \( h \). Consider \( h'((\psi(\pi, \varphi))) = h((\varphi), h''(h_{can}(\varphi))) \). We will now show that \( h'((\psi(\pi, \varphi))) \) can be written as \( h''(h_{can}((\psi(\pi, \varphi)))) \). In fact, we know that, for every \( x \in \pi, h(x) = h'(x) = h_{can}(x) = c \in \text{CONS} \). As a consequence, it is also the case that \( h(x) = h''(h_{can}(x)) = c \). Therefore, we have that:

\[
\begin{align*}
\psi(\pi, \varphi) &= h((\psi(\pi, \varphi)), h'(h_{can}(\varphi))) = h''(h_{can}(\psi(\pi, \varphi))) = h''(h_{can}(\psi(\pi, \varphi)))
\end{align*}
\]

Since \( h_{can} \) maps \( \psi(\pi, \varphi) \) into \( \text{Upd}_{can}^{h_{de}}(J) \), and \( h'' \) maps tuples in \( \text{Upd}_{can}^{h_{de}}(J) \) into tuples in \( \text{Upd}_{map}(J) \), we also have that \( h'((\psi(\pi, \varphi))) \subseteq J_{map} \). Indeed, the composition of two homomorphisms is again a homomorphism. In other words, \( h' \) is an extension of \( h \) that maps \( \psi(\pi, \varphi) \) into \( J_{map} \). This contradicts the fact that \( h \) could not be extended. That is, \( \langle I, J_{map} \rangle \models m \). This concludes the proof of the Lemma.

We now discuss the relationship of core solutions (for \( \mathcal{M}_{de} \)) and minimal solutions for \( \mathcal{M}_{de}^{map} \).

**Minimal Solution to Core Solution** We first show that minimal solutions of \( \mathcal{M}_{map} \) that generate universal solutions correspond to core solutions of the data exchange scenario \( \mathcal{M}_{de}^{map} \). Let \( \text{Upd}_{min} \) be a minimal solution of \( \mathcal{M}_{map} \). Let \( J_{min} = \text{Upd}_{min}(J = \emptyset) \) be the corresponding target instance. We have previously shown that \( J_{min} \) is a (domain-based) solution of \( \mathcal{M}_{de}^{map} \). Let us assume that \( J_{min} \) is universal.
Assume, for the sake of contradiction, that \( J_{min} \) is not a core. Since \( J_{min} \) is universal, there exists a proper sub-instance \( J' \subseteq J_{min} \) and \( J' \) is the core solution for \( \mathcal{M}_{map} \) and \( I \). Furthermore, there exists a homomorphism \( h' : J_{min} \rightarrow J' \).

We next define an update \( \text{Upd}' \) such that \( \text{Upd}'(\emptyset) = J' \) and furthermore, \( \text{Upd}' \preceq_{\text{nl-user}} \text{Upd}_{\text{min}} \). More precisely, consider the non-empty set of tuples \( \Delta = J_{min} - J' \). Update \( \text{Upd}' \) is obtained from \( \text{Upd}_{\text{min}} \) by changing its cell groups in such a way to remove any occurrence with a tuple id in \( \Delta \). More precisely:

- from any cell group in \( \text{Upd}_{\text{min}} \), we remove occurrences whose tuple ids appear in \( \Delta \);
- at the end of the process, we remove cell groups with empty occurrences.

It is easy to see that \( \text{Upd}' \) is still a solution to the mapping and cleaning scenario. In fact, \( \text{Upd}'(\emptyset) = J' \), since any cell with a tuple id in \( J' \) has exactly the same value it had in \( \text{Upd}_{\text{min}} \), and we know that \( J' \) is a solution to the data exchange scenario.

In addition, \( \text{Upd}' \preceq_{\text{nl-user}} \text{Upd}_{\text{min}} \). Indeed, consider the id mapping \( h_{id} \) that is the identity on the tuples of \( J' \). For any cell group \( g \in \text{Upd}' \), there exists a cell group \( g' \in \text{Upd}_{\text{min}} \) that has at least the same occurrences, equal justifications (if any), and equal value. Therefore \( g \preceq_{\text{nl-user}} g' \).

