A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification

Philip Bohannon
Lucent Technologies—Bell Laboratories
bohannon@research.bell-labs.com

Wenfei Fan*
Univ. of Edinburgh & Bell Labs
wenfei@inf.ed.ac.uk

Michael Flaster
Lucent Technologies—Bell Laboratories
mflaster@research.bell-labs.com

Rajeev Rastogi
Lucent Technologies—Bell Laboratories
rastogi@research.bell-labs.com

Abstract

Data integrated from multiple sources may contain inconsistencies that violate integrity constraints. The *constraint repair problem* attempts to find "low cost" changes that, when applied, will cause the constraints to be satisfied. While in most previous work repair cost is stated in terms of tuple insertions and deletions, we follow recent work to define a database repair as a set of *value modifications*. In this context, we introduce a novel cost framework that allows for the application of techniques from record-linkage to the search for good repairs. We prove that finding minimal-cost repairs in this model is NP-complete in the size of the database, and introduce an approach to heuristic repair-construction based on equivalence classes of attribute values. Following this approach, we define two greedy algorithms. While these simple algorithms take time cubic in the size of the database, we develop optimizations inspired by algorithms for duplicate-record detection that greatly improve scalability. We evaluate our framework and algorithms on synthetic and real data, and show that our proposed optimizations greatly improve performance at little or no cost in repair quality.

1. Introduction

When overlapping or redundant information from multiple sources is integrated, inconsistencies or conflicts in the data may emerge as violations of *integrity constraints* on the integrated data (see, e.g., [1, 3, 4, 5, 7, 10, 13, 27]). One important example of this situation is in the enterprise, where different departments such as sales, billing, and order- or service-fulfillment often have separate applications storing overlapping data. Conflicts in this data may be introduced for many reasons [24], including misspellings or differing conventions used during data entry (e.g., a person's name may appear as "John Smith" and "J. Smith"), different processes and time-scales for performing updates (e.g., address changes may take a few days to a few months to propagate), and so on. This problem becomes particularly evident with data warehousing or other integration scenarios because combining data makes conflicts visible: errors in a single database can seldom be detected without inspection of the real world or other manual effort. Yet the consequences of poor enterprise data can be severe—for telecommunication service providers, for instance, errors routinely lead to problems like failure to bill for provisioned services, delay in repairing network problems, unnecessary leasing of equipment, and so on [22]. As a result, data sources may be integrated in order to *reconcile* and correct the source data. For example, *revenue recovery* applications [18, 19] compare billing and service databases to ensure that all services are billed (and presumably vice-versa). We now introduce our running example to illustrate these issues.

* Supported in part by EPSRC GR/S63205/01, GR/T27433/01, and NSFC 60228006.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

Example 1: Consider a hypothetical provider of network services to residential users (e.g., a phone or cable company). Customer and equipment information is maintained by separate databases in the Billing and Maintenance departments. Data from these two databases is merged according to the following target schema with two tables, cust and equip:

cust(<u>phno</u>, name, street, city, state, zip)
equip(phno, <u>serno</u>, eqmfct, eqmodel, instdate)

The cust table contains address information on customers with phone number as a key, while the equip table catalogs equipment installed at the customer's location and includes manufacturer, model number, install date and the serial number which serves as the key. Figure 1 depicts an example instance, $\mathcal{D}$, of the cust and equip tables. Tuples are labeled as $t_0, t_1, \ldots$ for ease of reference, and tuples t_0, t_1, t_5, t_6 are from the Billing database.

Figure 1 also shows the set $\mathcal{C}$ of Inclusion Dependencies (referred to as IND (i)) and Functional Dependencies (referred to as FD (i)) on $\mathcal{D}$. For example, IND (1) ensures that every piece of equipment is associated with a valid customer in cust, while FDs (2) and (5) are key dependencies specifying that phone number and serial number are keys for the customer and equipment tables, respectively. FD (6) is not a traditional key dependency, but asserts (perhaps somewhat arbitrarily) that a given customer will have only one instance of a given piece of equipment.

The *wt* column in the figure does not appear in the original data, and instead reflects the confidence placed by the user in the accuracy of the data. In this example, a greater confidence is placed in records from Billing.

