A Unified Model for Data and Constraint Repair

Fei Chiang, Renée J. Miller

Department of Computer Science, University of Toronto
Toronto, Canada
{fchiang, miller}@cs.toronto.edu

***Abstract*—Integrity constraints play an important role in data design. However, in an operational database, they may not be enforced for many reasons. Hence, over time, data may become inconsistent with respect to the constraints. To manage this, several approaches have proposed techniques to repair the data, by finding minimal or lowest cost changes to the data that make it consistent with the constraints. Such techniques are appropriate for the old world where data changes, but schemas and their constraints remain fixed. In many modern applications however, constraints may evolve over time as application or business rules change, as data is integrated with new data sources, or as the underlying semantics of the data evolves. In such settings, when an inconsistency occurs, it is no longer clear if there is an error in the data (and the data should be repaired), or if the constraints have evolved (and the constraints should be repaired). In this work, we present a novel unified cost model that allows data and constraint repairs to be compared on an equal footing. We consider repairs over a database that is inconsistent with respect to a set of rules, modeled as functional dependencies (FDs). FDs are the most common type of constraint, and are known to play an important role in maintaining data quality. We evaluate the quality and scalability of our repair algorithms over synthetic data and present a qualitative case study using a well-known real dataset. The results show that our repair algorithms not only scale well for large datasets, but are able to accurately capture and correct inconsistencies, and accurately decide when a data repair versus a constraint repair is best.**

I. INTRODUCTION

Integrity constraints are the primary means for preserving data integrity. Constraints represent domain specific rules and relationships that hold over any database instance that accurately reflects the domain. Typically, constraints are defined at design time when a data architect with domain knowledge precisely defines the semantics of the application data.

If every constraint is enforced within a database, then the data, as it evolves, will continue to conform to the constraints. In reality however, constraints may not be enforced (often for performance reasons). Relaxed enforcement policies may allow the data to become inconsistent with respect to the constraints. To manage this, several approaches have proposed techniques to repair the data, by finding minimal or lowest cost changes to the data that make it consistent with the constraints [1], [2], [3]. Such techniques are appropriate for the old world where data changes, but schemas and their constraints remain fixed.

In many modern applications however, constraints may evolve over time as application or business rules change, as data is integrated with new data sources, or as the underlying semantics of the data evolves. In such settings, when an inconsistency occurs, it is no longer clear if there is an error in the data (and the data should be repaired), or if the constraints have evolved (and the constraints should be repaired). More precisely, if the data values are incorrect, then the data values should be corrected. If the constraints are incorrect, then they should be corrected to accurately (but concisely) model the data. Although constraint discovery techniques can be applied, they are well-known to find ”accidental” constraints, i.e., constraints that happen to hold over an instance, but not application constraints that hold over time. In addition, constraint discovery techniques [4], [5], [6] are expensive to compute and they do not consider in their search the existing set of constraints. Hence, they may discover a new and very different set of constraints. We propose an alternative approach that considers both data and the existing constraints in the repair process. Hence, discovered constraints are modifications of known constraints that have already been validated as application constraints. For both constraint and data repairs, we will want to make the lowest cost change possible to bring the data and constraints back into a consistent state.

Sources of data errors are well-known (careless data entry practices, unnormalized data containing redundancies, integration of data sources, etc. [7]). Sources of constraint errors include evolution of domain or application requirements and of course integration as well. For example, each university department may maintain its own courses. If the university decides to consolidate all courses to a central table, ambiguities among department courses can arise. That is, Classical Studies (CS 2400) vs. Computer Science (CS 2400) will be difficult to decipher without a distinguishing attribute such as the originating DEPT. Constraints such as F : [CourseID] $\rightarrow$ [ROOM, DAY, TIME] may no longer hold over the integrated table. A possible constraint repair would be to add DEPT to F's antecedent to resolve the inconsistency. Notice that if we only consider data repairs, we would have to modify all courses with the same CourseID which may not be the desired semantics. If we consider only constraint discovery [4], [8], we might find numerous extraneous constraints such as E : [ROOM,DAY,TIME,TEXT] $\rightarrow$ [CourseID], a rule that may hold simply because of the large domains of attributes like TEXT, but does not reflect any application constraint. Constraints may also be violated if they are unable to handle correct but exceptional cases, if they are not updated as data and policies evolve, or if there is poor management of the constraints. The Belgian social security agency [9], which handles social security contributions from employers

TABLE I: Example table

tid	District	Region	Municipal	AreaCode	PhNo	Street	Zip	City	State
t_1	Brookside	Granville	Glendale	613	974-2345	Boxwood	10211	NY	NY
t_2	Brookside	Granville	Glendale	613	974-2345	Boxwood	10211	NY	NY
t_3	Brookside	Granville	Glendale	613	299-1010	Westlane	10211	NY	MA
t_4	Brookside	Granville	Guildwood	515	220-1200	Squire	02215	Boston	MA
t_5	Brookside	Granville	Guildwood	515	220-1200	Squire	02215	Boston	MA
t_6	Alexandria	Moore Park	NapaHill	415	220-1200	Napa	60415	Chicago	IL
t_7	Alexandria	Moore Park	NapaHill	415	930-2525	Main	60415	Chicago	IL
t_8	Alexandria	Moore Park	NapaHill	415	555-1234	Tower	60415	Chester	IL
t_9	Alexandria	Moore Park	QueenAnne	517	888-5152	Main	60415	Chicago	IL
t_{10}	Alexandria	Moore Park	QueenAnne	517	888-5152	Main	60601	Chicago	IL
t_{11}	Alexandria	Moore Park	QueenAnne	517	888-5152	Bay	60601	Chicago	IL

and workers, faced similar issues in trying to distribute benefits. For example, workers may work both weekdays and weekends leading to a higher number of days worked than the constraints allowed, and changes to the contribution rates were not reflected in the constraints due to costly updates.

Consider the following example that illustrates the power of considering both data and constraints in the repair process.

Example 1.1: Consider Table I and the constraints:

F_1: [District, Region] $\rightarrow$ [AreaCode]
The table violates this constraint. We could repair the violations by changing the data, but we would have to change either $t_1 - t_3$ or $t_4 - t_5$ and either $t_6 - t_8$ or $t_9 - t_{11}$. There is no evidence in the data to tell us which tuples would be "better" to change, but more importantly, we would be changing tuples that all have the exact same error. So perhaps it is not the data that is inconsistent, but the constraint. Further evidence supporting this conclusion is that the tuples in disagreement, can be distinguished by their Municipality. Hence, it may be that the context in which this constraint applies has changed. We can repair this inconsistency by adding Municipal to the antecedent to make F_1 consistent.

F_2: [Zip] $\rightarrow$ [City, State]
Tuples $t_1 - t_3$ do not satisfy F_2, and t_8 is in conflict with t_6, t_7, t_9. Based on the data, it appears that the City value of Chester in t_8 may be incorrect (since t_6, t_7, t_9 are all in agreement). Similarly, we may conclude it is more likely that the State of t_3 is incorrect (MA should be NY).

F_3: [PhNo, Zip] $\rightarrow$ [Street]
Tuples t_{10}, t_{11} do not satisfy F_3 and we can consider updating the Street value of either tuple to remove the violation. However, it is not clear what the correct value should be (Bay or Main). There is more support for Main in the data, but Main always appears with the Zip 60415, except in tuple t_{10}. Hence, based on the data, there is more evidence to indicate that the Zip of tuple t_{10} may actually be incorrect, rather than its Street.

In the example, we have justified intuitively a case where a constraint repair seems better than a data repair, and a case where one data repair seems better than another. In this paper, we present a new cost model that reflects this intuition and allows us to quantify the trade off between different constraint repairs and different data repairs that may be possible.

In this work, we focus on functional dependencies (FDs), as such constraints are a primary tool used to enforce data consistency in practice and are known to be very important in improving data quality [10], [2]. Changes in the domain semantics will necessitate changes to these rules. Towards a goal of improving long term data quality, we focus on rules that have sufficient evidence in the data. That is, we are interested in repairing rules that will continue to hold over time, and that are well supported in the data. We formalize the notion of evidence (support) by measuring the amount of data redundancy that exists with respect to the rule. For well-supported rules, we have a hope of automating the repair process and determining with reasonable accuracy good repairs. In contrast, we do not consider low support rules, such as keys, as there may be little evidence in the data to determine which attribute (the key value itself or a descriptive attribute) should be repaired. Such rules are already well studied in the duplicate detection and identification literature where other techniques, including clustering, are most useful.