However, we can show also that \( \text{Upd}' \preceq_{\text{nl-user}} \text{Upd}_{\text{min}} \). Indeed, consider the id mapping \( h_{id} \) that is the identity on the tuples of \( J' \). For any cell group \( g \in \text{Upd}' \), there exists a cell group \( g' \in \text{Upd}_{\text{min}} \) that has at least the same occurrences, equal justifications (if any), and equal value. Therefore \( g \preceq_{\text{nl-user}} g' \).

For case (a) we may assume that \( \text{Upd}' \) is minimal. Since we know that \( \text{Upd}' \preceq_{\text{nl-user}} \text{Upd}'_{\text{core}} \), by Lemma 14, there exists a homomorphism \( h \) of \( \text{Upd}'(\emptyset) \) into \( \text{Upd}'_{\text{core}}(\emptyset) = J_{core} \). As a consequence, \( J' = \text{Upd}'(\emptyset) \) is a universal solution. Thus, it is also a core solution, as we have just shown. This implies that \( J' \) and \( J_{core} \) are two core solutions and thus isomorphic. Recall that by construction, \( \text{Upd}'_{\text{core}} \) is an update in singleton form. Since \( \text{Upd}' \preceq_{\text{nl-user}} \text{Upd}'_{\text{core}} \), we may further assume that \( \text{Upd}' \) is also in singleton form.

Let \( h : J_{core} \rightarrow J' \) be an isomorphism and define an id mapping, \( h_{id} \), such that \( h_{id}(t.tid) = h(t).tid \). We will now show that \( \text{Upd}_{\text{core}} \preceq_{\text{nl-user}} \text{Upd}' \) according to \( h_{id} \), which is in contradiction with the hypothesis in case (a).

Let \( v \) be a constant in \( \text{dom}(J_{core}) \) occurring in cell \( c = t_{i}.A_{j} \). Then \( g_{v,c} \in \text{Upd}_{\text{core}} \) and clearly \( h_{id}(g_{v,c}) = g_{v,h_{id}(c)} \) where \( h_{id}(c) \) is a single cell as well (because of the isomorphism). Since \( J' \) and \( J_{core} \) are isomorphic, there must exists a cell group \( g' \in \text{Upd}' \) such that \( g_{v,h_{id}(c)} \preceq_{\text{nl-user}} g' \), since \( h \) maps \( v \) to the same value in the corresponding cell in \( J' \).

Let \( N \) be a null value in \( \text{dom}(J_{core}) \). Consider \( g_{N} \in \text{Upd}_{\text{core}} \). We have that \( h_{id}(g_{N}) = (N \rightarrow h_{id}(occ_{N}), by \emptyset) \) and \( h_{id}(occ_{N}) \) consists of as many cells as \( occ_{N} \) (because of the isomorphism). Since \( h \) maps \( N \) to a (possibly different) null value \( N' \) in \( J' \), and because there is only one cell group in \( \text{Upd}_{\text{core}} \) with that null value (by definition of update), there must be a cell group \( g' = (N' \rightarrow occ_{N'}, by \emptyset) \) in \( \text{Upd}' \) such that \( h_{id}(occ_{N'}) \subseteq occ_{N} \). In other words, \( \text{Upd}_{\text{core}} \preceq_{\text{nl-user}} \text{Upd}' \) which contradicts our assumption. Thus, case (a) cannot occur.

For case (b) observe that we have just shown that \( \text{Upd}' \preceq_{\text{nl-user}} \text{Upd}_{\text{core}} \) implies that \( \text{Upd}_{\text{core}} \preceq_{\text{nl-user}} \text{Upd}' \) and furthermore, \( \text{Upd}_{\text{core}}(J) \) and \( \text{Upd}'(J) \) are isomorphic. Consider \( \text{Upd}'' \) obtained from \( \text{Upd}' \) by relabeling nulls such that \( \text{Upd}''(J) = \text{Upd}_{\text{core}}(J) \). Suppose, for the sake of contradiction, that there is an id mapping \( h_{id} \) from \( \text{Upd}' \) to \( \text{Upd}_{\text{core}} \) that is not surjective. Then, \( h_{id} \) can also be used to map \( \text{Upd}'' \) to \( \text{Upd}_{\text{core}} \). Denote this mapping by \( h_{id}'' \). By definition, \( h_{id}'' \) is the identity because \( \text{Upd}''(J) = \text{Upd}_{\text{core}}(J) \) and since it is non-surjective, this implies that we have that \( \text{Upd}''(J) \) is a proper subset of \( J_{core} \). Furthermore, we also have that \( \text{Upd}_{\text{core}} \preceq_{\text{nl-user}} \text{Upd}'' \) and thus there is a homomorphism from \( J_{core} \) to a strict subset of \( J_{core} \). This contradicts the assumption that \( J_{core} \) is a core solution.