An example of a source database inconsistency is differing spellings for "Alice Smith" in tuples t_0 (from Billing) and t_4 (from Maintenance), which violates FD (2). Other constraint-violating discrepancies include a) the tuples t_1 and t_2 for Bob Jones with different phone numbers and states which violate FDs (3) and (4), and b) the tuple t_9 which violates the inclusion dependency IND (1) between the equip and cust tables. □

While substantial previous work has explored query answering and constraint repair in inconsistent databases, the bulk of that work [1, 3, 4, 5, 7, 13, 14] restricts repair actions to inserting and deleting tuples. However, in these models, repairs of inclusion de-

cust

	phno	name	street	city	state	zip	*wt*
t_0	949-1212	Alice Smith	17 bridge	midville	az	05211	*2*
t_1	555-8145	Bob Jones	5 valley rd	centre	ny	10012	*2*
t_2	555-8195	Bob Jones	5 valley rd	centre	nj	10012	*1*
t_3	212-6040	Carol Blake	9 mountain	davis	ca	07912	*1*
t_4	949-1212	Ali Stith	27 bridge	midville	az	05211	*1*

equip

	phno	serno	eqmfct	eqmodel	instdate	*wt*
t_5	949-1212	AC13006	AC	XE5000	Mar-02	*2*
t_6	555-8145	L55001	LU	ze400	Jan-03	*2*
t_7	555-8195	L55011	LU	ze400	Mar-03	*1*
t_8	555-8195	AC22350	AC	XE5000	Feb-99	*1*
t_9	949-2212	L32400	LU	ze400	Oct-01	*1*

Inclusion Dependencies:
(1) equip[phno] $\subseteq$ cust[phno]

Functional Dependencies:
(2) cust[phno] $\rightarrow$ cust[name, street, city, state, zip]
(3) cust[zip] $\rightarrow$ cust[city, state]
(4) cust[name, street, zip] $\rightarrow$ cust[phno]
(5) equip[serno] $\rightarrow$ equip[phno, eqmfct, eqmodel, instdate]
(6) equip[phno, eqmfct, eqmodel] $\rightarrow$ equip[serno]

Figure 1: Customer Equipment Example Data $\mathcal{D}$, and Dependencies $\mathcal{C}$.

pendencies may lose important information. For example, deletion of tuple t_9 to repair the violation of inclusion dependency ID (1) in Figure 1 would lose information about a piece of equipment, and inserting a new tuple in the cust table would not help the user resolve the location of that equipment accurately. Recent work [10, 27] has introduced repairs in which attribute values are *modified* to restore the database to a consistent state, allowing more satisfying resolution of common constraint violations.

Record linkage is a broad field (see, e.g., [9, 20, 28]), also known as "duplicate removal" or "merge-purge", and refers to the task of linking pairs of records that refer to the same entity in different data sets. This is commonly applied to household information in census data, mailing lists or medical records as well as many other uses. While to our knowledge not observed by prior work, there are in fact strong connections between record linkage and constraint repair. First, individual repairs of Inclusion and Functional Dependencies involve entity matching. Consider IND (1) in the above example: repairing this constraint requires matching customer entities, in this case represented by their phone numbers. If this constraint were specified on several attributes including name and address information, the task of finding the appropriate repair for an unmatched tuple in the equipment table would exactly correspond to a record linkage task. Second, specific record linkage tasks for a set of tables can be accomplished by specifying inclusion and functional dependencies and then invoking constraint repair. Consider the task of removing approximate duplicates between tables R and S. This may be accomplished by specifying a pair of inclusion constraints from R to S and back on the set of attributes which should match (perhaps all the attributes).

While there is a compelling need to help users correct conflicting data, it may be difficult to see what sort of automatic support will be helpful. Clearly, it would be helpful to a user to enumerate constraint violations, but it may still be exceedingly onerous for the user to manually correct all the problems. In this situation, it may be more helpful to automatically propose a *repair* [1, 2, 7], which can be informally thought of as a database that is "close" to the original but which satisfies integrity constraints.