Problem Statement: Given a set of functional dependencies Σ, that are inconsistent with respect to a database instance I, find a set of low cost data repairs and a repair of Σ that will create an I' and concise Σ' that are consistent.

Contributions:

- A new cost model for data and constraint repairs over a database that is inconsistent with respect to a set of constraints.
- A data repair algorithm that searches for data modifications such that the constraints hold and the repair cost is minimal. For a functional dependency $F : X \rightarrow Y$ and tuples t_1, t_2 that violate F, we can repair these tuples by either modifying their Y values to be the same, or by modifying their X values to be different. In both cases, we require that the data support the change. In particular, when modifying X values, we only consider values for X that are supported by other tuples.
- A constraint repair algorithm that determines which attribute to add to a constraint to resolve the inconsistency. We select attributes for the repair using an adapted notion of the variance of information between the new attribute and the inconsistent FD that quantifies how well an attribute repairs the inconsistency without lowering the redundancy in the constraint.
- We evaluate our repair algorithms separately (as stand-alone algorithms). We do this because in some cases we may have a

carefully curated set of constraints that we trust and we may choose to only make data repairs. Alternatively, we may have a data set that cannot be updated (due to security reasons, or to ensure compliance with business regulations), so we may choose to only make constraint repairs. Hence, we present an evaluation to show the accuracy and performance of our data repair and our constraint repair algorithms individually.

- We conduct a case study using the Cora bibliographic dataset to evaluate our cost model and the quality of the recommended repairs. We show that our model can effectively determine when a data vs. constraint repair is best, and that our algorithms recommend repairs that help to improve the quality of the data.

II. Preliminaries

We begin by formally defining the constraints, constraint violations, and repairs that we will consider.

For an instance (relation) I over a set of attributes $\mathbf{R}$, $N = |I|$ denotes the cardinality (number of tuples) of I and $|\mathbf{R}|$ denotes the arity (number of attributes). For attribute sets X and Y, we will follow convention and use XY to denote the union of X and Y.

A functional dependency (FD) F over a relation $\mathbf{R}$ is represented by $F : X \to Y$, where X, Y are attribute sets in $\mathbf{R}$. We will denote the attributes of F by $\mathbf{F} = XY$. We will use the shorthand $|\mathbf{F}|$ to denote $|XY|$. An instance I of $\mathbf{R}$ satisfies F (written as $I \vDash F$) if for every pair of tuples t_1, t_2 in I, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. An instance I is *inconsistent* with respect to (wrt) F (or Σ) if it violates F (or any of the FDs in Σ). Without loss of generality, we assume all FDs $F : X \to Y$ have been decomposed so Y contains a single attribute.

A. Data Repairs

Suppose we have tuples t_1 and t_2 that violate an FD $F : X \to Y$, so t_1 and t_2 agree on X ($t_1[X] = t_2[X]$) and disagree on Y ($t_1[Y] \neq t_2[Y]$). To repair such a violation, we can change some attribute values in either of the following ways (without loss of generality, we assume t_2 is changed).

Type 1 *Repair the* Y *values to be the same.*
Here, we would change $t_2[Y]$ to equal $t_1[Y]$. We will do this when there is sufficient evidence in the data to conclude the $t_1[Y]$ is the correct value to associate with $t_1[X]$.

Type 2 *Change the* X *values to be different.*
Here, we would change $t_2[X]$ to be different from $t_1[X]$. Obviously, we would not pick an arbitrary X value. Instead, we will look for evidence in the data that one or more of the attribute values in X is incorrect by finding a tuple t_3 where $t_3[Y] = t_2[Y]$ and $t_3[X]$ is close to $t_2[X]$ (where we will define a precise notion of closeness). If $t_3[X]$ appears frequently in the data, then we will consider changing $t_2[X]$ to $t_3[X]$. In other words, we have $t_2[X] \neq t_3[X]$ but these values are similar, and $t_2[Y] = t_3[Y]$ where there is a lot of evidence supporting $t_3[X]$. Hence, we change t_2's X values to match t_3. We impose no preference on the type of repair selected, unlike previous work that focused on data repairs only on Y (Type 1) [1]. Rather than such an *a priori* preference, we will present a cost model for data repairs and use this as the basis for selecting repairs.

B. Constraint Repairs

In addition to the above two types of data repairs, we consider repairing violated FDs. In this work, we consider one type of constraint repair for an FD $F : X \to Y$: the addition of an attribute to X. The additional attribute provides additional context for defining when the rule actually holds. Such a repair is necessitated frequently when integrating data, or as legacy relations evolve to be used for new types of information.

In considering a constraint repair, we will also evaluate whether there is a cheap way to repair the data to resolve the inconsistency. This may be the case if only a few tuples violate the constraint and these tuples are close to other well-supported values in the relation. However, if many tuples violate the constraint and there is an attribute whose values neatly separate the conflicting tuples (as was the case in Example 1.1, for constraint F_1), then clearly a constraint repair should be considered. However, the trade-offs can be subtle between constraint and data repair. If there are only a few violators, it still may be the case that a constraint repair is in order, especially if no low-cost data repairs can be found. Notably, previous work on repairing inconsistency only considered data, not constraint repairs [1], [2], [3]. An alternative approach assumes that a schema mapping is given between an old (source) and new (target) schema and infers a set of target dependencies that are logically implied by the source constraints and the mapping [11]. This is called constraint propagation and it assumes that the constraints have not evolved or changed, they are simply being updated to reflect a new schema. Such dependency propagation (inference) is complementary, but orthogonal, to our approach. Notably, we do not assume the schema has changed or that any well-design and correct schema mapping is given.

III. A Unified Repair Model

One of the contributions of our work is a new cost model that quantifies the trade-off of when an inconsistency is a true data error (warranting a data repair) vs. an update to the model (justifying a constraint repair). Our goal is to find a set of minimal repairs to either I or Σ such that the updated I', Σ' are consistent, i.e., $I' \vDash \Sigma'$. We present a cost model that evaluates the cost of potential data repairs and constraint repairs, and selects repairs with lowest cost. Our cost model is based on the following properties.

- **Accuracy**: We consider updates to a data value v_s that will transform it to a target value v_t with frequency f_{v_t}. Let θ be a given support threshold. To measure accuracy, we use a distance measure $r = dist(v_s, v_t)$ between values, and seek updates that minimize r, and where $f_{v_t} \geq \theta$ such that there is sufficient support in the relation for this change.
- **Redundancy**: Let $F : X \to Y$ be a constraint that holds over I (even approximately). Let $|I \models F|$ be the size of

the largest subset of I that satisfies F. We seek repairs for inconsistent data values $x \in X, y \in Y$ to x', y' such that $|\sigma_{X=x',Y=y'}(I)|$ has sufficient support. For constraint repairs, we seek attributes $A \in \mathbf{R}$, such that $|I \models F'| > |I \models F|$ for $F' : (X \cup A) \rightarrow Y$ (that is, after adding A to F there are more tuples satisfying the FD).

- **Conciseness**: Our repaired constraints should explain as much of the data as possible, but without overfitting. That is, large constraints (with many attributes in the antecedent) may explain the data precisely but with a high representation cost. Conversely, constraints with fewer attributes may not be as precise. Our goal is to find repairs leading to concise constraints that maximizes $|I \models F'|$.

A. Minimum Description Length Principle

To explain our framework, we begin by assuming there is a single FD $F : X \rightarrow Y$ over a relation I, and we seek to build a model M for F. (We consider multiple FDs later.) The better F is at encoding a set of values, the smaller its description length (DL) will be. The Minimum Description Length (MDL) principle [12], [13] defines DL for M as the length of the model $L(M)$, plus the length to encode the data values in a relation I given the model $L(I|M)$. We seek accurate repairs to I, and repairs leading to concise F (hence small $L(M)$) that capture as much redundancy in the data as possible (hence minimizing $L(I|M)$).

Assume that $I \models F$ and that we start with a model $M = \emptyset$. Then $L(M) = 0$, but to describe the data, we need $L(I|M) = |XY| * |I|$ "cells" (given a tuple, we refer to an attribute value as a cell, and we will use cells as the unit for our DL). Now suppose there is a tuple $s \in \Pi_{XY}(I)$ that appears frequently in I. If the single tuple s is added to M, then this precisely describes the Y value associated with this X value. Suppose s appears f times in $\Pi_{XY}(I)$. Then, if we add s (a single tuple with $|XY|$ cells) to M yielding a new model M', we can reduce $L(I|M)$ by $f * |XY|$. Hence, $(L(M') + L(I|M')$ $= |XY| + (|I| - f * |XY|)) < L(M) + L(I|M)$, so we have produced a model M' with a lower description length. Of course this simple idea is made more complicated when $I \not\models F$. In general, we will try to find repairs for erroneous tuples in I, that allow us to find a concise model for the repaired instance I'. Alternatively, we will try to repair F so that we can find a concise model for F'.