Hence, \( \text{Upd}_{\text{core}} \) is indeed a minimal solution of \( \mathcal{M}_{de} \) and the claim is proven. \( \square \)
THEOREM 6 Given a cleaning scenario CS = \{⟨S , T⟩, Σe , Π⟩ and an input instance ⟨I , J⟩, there always exists a solution for CS and ⟨I , J⟩.

Proof: Indeed, there is always a solution corresponding to the update that changes all cells of J together, and justifies it by all cells in I , with both metacells i.e., Upd_{top} = \{⟨cells(J) , cells(I) , \{c_x , c_T⟩}⟩

In fact, Upd_{top} is an upgrade of the initial target instance, J. In addition, it is fairly easy to see that this update satisfies after repairs any egd e ∈ Σ_e. In fact, assume the conclusion of e is of the form x_i = x_j. Then:

(i) either x_i, x_j have only target occurrences within the premise of e; in this case, e will be vacuously satisfied in the standard sense in Upd_{top}(J), since all cells will have the same value;

(ii) or one of x_i, x_j (or both) has both source and target occurrences within the premise of e; however, in this case, e will be satisfied after upgrades by Upd_{top}, since for any homomorphism h of φ(\bar{x}) into Upd_{top}(J), g_h(x_i) and g_h(x_j) are the same cell group g_{top} = \{cells(J) , cells(I) , \{c_x , c_T⟩}⟩

(iii) or one of x_i, x_j (say x_j) has only source occurrences (i.e., it was initially a constant e that was encoded as the associated special authoritative cell); also in this case, e is satisfied after upgrades by Upd_{top}, since, for any homomorphism h of φ(\bar{x}) into Upd_{top}(J), g_h(x_i) corresponds to g_{top} = \{cells(J) , cells(I) , \{c_x , c_T⟩}⟩ and this includes all justifications, including the authoritative cell associated with e in e.

We make the straightforward assumption that this solution is never refused by the user function, User.

THEOREM 7 Given two solutions Upd , Upd’ for a scenario CS over instance ⟨I , J⟩, one can check Upd \preceq_{\Pi} Upd’ in O(n + km\log(m)) time, where n is the number of cells in J , k is the maximum number of cell groups in Upd , Upd’ , and m is the maximum size of a cell group in Upd , Upd’. 

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

THEOREM 8 Given a mapping & cleaning MC = \{S , S_o , T , Σ_t , Σ_e , Π, User⟩, instances I of S ∪ S_o and J of T, and oracle User, the chase of ⟨I , J⟩ with Σ_t , Σ_e , User may not terminate after a finite number of steps. If it terminates, it generates a finite set of results, each of which is a solution for MC over ⟨I , J⟩. Even if the chase terminates, not every minimal solution is generated.

Proof: The existence of non-terminating mapping & cleaning scenarios is an immediate consequence of the fact that these are an extension of mapping scenarios for which such non-terminating examples are known [14]. If the chase terminates, then by the definition of chase this means that no chase rule is applicable, which in turns means that all tgds and egds are either satisfied (in the standard sense) or are satisfied after upgrades.