Our Contributions. The first contribution of this paper is a *repair framework* that focuses, like [10, 27], on value modification, but which improves on previous work in several ways. First, we offer a general framework that deals with both FDs and INDs, unlike [10], which focuses on the specific domain of census data, and [27], in which cleaning is restricted to relatively simple constraints that cannot express INDs, for example. Furthermore, we propose a novel *cost* model for repairs based on two factors, accuracy and similarity, which we now define. The *accuracy* of data is reflected in a weight $w(t)$ for each tuple and represents the confidence placed by the user in the values therein. For example, the tuples in Fig. 1 from the Billing department are given weight 2, reflecting greater confidence in their accuracy than records from the Maintenance department, which are given a weight 1. In this work, we assume that the weights default to 1 but may be given by the user, and leave to future work the development of techniques for automatically setting them. A variety of *similarity* of data is available at the attribute or tuple level, for example, string-edit distance. While we use a relatively simple similarity function in our experiments, our goal is to allow a variety of attribute-similarity metrics (see [8, 28]) from the data linkage community to be applied.

Our second contribution consists of *complexity results* for minimum-cost constraint repair based on value-modification. We show that value-modification complicates the analysis of the problem: it becomes NP-complete in the size of the data, even with a small, *constant* number of either FDs or INDs. In contrast, the corresponding problems in which a database is repaired by deleting a minimal set of tuples is in PTIME; the problem only becomes intractable if arbitrary FDs and INDs are both present [3, 7]. In this context, we find that simple heuristic repair approaches based on repairing one constraint at a time suffer from a number of problems including a failure to terminate.

In light of the intractability results and problems with simple heuristics, our third contribution is an approach to repair construction based on *equivalence classes* of (tuple, attribute) pairs that are assigned identical values in the repair. The introduction of equivalence classes has three benefits. First, it is fundamental to ensuring termination of the algorithm, even when tuple inserts are allowed. Second, it separates the relationship among attribute values (represented by the equivalence classes) from the choice of value assigned to each attribute in the suggested repair (dictated by a cost metric). This separation has the potential to improve value selection by delaying it until a larger group of related values is known—for example, the phone number "555-8145" is the highest weighted value of all the numbers for Bob Jones, while locally in the equip table "555-8195" may look like a better choice. Third, equivalence classes potentially ease user interaction by illustrating the relationships between parts of a proposed repair and allowing the user to validate updates one equivalence-class at a time.

In this framework, we first consider straightforward use of the greedy method. In the example of Figure 1, our repair procedure will group the phone-number attributes of t_1 and t_2 in a single equivalence class, and pick one of the values as the value proposed in the repair. We consider different cost models for the greedy step, as well as a variant of the greedy method which resolves FDs aggressively. While naive implementations of our greedy methods require time cubic in the size of the data in practical cases, we

introduce two optimizations, one relaxing the greedy method by sometimes taking a step which is not locally optimal, and another inspired by optimizations for duplicate elimination [15]. Together and under reasonable assumptions, these optimizations improve our algorithm's running time to $O(n \lg^2 n \cdot |\mathcal{C}|^2)$, where n is the size of the database including inserted tuples, and $\mathcal{C}$ is the set of constraints to be enforced.

Our fourth and final contribution is an *experimental study* of our heuristic constraint repair methods. We evaluate the quality and scalability of our methods with both synthetic data from the TPC-H benchmark and real-world data scraped from Web pages. We further evaluate the effectiveness of our two optimizations on running time and result quality. We find that the optimizations seldom degrade quality, but improve the performance significantly and hence allow the approach to be applied to real-world size problems. Based on these results, we contend that heuristic constraint repair based on tuple-attribute equivalence classes is a promising tool for constraint repair in data integration and reconciliation scenarios.

Organization. In the next section, we introduce our repair model. We prove intractability results in Section 3 and introduce our heuristic approach in Section 4, giving detailed algorithms in Section 5. We present our experimental results in Section 6, discuss related work in Section 7 and conclude in Section 8.

2. System Model and Problem Formulation

In this section, we present our cost-based constraint repair problem formulation. In our model, each database instance $\mathcal{D}$ (equivalently "database") contains a fixed set of tables $R_1, \ldots, R_n$ where table R_i is defined over a set of attributes $\mathsf{attr}(R_i)$. Each tuple t is associated with a table R_i and a weight $w(t) \geq 0$. Note that weights are assigned at the tuple level to simplify the presentation, but in practice attribute-level weights may be preferable. To simplify the discussion we assume that one can keep track of a given tuple t in R_i during the constraint repair process despite arbitrary attribute value changes (using a temporary unique id, for example). We use $\mathcal{D}(t, A)$ to denote the value of a given attribute $A \in \mathsf{attr}(R_i)$ of t in some database $\mathcal{D}$. This value is drawn from $\mathsf{dom}(A)$, the domain of A, plus the special value null. Further, for a subset X of attributes from $\mathsf{attr}(R_i)$, we use $\mathcal{D}(t, X)$ to represent the projection of t on attributes in X.