Assume we begin with an empty model M. We add to M tuples from $\Pi_{XY}(I)$ that have high support, and that do not conflict with tuples in M wrt F.

Defn 3.1: A **tuple pattern** p is a single tuple over XY that exists in $\Pi_{XY}(I)$. The frequency of p, f_p, is the number of times p occurs in I.

Defn 3.2: A **model** M is a set of **signatures** where each signature $s \in \Pi_{XY}(I)$ is a tuple pattern.

For example, in Table I, we can consider '60415 Chicago IL' as a signature since it is a frequently occurring tuple pattern. Our goal will be to find signatures that lower the overall DL cost (via $L(I|M)$, by summarizing frequently occurring tuple patterns in I). For a constraint F, we build a model M by adding signatures to represent as much of the data I as possible wrt F. (That is, we do not add tuples to M that together with M would violate F.) Hence, we seek signatures that capture as much regularity (redundancy) as possible in I. The length of the model $L(M)$ plus the length to encode I given M, $L(I|M)$, should be as small as possible. We can consider general models where $L(M)$ is very small, and $L(I|M)$ is very large because M will accept almost any data value, and we must explicitly write out the values in I. On the other end, we can consider very specific models where M models almost the exact values in I and hence $L(I|M)$ is close to zero. We seek an M between the general and specific models that minimizes the description length $DL = L(M) + L(I|M)$. In our model, we use a unit cost for each cell in a relation. Specifically, we compute the description length as

$$L(M) = |XY| * S, \text{ and } L(I|M) = |XY| * E$$

where S is the number of signatures s in M. E is the number of tuples in I not represented by an s in M. Hence, $L(I|M)$ is the length of encoding these tuples if we leave the data as is and we do not repair the data.

IV. Data Repair

To begin, assume we have a single FD $F : X \rightarrow Y$. We will start with an empty model M. Hence, $L(M) = 0$ and $L(I|M) = |XY| * N$ the cost to represent all tuples in I wrt F. As we add signatures to M (that do not conflict with existing s in M), our goal will be to minimize $L(M) + L(I|M)$. Each new signature increases $L(M)$ but also decreases $L(I|M)$. If there are tuple patterns that occur frequently, then replacing them with a signature in M should reduce $L(I|M)$ and only slightly increase $L(M)$, resulting in an overall lower DL.

A. Tuple Patterns: Cores and Deviants

To compare potential data repairs, we assume for each attribute $A \in \mathbf{R}$, a distance function $0 \leq dist_A(v_1, v_2) \leq 1$ over values in the domain of A. Here, $dist_A(v, v) = 0$. For different values, $dist_A(v_1, v_2)$ indicates how different the values are. The distance functions can be a simple 0 or 1 function where 0 indicates the values are the same, and 1 indicates they are different. Alternatively, we can use distance functions tailored to the domain (for example, in our experiments, we use the Jaro-Winkler measure for strings [14], see Section VII).

For tuple patterns, we assume the existence of a similarity measure between tuple patterns $0 \leq sim(p_1, p_2) \leq 1$ where $sim(p, p) = 1$. This similarity function may be a weighted sum of the attribute distance, or an aggregate over the attribute distance. For our work, we define sim as the fraction of equal attributes in the patterns.

$$\sigma_A(p_1, p_2) = \begin{cases} 1 & \text{if } dist_A(v_1, v_2) = 0 \\ 0 & \text{else} \end{cases}$$

$$sim(p_1, p_2) = \frac{\sum_{A \in XY} \sigma_A(p_1, p_2)}{|XY|}$$

To guide our algorithm, we will use a support threshold θ and a pattern similarity threshold β. Intuitively, tuple patterns with a frequency over $\theta * N$ will not be considered as candidates for a data repair. Furthermore, a candidate for a data repair must be at least as similar to some other pattern as our threshold β. Our similarity function is appropriate when data errors in different attributes are introduced independently. If this is not the case, a different similarity function could be used. Specific data cleaning requirements will often determine appropriate values for β. For example, if data errors are rare, we will want to use a higher threshold β. Based on this, we define core tuple patterns and deviations.

Defn 4.1: A **core (tuple) pattern** $p \in \Pi_{XY}(I)$ is any tuple pattern with freq(p) $\geq (\theta * N)$.

Defn 4.2: A **deviant (tuple) pattern** $d \in \Pi_{XY}(I)$ is a tuple pattern with freq(d) $< (\theta * N)$ such that there exists at least one core pattern c such that $sim(d, c) \geq \beta$.

In developing M, we initially consider all core patterns as signatures (if there are two conflicting core patterns, we select the pattern with higher support). As a simple example, if $\theta * N$ is 3, then for a core pattern we save $(2 * |XY|)$ since the cost to represent the signature is $|XY|$ and we are removing three tuples (that otherwise would be encoded in $L(I|M)$). Deviants are candidates for repair. In our algorithm, we compare the cost of different data repairs. We do this in two ways. The first is by using the similarity functions on attributes (sim_A). Second, we evaluate the potential impact of the data repair on other constraints, an issue we consider in the next section.

Example 4.3: Consider Table I and the FD F_2. Let $\theta = 0.18$. The core patterns are given below.

- p_1: '10211 NY NY'
- p_2: '02215 Boston MA'
- p_3: '60415 Chicago IL'
- p_4: '60601 Chicago IL'

Let $\beta = 0.66$, a deviant must have at least two attribute values in common with a core pattern. The deviants are:

- d_1: '10211 NY MA'
- d_2: '60415 Chester IL'

Notice that d_1 is a deviant of p_1, while d_2 is a deviant of p_3. If we lower $\beta = 0.33$, then d_1 is a deviant of p_1 and p_2, and d_2 would be a deviant of p_3 and p_4. Selecting a lower β clearly leads to greater evaluation costs as there are more options to consider for a data repair. Furthermore, as this example shows, a β that is too low may lead to undesirable repairs.

B. Handling Multiple Constraints

Constraints that share common attributes have the potential to contain overlapping data inconsistencies. If a data repair updates a value v_s to v_t when considering an FD F_i, this update may also impact F_j. We would like to be able to detect these shared interactions such that when we consider the repair wrt F_i, we are aware of whether (and how) this update will impact F_j. Specifically, we compute the increase to $L(M_{F_i})$, to add the signature s to M_{F_i}, the potential reduction to $L(I|M_{F_i})$, the cost to transform v_s to v_t, and the change to the description length ΔDL_{F_j} wrt F_j due to this update. We check whether implementing this update creates a new core pattern that could be added as a signature for the model M_{F_j} and lowers $L(I|M_{F_j})$, thereby decreasing the overall DL_{F_j}. Alternatively, the update may create a new deviant and remove a potential core pattern for F_j. Hence, we consider the change ΔDL_{F_j}. For the data repair cost, we consider updating each v_s in d to the corresponding v_t in p. The cost of the data repair is $(1+r)$, where $r = dist_A(v_s, v_t)$. For our case, $dist_A$ is the Jaro-Winkler distance if v_s and v_t are strings, and the normalized Euclidean distance if the values are numeric. We assume $r \in [0, 1]$, with 0 indicating exact similarity and 1 indicating no similarity. The leading 1 is the unit cost for updating a cell value. We evaluate (recursively) the impact of this data repair on affected constraints.

While our data repair algorithm is in similar spirit to previous techniques [1], [2], our distinct look ahead mechanism is useful in recognizing which data repairs have added benefits of resolving future inconsistencies. Alternatively, a current data repair may not be selected because it negatively reduces the amount of redundancy captured by other constraints. We note that to avoid a non-terminating cascade of updates, our algorithms are greedy and do not undo updates that have already been done. When the DL can no longer be improved via data repairs, the learning process terminates and we return the current model M as the best model (along with the associated data repair costs), which will be compared against the constraint repair model costs. Further details are given in Algorithm 1.