We now show a cleaning scenario for which the chase does not generate all minimal solutions. Consider R(A,B,C) with A → B, and the following instance J = \{t_1 : (1 , a , x) , t_2 : (1 , b , y) , t_3 : (2 , b , z)⟩. Assume that for the A attribute the partial order specification states that higher values are always preferable. The chase generates the following solutions, all of them minimal:

Upd_1 = \{g_1 = \{L_1 → \{t_1.B , t_2.B\}\}\}
Upd_2 = \{g_2 = \{L_2 → \{t_1.A , \{c_x\}\}\}\}
Upd_3 = \{g_3 = \{L_3 → \{t_2.A , \{c_x\}\}\}\}

There is, however, a fourth minimal solution, as follow:

Upd_4 = \{g_4 = \{2 → \{t_2.A , t_3.A\}\}\}

This yields the following instance: Upd_4(J) = \{t_1 : (1 , a , x) , t_2 : (2 , b , y) , t_3 : (2 , b , z)⟩, that satisfies the constraint, and is incomparable to any of the other solutions. In essence, Upd_4 improves the database by changing cell t_2.A to a “better value” taken from t_1.A, a change that the chase never does.

For soundness recall that, according to the Definition 12, a solution for MC is an update Upd s.t. (i) J \preceq_{\Pi, User} Upd and (ii) ⟨I , Upd(J)⟩ satisfies after upgrades Σ_t ∪ Σ_e under \preceq_{\Pi, User}. Regarding the point (ii) we know, by definition, that...
every leaf is a valid update \( \text{Upd}_i \), such that there is no dependency or user inputs applicable to \((I, \text{Upd}_i(J))\). For \(i\) it suffices to observe that for each path \( J, \text{Upd}_i(J) \), in the chase tree from the root, \( J \) to a leaf, \( \text{Upd}_i(J) \), the following condition holds: \( J \implies \text{Upd}_i(J) \implies \text{Upd}_i(J) \implies \ldots \implies \text{Upd}_i(J) \). This is due to the fact that going from \( \text{Upd}_i \) to \( \text{Upd}_{i+1} \) cell groups are changed in four possible ways:

(a) \( \text{Upd}_{i+1} \) is the result of a chase step for user inputs, where User applies to a cell group \( g \in \text{Upd}_i \), and \( \text{Upd}_{i+1} \) contains a cell group \( g'_{\text{User}} \) with the same occurrences and justifications, with the addition of the user-cell \( c_T \). Clearly \( g \implies g', g'_{\text{User}} \), and since \( \text{Upd}_{i+1} \) differs only on \( g, \text{Upd}_i \subseteq \text{Upd}_{i+1} \); 

(b) \( \text{Upd}_{i+1} \) is the result of a chase step for a tgd, where an extended tgd \( m \) applies to \( \text{Upd}_i \), and \( \text{Upd}_{i+1} \) is the canonical update for \( m \). \( \text{Upd}_{i+1} \) contains new cell groups for existential variables in \( m \), and it changes cell groups for universal variables adding new occurrences and possibly new justifications. So \( \text{Upd}_i \subseteq \text{Upd}_{i+1} \); 

(c) it is the case that an \( \text{Upd}_{i+1} \) is the result of a forward chase step of an extended egd \( e \) on \( \text{Upd}_i \). \( \text{Upd}_{i+1} \) is obtained from \( \text{Upd}_i \) by taking the union of the cell groups for variables \( x \) and \( x' \). Since the new cell group satisfies the cell-containment properties with both \( g(x) \) and \( g(x') \), \( \text{Upd}_i \subseteq \text{Upd}_{i+1} \); 

(d) finally it is the case that an \( \text{Upd}_{i+1} \) is the result of a backward chase step of an extended egd \( e \) on \( \text{Upd}_i \). It is obtained by changing a cell group \( g_{ij} \) in \( \text{Upd}_i \), to another cell group \( g'_{ij} \), that has same occurrences and justifications, with the addition of the invalid cell. So \( \text{Upd}_i \subseteq \text{Upd}_{i+1} \).

\[ \square \]

**Theorem 9** Given a mapping & cleaning \( \mathcal{MC} = \{S, S_u, T, \Sigma_t, \Sigma_e, \Pi, \text{User}\} \), instances \( I \) of \( S \cup S_u \), and \( J \) of \( T \), and oracle \( \text{User} \), if \( \Sigma_t \) is a set of weakly acyclic tgd's, then the chase of \((I, J)\) with \( \Sigma_t, \Sigma_e, \text{User} \) terminates after a finite number of steps, and each leaf in the chase tree is a solution for \( \mathcal{MC} \).