Constraints. We consider the following two types of constraints:

1. FUNCTIONAL DEPENDENCIES (FDs). Each functional dependency has the form $\mathsf{R[X]} \to \mathsf{R[Y]}$, where X and Y are subsets of attributes from $\mathsf{attr}(R)$. A database $\mathcal{D}$ is said to satisfy the FD $\mathsf{R[X]} \to \mathsf{R[Y]}$ if for every pair of tuples $t_1, t_2 \in R$ such that $\mathcal{D}(t_1, X) = \mathcal{D}(t_2, X)$, it is the case that $\mathcal{D}(t_1, Y) = \mathcal{D}(t_2, Y)$.
2. INCLUSION DEPENDENCIES (INDs). Inclusion dependencies have the form $\mathsf{R_1[X]} \subseteq \mathsf{R_2[Y]}$, where X and Y are lists of attributes (with the same cardinality) from $\mathsf{attr}(R_1)$ and $\mathsf{attr}(R_2)$, respectively. A database $\mathcal{D}$ is said to satisfy the IND $\mathsf{R_1[X]} \subseteq \mathsf{R_2[Y]}$ if for every tuple $t_1 \in R_1$ there exists a tuple $t_2 \in R_2$ such that $\mathcal{D}(t_2, Y) = \mathcal{D}(t_1, X)$.

A database $\mathcal{D}$ satisfies a constraint set $\mathcal{C}$ of FDs and INDs if it satisfies every constraint in $\mathcal{C}$.

Database Repairs. We now formally define the notion of a *database repair* introduced above. A repair of a database $\mathcal{D}$ is a database $\mathcal{D}'$ such that 1) tuples appearing in $\mathcal{D}$ are carried over to $\mathcal{D}'$ (identified by, e.g., id), *possibly with modified attribute values*, 2) zero or more *inserted tuples* appear in $\mathcal{D}'$ but not in $\mathcal{D}$, and 3) $\mathcal{D}'$ satisfies the constraint set $\mathcal{C}$. For convenience, we refer to the inserted tuples appearing in table R_i in $\mathcal{D}'$ as $\mathbf{new}(R_i)$.

Intuitively, an inconsistent database may be neither sound nor complete [3], and thus our model supports both value modifications and tuple insertions. We modify the values of tuples in $\mathcal{D}$ rather than simply deleting them as in other models (e.g., [7]) in order to minimize loss of information.

Repair Cost. The cost of an attribute-level modification in a repair is essentially the weight $w(t) \geq 0$ of the changed tuple times the *distance* according to a similarity metric between the original value of the attribute and its value in the repaired database. Similarity measurement for strings and other structured values is itself a broad field (see, e.g., [8]), and our setting does not depend on a particular approach. Rather, we assume that for two values v, v' from the same domain, a *distance* function $\mathsf{dis}(v, v')$ is available, with lower values indicating greater similarity. A common distance function for strings (the Damerau-Levenshtein or D-L metric [12]) is defined as the minimum number of single-character insertions, deletions and substitutions required to transform v to v'. We use this metric in the examples below and a similar metric in the experiments.

Finally, we assume that a cost $\mathsf{inscost}(R_i) > 0$ is associated with each table R_i, which is the cost of inserting tuples into R_i in $\mathcal{D}'$. This cost is a user-defined parameter closely related to the threshold set for a good match by the similarity metric. We find that an effective setting for inscost is a value slightly higher than the distance between two "similar" strings (see Section 6.1 for details).