Example 4.4: We show a simple example of how a model M is found and how $L(M)$ and $L(I|M)$ are computed for $F_2 : [Zip] \rightarrow [City, State]$. Consider only the first five tuples in Table I, so $I = \{t_1, t_2, t_3, t_4, t_5\}$. Let $\theta = 0.4$, and $\beta = 0.66$, then we have:

- p_1 = '10211 NY NY'
- p_2 = '02215 Boston MA'
- d = '10211 NY MA', v_s = 'MA' in t_3 State

If $\theta = 0.2$, we identify '10211 NY MA' as a core pattern, however, it would not be used since adding this pattern to M would not lower the DL cost. The learning proceeds as follows:

- Initialize M to contain p_1 and p_2 as signatures. Since each core pattern has a representation cost of 3, we get $L(M) = 6$. There are (3 * 5) cells to represent I wrt F_2, and initially $L(I|M) = 15$. After adding the signatures, we reduce $L(I|M)$ by 12 since we save $(f_p * |XY|)$ for each core pattern, thus $DL = 9$.
- Adding d to M will not reduce the DL cost. We consider two possible data repairs, based on the value of β:

1) If $\beta = 0.66$, then d is a deviant of p_1. We update t_3 'MA' to 'NY'. $dist_{state}('MA', 'NY') = 1$, then the repair cost is 2.
2) If $\beta = 0.33$, then d is a deviant of p_1 and p_2, and there are two possible data repairs. We can update t_3 as described above ($r_1 = 2$). Alternatively, we could update d to p_2, which requires updating '10211' to '02215', and 'NY' to 'Boston'. The cost of this repair would be r_2 = (1 + $dist_{zip}$('10211','02215')) + (1 + $dist_{city}$('NY','Boston')) = (1+0.267) + (1+1) = 3.26. Since $r_1 < r_2$, we choose the first repair.

We evaluate the impact of this repair on the DL. By leveraging the signature '10211 NY NY' in M, we decrease $L(I|M)$ by 3. However, we need to add the repair cost to the model, so $DL' = L(M) + r_1 + L(I|M) = 6 + 2 + (3 - 3) = 8$.

V. Constraint Repair

In Example 1.1, F_1 is inconsistent with the given relation I. All the tuple values are relatively evenly distributed and there is no consequent value that is clearly erroneous, and hence no obvious data repair. Some inconsistencies such as this reveal that the constraints themselves may be outdated and need to be repaired. If we consider adding the attribute `Municipal` to F_1's antecedent, this will make the new F_1' consistent with I. Furthermore, the original DL_{F_1} cost is 33 (that is, if $M = \emptyset$). None of the core patterns for F_1 satisfy the constraint, so we cannot add all of them to the model without doing data repairs. For this example, and in general, such repairs may be expensive. An alternative that we consider is to modify F_1 by adding `Municipal`. This constraint repair allows us to add 4 signatures to the model (albeit longer signatures each of length four) that are consistent with the new constraint giving a new $DL'_{F_1'} = 16$. In this section, we discuss how we evaluate candidate attributes for constraint repair. Due to space limitations, we consider only repairs that create a new FD by adding an attribute A to X, such that I is (more) consistent with respect to the new F', rather than repairs that add a condition to create a conditional functional dependency (CFD). Our algorithm can be extended to search for (conditional) constant values by applying techniques from CFD discovery algorithms [10].

A. Attribute Partitionings

We first describe how each candidate attribute A is evaluated when it is being considered as a possible addition to X to repair a constraint. We want to find an A that has the most similar distribution of values with Y on inconsistent tuples. For F_1, the `Municipal` attribute has the same distribution of values as `AreaCode` in tuples (t_1, t_2, t_3), (t_4, t_5), (t_6, t_7, t_8), and (t_9, t_{10}, t_{11}) where the inconsistencies were found. If A contains different values in consistent (non-violating) tuples, our model for $F' : A \cup X \rightarrow Y$ becomes larger at the expense of a loss in redundancy and so the addition of A may be undesirable. We need to compare how the values in X, Y compare with values in A. We do this by modeling the attribute values as classes.

Let $\mathcal{C}_X$ be a partitioning where each class in the partitioning contains all tuples that share the same X value. Let $\mathcal{C}_{XY}$ be a refinement of $\mathcal{C}_X$ containing a class for each set of tuples that share the same X and Y values. The partitioning $\mathcal{C}_{XY}$ ranges over the possible XY values, and two $\mathcal{C}_{XY}$ classes will differ in either X or Y. A class in $\mathcal{C}_X$ may contain tuples from multiple $\mathcal{C}_{XY}$ classes, and if this occurs, these are tuples involved in a violation of F.

Example 5.1: The partitionings for $F_1 - F_3$ of Example 1.1 are given below. Again, we use $\mathbf{F_i}$ as a shorthand for the attributes in F_i, that is X_iY_i. For each class of X_iY_i, we divide the tuples of the classes based on their X_i classes (so each class of $\mathcal{C}_{X_i}$ is shown within braces {...}). Since the partitioning for X_iY_i are a refinement of the classes for X_i, we show the partitioning for X_iY_i where each $\mathcal{C}_{XY}$ is in parentheses (...). Hence, we show the partitionings for X_i and X_iY_i in the same line. The partitioning for attribute $A =$ `Municipal` is also given.

- $\mathcal{C}_{\mathbf{F_1}}$: {(1,2,3) (4,5)} {(6,7,8) (9,10,11)}
- $\mathcal{C}_{\mathbf{F_2}}$: {(1,2) (3)} {4,5} {(6,7,9) (8)} {10, 11}
- $\mathcal{C}_{\mathbf{F_3}}$: {(1,2)}{(3)}{(4,5)}{(6)}{(7)}{(8)}{(9)}{(10)(11)}
- $\mathcal{C}_A$: {1,2,3} {4,5} {6,7,8} {9,10,11}

B. Variance of Information

To understand if we should add an attribute A or an attribute B to F, we can use some ideas from clustering, specifically techniques for evaluating different clusterings. Suppose we view $\mathcal{C}_{\mathbf{F}}$ as a "ground truth" clustering. For a new attribute A, we would like to understand how well $\mathcal{C}_A$ matches $\mathcal{C}_{\mathbf{F}}$. However, there is one caveat. The classes $\mathcal{C}_{XY}$ indicate that each class should map to a distinct class in $\mathcal{C}_A$. However, for our purposes two classes in $\mathcal{C}_{XY}$ from distinct $\mathcal{C}_X$ may map to the same class in $\mathcal{C}_A$. We present a modified version of a clustering evaluation measure that evaluates the suitability of a candidate attribute.

We first present some terminology. Assume we have taken our data set I and divided it into a set of classes $\mathcal{C}_{XY}$. So the number of classes is $C = |\mathcal{C}_{XY}|$ and each class is denoted by c_{xy}. Furthermore, let us assume that this classification (which is based on F) constitutes our *ground truth* classification of I. We will consider new clusterings of I into a new set of K clusters. The clustering we will consider are groupings of tuples by a new attribute A. Hence, $K = |\mathcal{C}_A|$ classes each represented by c_a, where a ranges over the possible values of A. In a *homogeneous* clustering, every c_a only contains elements from a c_{xy} (intuitively, the value a corresponds only to the values x, y, but there may be an $a' \neq a$ that also corresponds to x, y). In a *complete* clustering, all elements of c_{xy} are assigned to the same c_a (intuitively, the values x, y only correspond to the value a). Assuming that the elements in the data set are drawn from a uniform distribution, we can define the conditional entropies of these clusterings

- $H(C|K) = -\sum_{xy} \sum_a p(c_{xy}, c_a) \log p(c_{xy}|c_a)$
- $H(K|C) = -\sum_a \sum_{xy} p(c_{xy}, c_a) \log p(c_a|c_{xy})$

where $p(c_{xy}, c_a) = \frac{|(c_{xy} \cap c_a)|}{N}$, and $p(c_{xy}|c_a) = \frac{p(c_{xy} \cap c_a)}{p(c_a)}$. The summation is taken over all values of $\Pi_{XY}(I)$ (respectively, $\Pi_A(I)$). For homogeneity, each c_a must be contained in a c_{xy}, hence minimizing $H(C|K)$ towards zero. Similarly, for a clustering to be complete, each c_{xy} must be contained in a particular c_a, minimizing $H(K|C)$. The *Variance of Information (VI)* measure [15] is defined as

$$VI(C, K) = H(C|K) + H(K|C)$$

In the least homogeneous (or complete) clustering, the value of $H(C|K)$ (or $H(K|C)$) is maximal. Hence, lower VI values are preferable, with the perfect homogeneous, complete solution having VI $= 0$. The value of VI ranges from $[0, 2 \log N]$.