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

Consider a tgd \( m : \phi(x) \rightarrow \exists y : \psi(x, y) \in \Sigma_t \). Given \((I, J)\), we define the premise tuples, PREM-TUPLES\((m, \langle I, J \rangle)\) for \( m \) over \((I, J)\) as the set of all tuple ids in \((I, J)\) for atoms that appear in \( \phi(x) \). Let us call \( n \) the size of PREM-TUPLES\((m, \langle I, J \rangle)\).

It is easy to see that, when chasing \((I, J)\) with \( m \), \( m \) can be fired for a number of times that is bounded by a function of \( n \). In fact, any homomorphism \( h \) for which \( m \) can be fired needs to map \( \phi(x) \) into a distinct combination of tuples from PREM-TUPLES\((m, \langle I, J \rangle)\).

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

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

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

**Base case:** tgd's in the first stratum can only fire once for any homomorphism \( h \) of \( \phi(x) \) into PREM-TUPLES\((m, \langle I, K \rangle)\); in fact, since the tgd's are non-recursive, during the chase no other tgd adds tuples to the relations in \( \phi(x) \). Similarly for egd's, that may change the cell groups, but do not add new tuples, neither break homomorphisms for which \( m \) has been already fired. Therefore, the tgd stops firing after a finite number of steps.

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

For soundness, see the proof of Theorem 8.

\[ \square \]

**Theorem 10** Given a data exchange scenario \( \mathcal{M}_{de} \), the corresponding cleaning scenario \( \mathcal{M}_{de}^{\text{map}} \) is such that any solution generated by the chase of \((I, J)\) is a universal solution to \( \mathcal{M}_{de} \) and \((I, J)\).

**Proof:** The proof is based on this observations:

(i) in the data exchange setting, the only chase steps are the ones for tgd's, and the ones for egd's that use forward changes remove nulls in favor of other nulls or constants;

49
(ii) at any chase step \( \text{Upd} \rightarrow_{d,h} \text{Upd}_i \), consider instances \( \text{Upd}(J = \emptyset) \) and \( \text{Upd}_i(J = \emptyset) \): it is easy to see that \( \text{Upd}_i(\emptyset) \) is exactly the instance that is generated by the standard chase of data exchange [14] over \( \text{Upd}(\emptyset) \) with \( d \) and \( h \).

\[ \text{Theorem 11} \quad \text{Given a cleaning scenario } \text{CS} = \{ S, S_a, T, \emptyset, \Sigma_e, \Pi, \text{User} \} \text{ and an instance } \langle I, J \rangle, \text{ the chase of } \langle I, J \rangle \text{ with } \Sigma_e \text{ (i) terminates; (ii) it generates a finite set of results, each of which is a solution for } \text{CS} \text{ over } \langle I, J \rangle. \]

Proof: For termination, it suffices to observe that for each path \( J, \text{Upd}_{i_0}, \ldots, \text{Upd}_{i_k} \) in the chase tree from the root, \( J \) to a leaf, \( \text{Upd}_{i_k} \), the following condition holds \( J \subseteq \text{Upd}_{i_0}(J) \subseteq \text{Upd}_{i_1}(J) \subseteq \cdots \subseteq \text{Upd}_{i_k}(J) \). This is due to the fact that, going from \( i_0 \) to \( i_{k+1} \) by a chase step, cell groups grow monotonically (either the set of occurrences/justifications grows due to a forward step, or the value is upgraded due to a backward step). Due to our semantics, there is a topmost element (the top update \( \text{Upd}_{\text{top}} = \{ \text{cells}(J), \text{cells}(I), \{ e_x, c_T \} \} \)). Hence, every path in the chase tree is bounded in length as, in the worst case, it reaches \( \text{Upd}_{\text{top}} \). For soundness, see the proof of Theorem 9.

\[ \text{Theorem 12} \quad \text{Given a cleaning scenario } \text{CS} = \{ S, S_a, T, \emptyset, \Sigma_e, \Pi, \text{User} \} \text{ and an instance } \langle I, J \rangle, \text{ CS may have at most an exponential number of solutions over } \langle I, J \rangle, \text{ and each solution is computed in a number of steps that is polynomial in the size of } \langle I, J \rangle. \]