In our examples, the cost of a repair is the sum of the cost of the tuples in the repair. To summarize

$$\mathsf{cost}(t) = \begin{cases} \mathsf{inscost}(R_i) & \text{if } t \in \mathbf{new}(R_i) \\ w(t) \cdot \sum_{A \in \mathsf{attr}(R_i)} \mathsf{dis}(\mathcal{D}(t, A), \mathcal{D}'(t, A))) & \text{otherwise} \end{cases}$$

For instance, consider t_2 in the repair described at the end of Example 1. Given $w(t_2) = 1$ and string edit distances of 1 both from"555-8145" to "555-9145" and from "nj" to "ny" we get $\mathsf{cost}(t_2) = 1 \cdot (1 + 1) = 2$, while if t_1's phone number and state had been modified instead, we would have had $2 \cdot (1+1) = 4$ since $w(t_1) = 2$. The cost of the repair $\mathcal{D}'$ of database $\mathcal{D}$ is defined as $\mathsf{cost}(\mathcal{D}') = \sum_{t \in \mathcal{D}'} \mathsf{cost}(t)$.

We are now ready to formally state our database cleaning problem in terms of computing a minimum-cost repair.

Problem Statement. Given a database $\mathcal{D}$ comprising tables $R_1, \ldots, R_n$ and a set of constraints $\mathcal{C}$ defined on them, find the repair $\mathcal{D}'$ of $\mathcal{D}$ for which $\mathsf{cost}(\mathcal{D}')$ is minimum. □

3. Constraint Repair Approach Overview

In this section, we investigate solutions to the constraint repair problem defined in the last section. We begin by outlining our approach to repairing individual constraint violations. We then consider the problem of finding minimum-cost repairs, but show that optimal solutions are generally intractable to find.

3.1 Constraint Repairs

In general, it is useful to think of a database repair $\mathcal{D}'$ as the result of some modifications to database $\mathcal{D}$ as illustrated by the following example.

Example 2: We present a set of modifications that together constitute a possible repair of the constraint violations discussed in Example 1.

1. Tuple t_2: Modify phone number to "555-8145" repairing

FD (4) and state to "ny" (repairing FD (3)), in both cases by matching t_1.

2. Tuple t_4: Modify name to "Alice Smith" and street to "17 bridge" (repairs FD (1) by matching t_1).

3. Tuple t_7: Modify phone number to "555-8145" (repairs IND (1)), serial number to "L55001" (repairs FD (6)) and installation date to "Jan-03" (repairs FD (5)).

4. Tuple t_8: Modify phone number to "555-8145" (repairs IND (1)).

5. Tuple t_9: Modify phone number to "949-1212" (repairs IND (1) by matching t_4). □

For a functional dependency $F = \mathsf{R}[\mathsf{A}] \rightarrow \mathsf{R}[\mathsf{B}]$ over attributes A and B of table R, consider a pair of tuples (t, t') in R that violate F, that is, $\mathcal{D}(t, A) = \mathcal{D}(t', A)$, but $\mathcal{D}(t, B) \neq \mathcal{D}(t', B)$. In this case, we can resolve this constraint violation by setting the B-attribute value of t to be equal to t' (or vice versa) in the repair $\mathcal{D}'$. This is illustrated in step 1 of Example 2, where t_2 is modified to match t_1. Note that it is also possible to fix the FD by setting the value of attribute A in tuple t_1 to be different from the A-attribute value in tuple t_2. We do not consider this option for FD repair because it is unclear as to what (different) value should be assigned to tuple t_1's A attribute, and moreover, when the FDs are keys, it may lead to insertions of entities that are not meaningful. For example, to repair the violation of FD (3) in the first step, we could have made up a new zip code for either t_1 or t_2, but there does not exist a small reasonable set of candidates from which to choose.

Similarly, INDs can be repaired by modifying attribute values. For example, if a tuple $t_1 \in R_1$ does not satisfy IND $I = \mathsf{R}_1[\mathsf{A}] \subseteq \mathsf{R}_2[\mathsf{B}]$, then we can modify t_1's A-attribute value so that it is equal to the B-attribute value for some tuple in table R_2. Alternately, we can consider modifying the B-attribute value for some tuple in R_2 so that it is equal to t_1's A-attribute value. Step 4 above illustrates such a correction. Here, when the phone numbers in t_4 and t_9 are similar, it is likely that one or the other is correct. It also seems clear from this example that an attribute-modification cost model is preferable to one based on tuple insertion and deletion: the violation of IND (1) can be repaired with the deletion of t_9, presumably with minimal cost in a tuple-cost model. However, this seems to lose important information in this situation.