C. Comparing Attributes

For constraint repair, we would like $\mathcal{C}_A$ to be a *homogeneous* clustering with respect to a "ground truth" classification $\mathcal{C}_{XY}$. That is, ideally, we do not want any class in $\mathcal{C}_A$ to contain values from multiple c_{xy} belonging to the same $\mathcal{C}_X$. Otherwise, attribute A has not helped to separate (correct) these inconsistent tuple values, and we cannot encode these values in our model. If $\mathcal{C}_A$ is incomplete, we encode all the tuple values for a given c_{xy} with additional signatures (one for each c_a that contains c_{xy} tuples). A clustering $\mathcal{C}_A$ may be heterogeneous with respect to $\mathcal{C}_X$ since a single value of A may map to multiple classes in $\mathcal{C}_{XY}$ without violating the FD F. To capture this, we compute a modified version of homogeneity, $H(C|K_X)$.

This is similar to $H(C|K)$ but instead of considering all the tuples in c_a, we compute the homogeneity of c_a wrt each class of X ($\mathcal{C}_X$) rather than XY. That is, we exclude from the computation all tuples in c_a that do not share the same value of X. We compute completeness $H(K|C)$ of each class c_a, for every $a \in A$. Our objective is to find attributes whose value distributions coincide with tuples that require repair. We select the attribute A with the lowest homogeneity score (contains the largest number of values that can be encoded). In cases of a tie, we select the A with the lower completeness score. An example calculation is given next.

Example 5.2: We show how $H(C|K_X)$ and $H(K|C)$ are calculated for each candidate attribute to repair an inconsistent FD. Consider F_1 : $\texttt{[District, Region]} \rightarrow \texttt{[AreaCode]}$, and the following attributes A:

- $\mathcal{C}_{F_1}$: {(1,2,3) (4,5)} {(6,7,8) (9,10,11)}
- $\mathcal{C}_{A_1=\texttt{Municipal}}$: (1,2,3) (4,5) (6,7,8) (9,10,11)
- $\mathcal{C}_{A_2=\texttt{Street}}$: (1,2) (3) (4,5) (6) (7,9,10) (8) (11)
- $\mathcal{C}_{A_3=\texttt{State}}$: (1,2) (3,4,5) (6,7,8,9,10,11)
- $\mathcal{C}_{A_4=\texttt{PhNo}}$: (1,2) (3) (4,5,6) (7) (8) (9,10,11)
- $H_{A_1}(K|C) = H_{A_1}(C|K_X) = 0$. Each of the c_a classes is complete and homogeneous.
- $H_{A_2}(C|K_X) = (\frac{1}{11}\log\frac{1}{3}) + (\frac{2}{11}\log\frac{2}{3}) = 0.075$, $H_{A_2}(K|C) = (2*(\frac{2}{11}\log\frac{2}{3}) + (5*(\frac{1}{11}\log\frac{1}{3}))) = 0.281$. Class (7,9,10) contains heterogeneous terms since all the tuples belong to the same $\mathcal{C}_X$.
- $H_{A_3}(C|K_X) = 0.239$, $H_{A_3}(K|C) = 0.075$ and $H_{A_4}(C|K_X) = 0$, $H_{A_4}(K|C) = 0.205$. $\mathcal{C}_{A_4}$ classes (3),(7),(8),(1,2), and (4,5,6) are incomplete. The clustering is homogeneous even though class (4,5,6) contains tuples from different c_{xy}, they are from distinct $\mathcal{C}_X$.

The homogeneity scores show that `Municipal` is the best attribute to repair F_1, followed by `PhNo`, `Street` and then `State`. The DL values of the revised F'_1 with each of the attributes are $DL_{A_1} = 16, DL_{A_2} = 36, DL_{A_3} = 40, DL_{A_4} = 24$, and reinforces homogeneity as the correct measure.

VI. MDL Repair Algorithm

We have presented algorithms for repairing data that is inconsistent with a constraint, and for repairing a constraint. We now take these pieces and put them together in an MDL Repair Algorithm. Specifically, given a set of constraints $\mathcal{F}$ and a database I that is inconsistent with $\mathcal{F}$, our algorithm will select a set of low cost data and constraint repairs that produce an $\mathcal{F}'$ and I' such that $I' \models \mathcal{F}'$. We first review the relationship between the repair costs and the model description length, followed by a discussion on the order in which constraints should be considered, and finally we present the algorithm.

A. Cost Model Review

For $F : X \rightarrow Y$, $DL_F = L(M) + L(I|M)$ where $L(M) = (S * |XY|) + r_T$, S is the number of signatures in the model M, and r_T is the total data repair cost for updated tuples in I. Then $L(I|M) = E * |XY|$ where E is the number of tuples not modeled by the S signatures in M. At each step, our greedy algorithm selects the data or constraint repair that reduces DL the most. If a data repair is selected, we add the smallest update cost (1+r) (remember $r = dist(v_s, v_t)$ for a source value v_s and a target value v_t) to r_T such that the repaired tuple(s) match an existing signature in M. Then we reduce $L(I|M) = (E - t_c) * |XY|$, where t_c are the number of new consistent (repaired) tuples.

For constraint repair, we consider a new FD $F' : (X \cup A) \rightarrow Y$ where $|I \models F'| > |I \models F|$ (there are more tuples satisfying F' than F). We choose the constraint repair F' only if $DL_{F'} = S' * (|XY|+1) + E' * (|XY|+1)$ is less than DL_F, where S' is the number of signatures in the model wrt F', and E' is the number of tuples not modeled by the S' signatures. As mentioned in Section V-C, we select an attribute A such that $\mathcal{C}_A$ is homogeneous in $\mathcal{C}_{XY}$ (and most complete in cases of a tie). If the clustering is not homogeneous, then A has not helped to separate these conflicting tuple values, and since they cannot be represented by a signature, they will remain inconsistent and do not contribute towards reducing DL.

B. Ordering Constraints

When a relation I is inconsistent with multiple constraints, our algorithms must choose an order for processing the constraints. We consider two criteria for ordering constraints. First, we consider the degree of inconsistency of each $F : X \rightarrow Y$. An x value is inconsistent if it appears in I with more than one y value. The degree of inconsistency for x is the number of y values such that $xy \in I$. For an FD F, we can then define the *degree of inconsistency*.

Defn 6.1: The *degree of inconsistency* (ic_F) of F with I is:

$$ic_F = \frac{\sum_{x \in \Pi_X(I)} (|\Pi_Y(\sigma_{X=x}(I))| - 1)}{|\Pi_{XY}(I)|}$$

If I is consistent, $ic_F = 0$. The closer ic_F is to 1, the more inconsistent the relation is wrt F.

The second criteria we consider are the potential conflicts F shares with other inconsistent constraints F', defined based on the number of attributes they have in common ($|\mathbf{F} \cap \mathbf{F'}|$).

Defn 6.2: The *conflict score* of an FD F with a set of FDs $\mathcal{F}$ is (recall $\mathbf{F}$ denotes the attributes in F):

$$cf_F = \frac{\sum_{F' \in \mathcal{F}} \frac{|\mathbf{F} \cap \mathbf{F}'|}{max(|\mathbf{F}|,|\mathbf{F}'|)}}{|\mathcal{F}|}$$

The conflict score $cf_F \in [0, 1]$, with 0 indicating no overlap and 1 indicating that all the attributes overlap (with all other FDs). Constraints with high cf values have the greatest potential of repair conflicts with other rules. Since both our evaluation scores are normalized, we average the two values to get a combined score O_F,

$$O_F = \frac{ic_F + cf_F}{2}$$

We evaluate multiple constraints in decreasing O_F order, since large values indicate rules with the highest degree of inconsistency and the greatest potential of repair conflicts with other constraints. Note that this is a heuristic meant to improve the efficiency of the repair process by doing the heavier work upfront, and minimizing the amount of conflicts later on.

C. Repair Algorithm

Our repair algorithm consists of a main driver routine to *COMPUTE REPAIRS* that compares the cost of data and constraint repairs, picking the repair of lower cost. For each inconsistent F, we compute the core patterns p, deviants d, and the ic_F, cf_F values. We select an F from priority queue L (in decreasing O_F order), and search for the data and constraint repairs. For the former (*FIND DATA REPAIRS*), given an F, then for each deviant d of F, we determine the lowest cost repair for d over all p for which it is a deviant. If the data repair impacts other constraints, then this impact cost is calculated and added to $cost_d$. If the new model M' (that includes the data update) results in a decrease to the description length DL, then we include this update in our recommended list of data repairs for F. The cost of the update is added to the cumulative repair cost for F. To search for constraint repairs for F (*FIND CONSTRAINT REPAIRS*), we compute the homogeneity and completeness score of each attribute not in F. The attribute with the lowest homogeneity score is the winning attribute (if there is a tie, the attribute with the lowest completeness score is used), and a constraint repair is recommended if $cost_{constr}$ (with the winning attribute) is less than $cost_{data}$. After applying the repairs to I and Σ, $I' \models \Sigma'$. Pseudocode is given in Algorithms 1 and 2.