Proof: In general, it is readily verified that a cleaning scenario can have at most an exponential number of solutions. When considering the disjunctive chase procedure, as outlined above, one can verify that each solution is computed in a number of steps that is polynomial in the size of the data. For this, it suffices to observe that one can associate an integer-valued function \( f \) on updates such that \( f(\text{Upd}) < f(\text{Upd}') \) whenever \( \text{Upd} \rightarrow_{e, H} \text{Upd}' \) during the chase. Intuitively, \( f \) depends on the number of \( \text{lln} \) values and sizes of cell groups in the updates. Since both the number of \( \text{llns} \) and size of cell groups is bounded by the input instance, we may infer that \( f \) cannot be increased further after polynomially many steps, i.e., when a solution is obtained.

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

\[ \text{Theorem 13} \quad \text{Consider the chase tree } \text{chase}_{\Sigma_i, \Sigma_e, \text{User}}(\langle I, J \rangle), \text{ generated by the chase of } \text{MC} \text{ over } \langle I, J \rangle \text{ as defined in Section 14. If the chase of } \langle I, J \rangle \text{ with } \Sigma_i, \Sigma_e, \text{User terminates, then the revised chase of } \langle I, J \rangle \text{ with } \Sigma_i, \Sigma_e, \text{User also terminates. In this case, the revised chase procedure generates a Chase tree } \text{chase}_{\Sigma_i, \Sigma_e, \text{User}}(\langle I, J \rangle) \text{ such that for any node in } \text{chase}_{\Sigma_i, \Sigma_e, \text{User}}(\langle I, J \rangle), \text{ there is an identical node in } \text{chase}_{\Sigma_i, \Sigma_e, \text{User}}(\langle I, J \rangle). \]

Proof: The proof of the first part is the same as the proof of Theorem 8.

To prove the rest, we proceed by induction on the level of nodes in the chase tree, and show that for any node \( \text{Upd} \) in \( \text{chase}_{\Sigma_i, \Sigma_e, \text{User}}(\langle I, J \rangle) \), there is an identical node \( \text{Upd}_{\text{rev}} \) in \( \text{chase}_{\Sigma_i, \Sigma_e, \text{User}}(\langle I, J \rangle) \).

\[ \text{Base case: the root node in both chase trees is the empty repair } \langle I, J \rangle; \]

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

If \( \text{Upd}_n \) is the result of a chase step for a tgd or a user input we have nothing to prove, because the revised chase differs from the standard one only on the definition of a chase step for egds.

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

Since \( \text{Upd}_{n-1}^{\text{rev}} \) is equivalent to \( \text{Upd}_{n-1} \), the same homomorphism \( h \) of \( e \) is applicable to \( \text{Upd}_{n-1}^{\text{rev}} \). Let’s call \( H \) the homomorphism class for \( \text{Upd}_{n-1}^{\text{rev}} \) and \( e \) that contains \( h \). We know that \( H \) generates a violation for \( \text{Upd}_{n-1}(J) \) and \( e \), because it contains in the set of conclusion groups c-groups \( H \), the cell groups \( g_h(x) \) and \( g_h(x') \), that we know to be different. Recall that, by definition 28, the revised chase step over for \( \text{Upd}_{n-1}^{\text{rev}} \) and \( e \) will generate a new update for each valid repair strategy \( r s_H \) of \( H \). We need to prove that there exists a repair strategy that applied to \( \text{Upd}_{n-1}^{\text{rev}} \) generates an update \( \text{Upd}_n^{\text{rev}} \) identical to \( \text{Upd}_n \). We need to distinguish two cases:
• \( \text{Upd}_n \) is the result of a forward chase, then by definition is

\[
\text{Upd}_n = \text{Upd}_{n-1} - \{ g_{ij}(x), g_{ij}(x') \} \cup \text{Union}(g_{ij}(x), g_{ij}(x')).
\]

In order to generate the same update, consider a repair strategy \( rs^f_H \) for \( \mathcal{H} \) that maps any cell group in \( \mathcal{c}-\text{groups}_H \) to “unaffected”, except \( g_{ij}(x) \) and \( g_{ij}(x') \) that are marked as “forward”. The chase of the chase step strategy \( css_f : \{ e, \mathcal{H}, rs^f_H \} \) over \( \text{Upd}_{n-1}^{rev} \) generates the following update:

\[
\text{Upd}_{n-1}^{rev} = \text{Upd}_{n-1} - \text{forw-}g_{rs^f_H} \cup \text{Union}(\text{forw-}g_{rs^f_H}).
\]

Since \( \text{Upd}_{n-1} \) and \( \text{Upd}_{n-1}^{rev} \) are identical, and \( \text{forw-}g_{rs^f_H} = \{ g_{ij}(x), g_{ij}(x') \} \), the resulting update \( \text{Upd}_{n}^{rev} \) is identical to \( \text{Upd}_n \).

• \( \text{Upd}_n \) is the result of a backward chase, then it is defined as

\[
\text{Upd}_n = \text{Upd}_{n-1} - \{ g_{ij} \} \cup \{ g'_{ij} \}
\]

where \( g_{ij} \) is the cell group of the cell \( c_j \in \text{cells}_H(x) \) in \( \text{Upd}_{n-1} \), and \( x_i \) is a witness variable in \( e \). We know, by definition 21, that \( \text{val}(g_{ij}) \in \text{CONSTs} \) and \( \text{auth-cells}(g_{ij}) = \emptyset \).

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

\[\square\]

## C Experimental Settings

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

### C.1 Hospital-Norm

The first dataset is Hospital-Norm, the normalized version of the hospital data, of which we considered 3 tables with 2 foreign keys, a total of 20 attributes, and approximately 150K tuples. We generated instances of size up to 1M tuples by replicating the original data several times.

Schema:

\[
\text{Hosp}(\text{ProviderNumber}, \text{HospName}, \text{Addr1}, \text{Addr2}, \text{Addr3}, \text{ZipCode}, \text{CountyName}, \text{PhoneNumber})
\]

\[
\text{Zip}(\text{ZipCode}, \text{City}, \text{State})
\]

\[
\text{Meas}(\text{ProviderNumber}, \text{Diagnosis}, \text{Cases}, \text{Footnote}, \text{Mid})
\]

Target tgds:

\[
m_{t1}. \text{Hosp}(\text{pn}, \text{hn}, \text{a1, a2, a3, zc, cn, ph}) \rightarrow \exists Y_0, Y_1 : \text{Zip}(\text{zc}, Y_0, Y_1)
\]

\[
m_{t2}. \text{Hosp}(\text{pn}, \text{hn}, \text{a1, a2, a3, zc, cn, ph}) \rightarrow \exists Y_0, Y_1, Y_2, Y_3 : \text{Meas}(\text{pn,Y0,Y1,Y2,Y3})
\]

Target cleaning egds:

\[
e_{t1}. \text{Hosp}(\text{pn}, \text{hn}, \text{a1, a2, a3, zc, cn, ph}), \text{Hosp}(\text{pn}, \text{hn'}, \text{a1', a2', a3', zc', cn', ph'}) \rightarrow \text{hn} = \text{hn'}, \text{a1} = \text{a1'}, \text{a2} = \text{a2'}, \text{a3} = \text{a3'}, \text{zc} = \text{zc'}, \text{cn} = \text{cn'}, \text{ph} = \text{ph'}
\]

\[
e_{t2}. \text{Zip}(\text{zc}, \text{ci, st}), \text{Zip}(\text{zc}, \text{ci', st'}) \rightarrow \text{st} = \text{st'}
\]

\[
e_{t3}. \text{Meas}(\text{pn, di, ca, fn, mi}), \text{Meas}(\text{pn', di', ca', fn', mi}) \rightarrow \text{pn} = \text{pn'}, \text{di} = \text{di'}, \text{ca} = \text{ca'}, \text{fn} = \text{fn'}
\]
C.2 Hospital-Den

Hospital-Den is a highly denormalized version of the same data, with 100K tuples and 19 attributes. This version has been used to test data cleaning algorithms that were restricted to single-table databases. We generated instances of size up to 1M tuples by replicating the original data.