VII. Experimental Evaluation

We conducted a qualitative and performance evaluation of our repair algorithms using both real and synthetic datasets. Our experiments were run using a Dual Core AMD Opteron Processor 270 (2GHz) with 6GB of memory. We used the TPC-H dbgen data generator (PART table) for the performance and first set of quality tests, the Veterans of America [16] real dataset for our comparative study, and the CORA bibliographic real dataset for our case study [17]. To evaluate string similarity in data repairs, we used the Jaro-Winkler distance measure [14], and the normalized Euclidean distance for numeric data. In the case study, as real data often contains empty (NULL) values, to favor meaningful data repairs, we imposed a penalty (larger data repair cost) on updates that transform a source value to an empty target value. Based on our experiments, we chose β to equal at least 0.4 (at least 40% of the attributes are similar between a core pattern and a deviant), and manually adjusted θ, to generate more selective data repairs.

Algorithm 1 Data repair algorithm

```
COMPUTE_REPAIRS ()
INPUT: $|\Sigma|$: number of constraints defined over $I$
bConstraintRepair: flag to search for constraint repairs
evalConstraints[]: list of already evaluated constraints
setup structure $FList$ that maintains info on each $F$
read_data_values(): read tuples from $I$
for $F$ in $|\Sigma|$ do
    compute_core_deviant_patterns($F$): get $p$ and $d$ wrt $F$
    initialize_model($F$): initialize $M_F$ to all core patterns $p$, and set $L(M_F)$, $L(I|M_F)$.
    compute $ic_F$ : set $FList(F)$.score
$L$ = sort($FList$.score, DSC)
for $F$ in pop($L$) do
    if (!bConstraintRepair) then
        $cost_{data}$ = FIND_DATA_REPAIRS($F$)
    push(evalConstraints[], $F$): mark $F$ as evaluated
    if (bConstraintRepair) then
        $cost_{constr}$ = FIND_CONSTRAINT_REPAIRS($F$)
    if ($cost_{data} \leq cost_{constr}$) then apply data repairs
    else apply constraint repairs

FIND_DATA_REPAIRS($F$)
$cost_F = L(M_F) + L(I|M_F)$
$F^*$: constraints impacted by current repair
for $d$ a deviant of $F$ do
    $cost_d$ = get_best_cost($d$): best cost of $d$
    $cost_d$ += get_impact_cost($d$, $F^*$)
    $\Delta DL$ =get_updated_model_cost($F, M_F, d, cost_d$)
    if ($\Delta DL < 0$) then
        $M_F$ =update($M_F, F, d$)
        update_constraints_after_repair($F$, $d$,$F^*$)
        $cost_F + = cost_d$
        add_repair($F, d$)
        add_repair($F^*, d$)
return $cost_F$
```

A. Repair Quality

We evaluated two types of errors: errors that are introduced during the repair process (incorrect repairs); and errors that are not resolved, because the algorithm determines the cost is too high. We used precision and recall to measure these two types of errors for data and constraint repairs:

- $precision_{data} = \frac{\#correctRepairs}{\#totalRepairs}$
- $recall_{data} = \frac{\#correctRepairs}{\#totalErrors}$
- $precision_{constraint} = \frac{\#correctedTuples}{\#totalModifiedTuples}$

Algorithm 2 Constraint repair algorithm

```
FIND_CONSTRAINT_REPAIRS ($F$)
Input: $n$ := $|\mathbf{R}|$
$cost_F = best\_cost_F$ = MAX_COST;
best_score = MAX_COST; cur_score = 0;
winning_attr = -1;
build_attribute_classes(): build $c_a$ for each $A$
for $i$ in $n$ (not in $F$) do
    cur_score = COMPUTE_SCORE($F, i,$ &$cost_F$)
    if (cur_score < best_score) then
        winning_attr = $i$
        best_score = cur_score; $best\_cost_F = cost_F$
return $best\_cost_F$

COMPUTE_SCORE($F, attr, cost_F$)
complete = homog = 0.0; $L(M_F) = 0$
$L(I|M_F) = ((|F| + 1) * N)$: +1 for the repair attribute
for ($c_a$ in attribute_classes($F$)) do
    overlap_reps[]: common reps between $c_{xy}$ and $c_a$
    compute_overlap($F, attr, c_a$, &overlap_reps)
    complete += computeComplete($F, c_a$,overlap_reps)
    homog += compute_homogeneity($F, c_a$,overlap_reps)
    update_cost_$c_a$($F$, &$L(M_F)$,&$L(I|M_F)$)
$cost_F = L(M_F) + L(I|M_F)$
return (homog)
```

- $recall_{constraint} = \frac{\#correctedTuples}{\#totalErrorTuples}$

Precision measures repair correctness whereas recall captures repair completeness. Let e be the error rate measured as the fraction of cells in the relation that are made to have errors. We generated a relation I, where $|I| = N$ that satisfies $F : X \rightarrow Y$, where $|XY| = m$ and we injected a total of $e * (N * m)$ errors by modifying attribute values (in XY), creating an inconsistent relation. For data repairs, the #correctRepairs are those repairs that resolve the (true) injected errors. For constraint repairs, not all the tuples in I are inconsistent. The #correctedTuples are tuples that are resolved wrt F by adding the recommended attribute. If the recommended attribute A has a large domain (contains many distinct values), adding A to F may not only neatly separate the conflicting tuples, but may also separate consistent tuples. The #totalModifiedTuples are the number of tuples that have been separated by A, regardless of whether they were inconsistent.

Figures 1 and 2 measure the quality of the recommended data and constraint repairs, by reporting the precision and recall, respectively. For these tests, we set $N = 50k$ for a single F with $m = 4$, $\theta = 0.01$ and $\beta = 0.7$. From Figure 1, we observed that as e increases, the precision for data repairs moderately decreases, as expected. The constraint repair precision values increase, as the number of modified tuples decreases for increasing e. For small e, most of I is consistent and adding A separates these clean tuples. As e increases, there are more inconsistent tuples (likely with low frequency) such that adding A does not cleanly distinguish these inconsistencies. We note that since we are working with synthetic data containing randomly generated data values, we are not able to leverage natural correlations among the attribute values that may exist in real data.

Figure 2 shows the recall values for data repairs are fairly high demonstrating that we are able to capture the majority of the true errors in the data. The recall values for the constraint repairs decline as e increases as it becomes increasingly difficult to find an attribute with a distribution of values that matches the random distribution of errors in I. Similar to precision, this is an artifact of working with the synthetic TPC data generator. We investigate the quality of repairs in a case study using real data in Section VII-D.

B. Scalability

We evaluated the scalability of our data and constraint repair algorithms (both individually and together) using the TPC-H data generator. In particular, we measured the running time against the number of tuples, the error rate, and varying the number of constraints, and the number of attributes.

Number of Tuples. In Figure 3, we vary N from 40k to 200k and measure the running times. We fix $e = 0.03, \theta = 0.01, \beta = 0.7$, for a single F with $m = 4$. The running time of the constraint repair algorithm is higher than the data repair algorithm since we need to build the classes for each attribute, and compare them with the clusters wrt F. There is approximately a 7% overhead on top of the constraint repair running time to run both the data and constraint repair algorithms together. This overhead is primarily due to maintaining the data structures that record the repairs and their costs.

Error Rate. Figure 4 shows the running time as the error rate e increases. We fix $N = 50k, \theta = 0.01, \beta = 0.7$, with a single constraint F with $m = 4$. We observe that all the running times moderately increase, but the composition of the total time (data and constraint repairs together) changes as e increases. For smaller e values, the constraint repair process consumes a larger portion of the total running time since we need to build and compare the attribute classes. As e increases, although there are more errors, and more classes to compare against, the potential overlap of erroneous values minimizes the constraint repair overhead. However, there are an increased number of data repairs found, which increases the running time of the data repair algorithm.

Number of Constraints. We studied the impact of varying the number of constraints on the running time for $N = 50k, e = 0.03, \theta = 0.01, \beta = 0.6$, and $m \in [2, 4]$. Figure 5 shows that the constraint repair time scales linearly as the number of constraints increases due to the search for a suitable repair attribute. The data repair time grows more aggressively because of the extra processing required to handle updates that affect other constraints. We observed these interdependencies where constraints F with larger m, that share overlapping attributes with other constraints, are evaluated first. For subsequent constraints, the update from F helped to resolve the inconsistency in the problematic tuples.

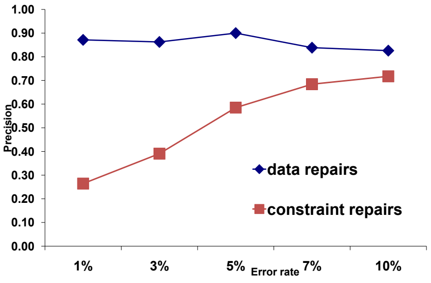

Fig. 1: Precision vs. error rate

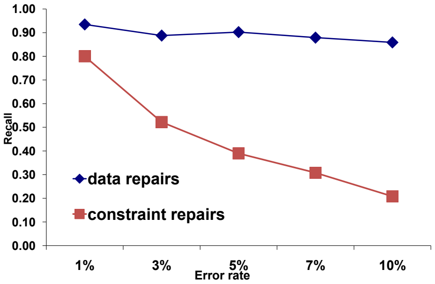

Fig. 2: Recall vs. error rate

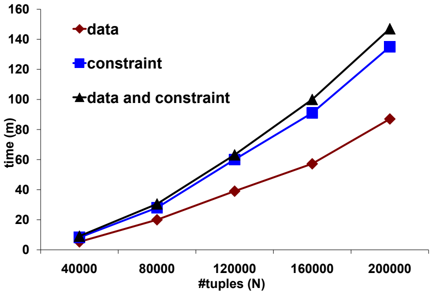

Fig. 3: Scalability wrt no. of tuples N

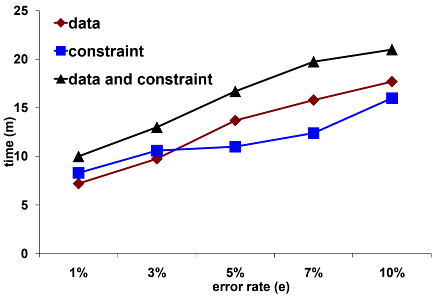

Fig. 4: Scalability wrt error rate e

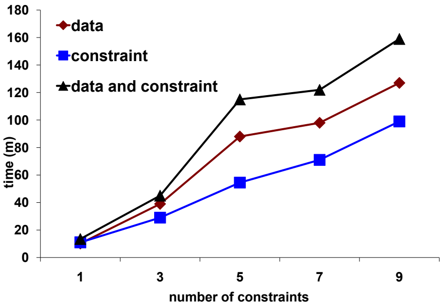

Fig. 5: Scalability wrt no. of constraints

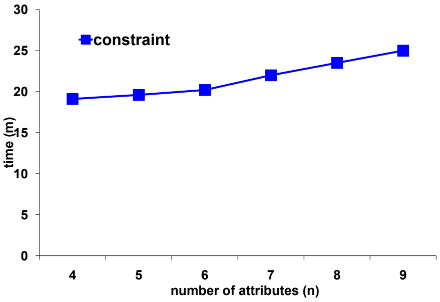

Fig. 6: Scalability wrt no. of attributes

Our running times compare favorably over the performance of GREEDY-REPAIR-FDFIRST (GF) [1] (on different systems, with approximately $50k$ tuples and 10 FDs); GF ran in 5hrs vs. our combined data and constraint repair algorithm ran in 2.7hrs.

Number of Attributes. We vary the number of attributes $n = |\mathbf{R}|$ when searching for a repair attribute in a constraint repair. We fix $N = 50k, e = 0.03, \theta = 0.01, \beta = 0.6$, for three constraints each having $m = 3$. The domain size of the attributes ranged from [4, 38600]. Figure 6 shows the running times of the constraint repair algorithm for $n \in [4, 9]$ (which includes the $m = 3$ attributes in F). The repair algorithm still builds the classes for all the attributes in I since each F has different attributes and together they span most of the attributes. We maintain a sorted index that is used to compare the attribute classes. The index minimizes unnecessary lookups, and avoids the worst case quadratic running time. As n increases, we observe that the running times moderately increase as expected.

C. Comparative Study

The most closely related work to ours is on data repairs. Bohannon et al. [1] propose a repair framework that updates data values based on accuracy and similarity. Accuracy is modeled based on a tuple's weight, and similarity is measured by record linkage techniques. Due to the large number of possible repairs, their approach for repairing FDs, like ours, is heuristic. They propose greedy algorithms that group X values (from the violated constraint) into classes and select the target Y value with minimal similarity cost. Their data repair algorithm considers modifications only to the consequent Y values, whereas we evaluate potential repairs to both X and Y values. While our work is in a similar spirit to this work, our cost model (and algorithms) were explicitly developed to permit us to compare the cost of data vs. constraint repairs, something that was not considered by Bohannon et al.

We evaluated our approach against the GF data repair algorithm [1] using the real 'Veterans of America' dataset [16]. This dataset contains demographic, socio-economic data of donors, and information on the frequency and amount of past and current donations. It is designed to assess the effectiveness of their marketing plan. The dataset consists of $N = 95412$ records and we use $n = 9$ attributes, with a schema of *(state, male-vet, num-promos, prev-hitrate, wealth-rating, income, edu, freq, amt)*. Continuous value attributes are transformed into categorical values. We postulated the following set of constraints:

- $F_1 : (freq, amt) \rightarrow (income)$
  The frequency, amount of a donation determines income.
- $F_2 : (prev\text{-}hitrate) \rightarrow (freq, amt)$
  How often donors replied to another organization determines their likelihood to donate now.
- $F_3 : (wealth) \rightarrow (amt)$
  A donor's wealth rating determines the amount donated.
- $F_4 : (income) \rightarrow (edu)$
  Income level determines education level.

We used $\theta = 0.2, \beta = 0.4$, and the FDs were processed in order of $F_1 - F_4$. We recommended constraint repair for F_2 and F_3, with the top-3 attributes, respectively, {*edu, income, wealth*}, and {*edu, prev-hitrate, state*}. Donors' education and income level (in addition to previous hit rate) influences their likelihood to donate. Similarly, education level and previous donations affect the amount donated wrt F_3. Interestingly, in F_3 the $wealth$ rating is state specific [16], coinciding with our $state$ attribute recommendation. Both F_2 and F_3 hold on approximately 25% of the data, and up to 20% and 45% of the (inconsistent) tuples (wrt F_2 and F_3, respectively), can be corrected by adding one of the recommended attributes to the constraint. Our recommendations give insight into data trends (e.g., direct marketing towards educated, philanthropic donors), which help to make the constraints not only more selective but more consistent with the data.

Our data repair algorithm considers the global impact of a data repair across all FDs, whereas GF makes local repair decisions involving the current inconsistent FD. For data repairs, we found that 68 of the total 130 data repairs were affected by data repairs from other FDs. For example, increasing the amt value, and increasing the $freq$ of a donation wrt F_1 (in 3303 tuples) helped to resolve inconsistencies in 3794 tuples wrt F_2. GF does not consider direct updates to amt based on F_1. However, since amt is in the consequent of F_2, and if this value is updated, it may indirectly cause an inconsistency in F_1. This could only be repaired, under their approach, by incorrectly updating $income$. Note that our data repair algorithm considered repairs to $income$ but determined that this was undesirable due to large update costs. The data repair for F_3 showed heavy skew (92% of the repairs) towards updating the $wealth$ value, indicating a constraint repair is needed, as expected. The GF algorithm would update the Y (amt) value (repair costs permitted), which would be incorrect in this case since $wealth$ is $state$ specific. Our repair algorithms propose repairs consistent with application trends in the data that previous techniques are unable to find.

D. Cora Case Study

We evaluated the quality of our data and constraint repairs using the well known bibliographic Cora dataset [17]. The dataset consists of $N = 1295$ records and $n = 13$ attributes with a schema of *(authors, volume, title, institution, venue, location, publisher, year, pages, editor, note, month, class)*. We postulate the following set of constraints:

- $F_1 : (venue, year) \rightarrow location$
  A *venue* and *year* is always held at the same *location*.
- $F_2 : (title, venue) \rightarrow authors$
  A paper *title* and *venue* determines the *authors*.
- $F_3 : (venue, location) \rightarrow editor$
  A *venue* held at *location* determines the *editor*.
- $F_4 : (venue) \rightarrow publisher$
  A given *venue* determines the *publisher*.

Our objectives are: (1) to evaluate the cost model and check the type of repairs recommended based on the inconsistencies in the data; and (2) to evaluate the quality of the repairs. We ran our data and constraint repair algorithms individually and together. At $\theta = 0.005$ and $\beta = 0.5$, the data repair and constraint repair algorithms ran in 2.5s and 1.6s, respectively, and together the total running time was 3.2s.