Schema:

```
Hosp(ProviderNumber, HospitalName, Addr1, Addr2, Addr3, Zip, County, Phone, City, State,
Type, Owner, Emergency, Condition, MsCode, MsName, Score, Sample, StAvg)
```

Target cleaning egds:

```
e_1. Hosp(p, n, a1, a2, a3, z, c, h, i, s, t, o, e, d, mc, mn, x, l, v),
    Hosp(p', n', a1', a2', a3', z', c', h', i', s', t', o', e', d', mc', mn', x', l', v') \rightarrow i = i'
```

```
e_2. Hosp(p, n, a1, a2, a3, z, c, h, i, s, t, o, e, d, mc, mn, x, l, v),
    Hosp(p', n', a1', a2', a3', z', c', h', i', s', t', o', e', d', mc', mn', x', l', v') \rightarrow z = z', i = i'
```

```
e_3. Hosp(p, n, a1, a2, a3, z, c, h, i, s, t, o, e, d, mc, mn, x, l, v),
    Hosp(p', n', a1', a2', a3', z', c', h', i', s', t', o', e', d', mc', mn', x', l', v') \rightarrow mn = mn', d = d'
```

```
e_4. Hosp(p, n, a1, a2, a3, z, c, h, i, s, t, o, e, d, mc, mn, x, l, v),
    Hosp(p', n', a1', a2', a3', z', c', h', i', s', t', o', e', d', mc', mn', x', l', v') \rightarrow v = v'
```

```
e_5. Hosp(p, n, a1, a2, a3, z, c, h, i, s, t, o, e, d, mc, mn, x, l, v),
    Hosp(p', n, a1', a2', a3', z', c', h', i', s', t', o', e', d', mc', mn', x', l', v') \rightarrow a1 = a1', a2 = a2', t = t'
```

C.3 Customers

Customers is the third dataset and corresponds to our running example in Figure 1. The database schema contains 3 tables with 4 to 9 attributes, plus 1 additional master data table with 5 attributes and 2 additional tables encoding constants in CFDs. We synthetically generated up to 1M tuples for the 4 source tables, with a proportion of 40% in MedTreatments, 40% in Surgeries, and 20% in Patients; the master-data table contains a few hundreds of the tuples present in Patients and in MedTreatments. For the target, we generated up to 1M tuples, with a proportion of 40% in the Customers table, and 60% in Treatments. Dependencies are the ones in Section 1.

Schema:

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

Source to target tgd:

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

```
m_2. MedTreat(ssn, name, phn, str, city, ins, treat, date, conf)
    \rightarrow \exists Y_3, Y_4 : Cust(ssn, name, phn, conf, str, city, Y_3), Treat(ssn, Y_4, ins, treat, date)
```

Target tgd:

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

Target cleaning egds:

```
e_1. Cust(ss, n, p, s, c, cc), Cust(ss, n, p', s', c', cc') \rightarrow p = p'
```

```
e_2. Cust(ss, n, p, s, c, cc), Cust(ss, n, p', s', c', cc') \rightarrow cc = cc'
```

```
e_3. Cust(ss, n, p, s, c, cc), Cust(ss', n, p, s, c, cc') \rightarrow ss = ss'
```

```
e_4. Treat(ssn, s, ins, tr, d), Cst_{tr}(ins, tr') \rightarrow tr = tr'
```

```
e_5. Cust(ssn, n, p, s, c, cc), MD(ssn, n', p, s', c') \rightarrow n = n'
```

```
e_6. Cust(ssn, n, p, s, c, cc), MD(ssn, n', p, s', c') \rightarrow s = s'
```

```
e_7. Cust(ssn, n, p, s, c, cc), MD(ssn, n', p, s', c') \rightarrow c = c'
```

```
e_8. Treat(ssn, n, p, s, c, cc), Treat(ssn, sal, ins, tr, d), Cst_{ps}(ins, c') \rightarrow c = c'
```

```
e_9. Treat(ssn, s, ins, tr, d), Treat(ssn, s', ins', tr', d') \rightarrow s = s'
```

52