TABLE II: Description length values

FD	DL_{start}	DL_{data}	DL_F
F_1	3885	2949	3244
F_2	3885	2995	4752
F_3	3885	3528	2792
F_4	2590	3215	2673

1) Cost Model Evaluation: Table 2 shows the starting and final DL values based on the type of repair recommended. The constraints were evaluated in the order of F_2, F_4, F_1 and F_3. We see that data repairs are recommended for F_1 and F_2, and constraint repairs are recommended for F_3 and F_4, as expected. For F_1, $year$ and $location$ contain more standardized values, hence the majority of the inconsistencies lie in $venue$, and most of the data repairs are focused here. For F_2, the constraint repair algorithm recommends adding *pages* to uniquely identify a published paper. However, the cost to correct the $venue$ and $authors$ data values is cheaper.

The FDs F_3 and F_4 were chosen as constraints that are close but not completely correct. F_3 does not hold for journals. Most journals in the dataset contain empty $location$ values and may have multiple editors. Our repair algorithm recommends a constraint repair for F_3 using $volume$, as expected. Similarly, we recommend adding $editor$ to F_4 for $venues$ that may have changed $publishers$.

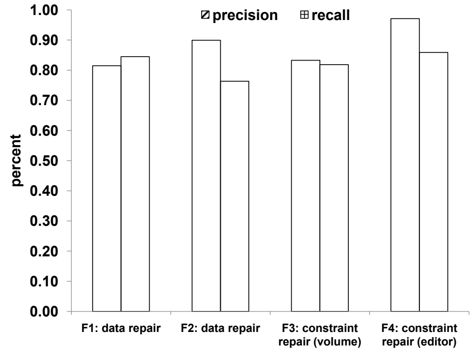

Fig. 7: Precision and recall (based on repair type) for each inconsistent constraint.

2) Quality Evaluation: Figure 7 shows the precision and recall values for each F based on the type of repair recommended. Overall, the precision values show that our repair algorithms do well in returning correct repairs. In particular, the data repairs for F_1 and F_2 not only focused on standardizing the names of venues and author names, but also fixing incorrect values. Table III gives some example repairs. The recall values for constraint repairs show that we are able to leverage natural correlations in the attribute values to resolve inconsistencies.

VIII. Related Work

Approaches for the management of inconsistent data include the definition and use of soft or relaxed constraints [18], [19],

TABLE III: Example repairs

F_1: (New York $\rightarrow$ Seattle,WA) Correct the *location* of 21st ACM STOC conference.
F_2: (Combining Regression Estimates $\rightarrow$ The Strength of Weak Learnability) Update paper *title* of R. Schapire in Machine Learning 5(2), '90.
F_3: (Advances in NIPS, san mateo, CA.) $\rightarrow$ (touretzky, d., mozer, m., hasselmo, m.) $\rightarrow$ (steven hanson, jack cowan, lee giles) Add attribute 'volume' to F_3 (v8 and v5) to distinguish editors. (journal of the ACM) $\rightarrow$ (F. Thomson Leighton) and (Joe Halpern) Distinguish via volumes 43(3) and 46(3), respectively.
F_4: (conf on compute learning theory (COLT) $\rightarrow$ (springer-verlag) and (acm press) Distinguish publisher by adding *editor* or *year*. In this case, (Kivinen, Jyrki and Sloan, Robert H., 2002), and (Shai Ben-David, Phil Long, 1999), respectively.

and conditional constraints [20], used in query processing and data cleaning. All still require some notion of satisfiability. While we have considered repairing FDs, a similar model could be used to ensure data continues to satisfy these more relaxed constraints. Additionally, a number of approaches discover constraints from the data [4], [8], [10], [5], [6], which are not only expensive to compute, but do not consider (modifying) the existing set of constraints that were known to hold for the application at one time.

In addition to the data repair work of Bohannon et al. [1] (considered in our evaluation Section VII-C), recent work by Beskales et al. [21] investigate *cardinality-set-minimal* data repairs, that balance the requirements of minimal data changes (cardinality) and necessary changes (set minimality). Their algorithm randomly samples from the space of possible data repairs. While our work is in a similar spirit to both these pieces of work, our cost model (and algorithms) were explicitly developed to permit us to compare the cost of data vs. constraint repairs, something that was not considered previously.

A data repair algorithm that resolves violations of conditional functional dependencies (CFDs) is given by Cong et al. [2]. This work extends the repair algorithm for FDs by Bohannon et al. [1] by considering updates to both the X and Y attributes, however preference is given to Y repairs. Hence, their repair model can exclude updates to attributes in X that may be better overall. While our focus is on the more general problem of FD repair, our data repair algorithm considers a forward check of the cost impact of each update on other dependencies.

Recent work in CFDs and data quality [10] propose a discovery algorithm that finds exact and approximate CFDs that hold over I. The approximate CFDs are used to suggest target values for potentially dirty data values. However, the relevant constraints are discovered based on the data (and are not given *a priori* as in our approach). Kolahi et al. [3] investigate the complexity of finding optimal data repairs using variables (called V-repairs) when functional dependencies are violated. For a given set of violated FDs, the authors present an approximation algorithm that finds V-repairs within a constant factor of the optimal repair.

IX. Conclusions and Future Work

We have presented the first algorithm for constraint repair that considers and compares modifications to the data and modifications to the constraints on an equal footing. We consider the most commonly used constraint, functional dependencies. Our results provide a foundation for constraint maintenance, one that does not overfit constraints to real data that may contain errors. As such our repaired constraints maintain their value in helping to ensure data quality. We are currently extending our work to handle other types of constraints including conditional functional dependencies, and to handle incomplete data. Missing values have a special semantics that should be considered within our cost model for both data repair and constraint repair.

References

[1] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi, "A cost-based model and effective heuristic for repairing constraints by value modification," in *SIGMOD '05*.

[2] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: consistency and accuracy," in *VLDB '07*.

[3] S. Kolahi and L. V. Lakshmanan, "On approximating optimum repairs for functional dependency violations," in *ICDT '09*.

[4] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, "Tane: An efficient algorithm for discovering functional and approximate dependencies," *Comput. J.*, vol. 42, no. 2.

[5] C. Wyss, C. Giannella, and E. L. Robertson, "Fastfds: A heuristic-driven, depth-first alg for mining fds from relations," in *DaWaK '01*, 2001.

[6] S. Lopes, J.-M. Petit, and L. Lakhal, "Efficient discovery of functional dependencies, armstrong relations," in *EDBT '00*.

[7] T. Dasu and T. Johnson, *Exploratory Data Mining and Data Clearning*, 2003.

[8] D. Wang, X. Dong, A. Sarma, M. Franklin, and A. Halevy, "Functional dependency generation and applications in pay-as-you-go data integration systems," in *WebDB '09*.

[9] I. Boydens, E. Zimanyi, and A. Pirotte, "Managing constraint violations in administrative information systems," in *Conference on Data Semantics*, 1997.

[10] F. Chiang and R. J. Miller, "Discovering data quality rules," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1166–1177, 2008.

[11] W. Fan, S. Ma, Y. Hu, J. Liu, and Y. Wu, "Propagating functional dependencies with conditions," *PVLDB 08*, vol. 1, no. 1.

[12] J. Rissanen, "Modeling shortest data description," in *Automatica*, 1978.

[13] T. Cover and J. Thomas, "Elements of information theory."

[14] W. E. Winkler, "The state of record linkage and current research problems," Statistical Division, U.S. Census Bureau, Tech. Rep., 1999.

[15] M. Meilă, "Comparing clusterings—an information based distance," *J. Multivar. Anal.*, vol. 98, no. 5, pp. 873–895, 2007.

[16] "Veterans of america dataset: http://mlr.cs.umass.edu/ml/databases/kddcup98/."

[17] M. Bilenko and R. Mooney, "Riddle: Repository of information on duplicate detection, record linkage, and identity uncertainty." [Online]. Available: http://www.cs.utexas.edu/users/ml/riddle

[18] A. Jha, V. Rastogi, and D. Suciu, "Query evaluation with soft-key constraints," in *PODS '08*, 2008, pp. 119–128.

[19] N. Dalvi and D. Suciu, "Management of probabilistic data: foundations and challenges," in *PODS '07*, 2007, pp. 1–12.

[20] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for data cleaning," in *ICDE '07*, 2007, pp. 746–755.

[21] G. Beskales, I. Ilyas, and L. Golab, "Sampling the repairs of functional dependency violations under hard constraints," in *PVLDB '10*.