Finally, note that if no similar $R_2[B]$ value exists for some unmatched tuple t from R_1, inserting a tuple $t^{\mathbf{new}}$ in R_2 (in $\mathcal{D}'$) may be preferable to modifying t. In this case, the B attribute(s) of $t^{\mathbf{new}}$ are set to match t_1's A attribute(s) (in $\mathcal{D}'$), and all other attribute values are set to the special value null.

A subtle issue arises from the null value. The SQL standard [16] supports three different semantics for comparing the values of $\mathcal{D}(t_1, X_1)$ and $\mathcal{D}(t_2, X_2)$ which may involve null, where X_1, X_2 are sequences of attributes. (1) The *simple* semantics defines $\mathcal{D}(t_1, X_1) = \mathcal{D}(t_2, X_2)$ to be true if *either one* of them contains null. (2) The *partial* semantics evaluates $\mathcal{D}(t_1, X_1) = \mathcal{D}(t_2, X_2)$ to true if each *non-*null *value* in $\mathcal{D}(t_1, X_1)$ equals its corresponding value in $\mathcal{D}(t_2, X_2)$ and vice versa. (3) The *full* semantics evaluates $\mathcal{D}(t_1, X_1) = \mathcal{D}(t_2, X_2)$ to false if *either one* of them contains null. While the SQL standard does not explicitly support arbitrary FDs and INDs, it allows one to use any of the three semantics when dealing with *unique* and *referential* constraints, which are (special cases of) FDs and INDs, respectively. In the sequel we assume *the partial semantics* when null is involved. Note that this semantics allows null to participate in comparisons of attribute values.

3.2 Minimum-Cost Repair (Intractability Results)

We present two intractability results for min-cost database repair, showing that the problem is NP-complete even for a small, fixed number of only FD or only IND constraints.

Theorem 1: *Let $\mathcal{C}$ be a set of only FD or only IND constraints defined on database $\mathcal{D}$. Then, for a constant W, the problem of determining if there exists a repair of $\mathcal{D}$ whose cost is at most W is NP-complete.* □

Proof Sketch: The proofs are by reductions from the *vertex cover* problem (even when $\mathcal{C}$ contains only FDs), and from the *3-dimensional matching* problem (even when $\mathcal{C}$ contains only INDs). Each reduction uses only a *constant* number of constraints (either FDs or INDs). Proofs are omitted due to space constraints, and can be found in the full version of the paper. □

Interestingly, the corresponding repair problems (when $\mathcal{C}$ contains only FDs or only INDs) are shown to be tractable for a delete-only repair model by [7]. This demonstrates that the repair problem becomes much more difficult when we consider value modifications.

4. Using Equivalence Classes for Constraint Repair

In light of Theorem 1, we necessarily consider heuristic approaches to constraint repair. A particular heuristic algorithm will take as input a database $\mathcal{D}$ and a set $\mathcal{C}$ of constraints defined on $\mathcal{D}$, and find a repair $\mathcal{D}'$ of $\mathcal{D}$. It should be able to find $\mathcal{D}'$ efficiently, with the tradeoff that $\mathsf{cost}(\mathcal{D}')$ is not necessarily minimum. In fact, we found it non-trivial to develop such a heuristic. The key difficulty, of course, is that repairing one constraint can break another, and most simple heuristics we considered could fail to terminate in the presence of complex, inter-related dependencies.

4.1 Equivalence Classes

To overcome these problems, our approach to constraint repair for FDs and INDs is built around the notion of *equivalence classes* of attribute value coordinates (t, A), where t identifies a tuple in a table R in which A is an attribute. The semantics of an equivalence class of (t, A) pairs is that the tuple attributes contained in the class are assigned the same value in $\mathcal{D}'$. (We assume that all such attributes in an equivalence class have the same domain). Our motivation for considering equivalence classes is that both FD and IND constraints can be seen as specifying equivalence between certain sets of attribute coordinates. For example, an FD $R[X] \rightarrow R[Y]$ essentially specifies that if a pair of tuples t_1, t_2 in R matches on the attribute set X, then (t_1, A) and (t_2, A) must be in the same equivalence class for all $A \in Y$. Similarly, for an IND $R_1[X] \subseteq R_2[Y]$, we require that each tuple $t_1 \in R_1$ is covered by some tuple $t_2 \in R_2$, or alternately, (t_1, A) and (t_2, B) are in the same equivalence class for each attribute A in X and the corresponding attribute B in Y.

A key observation here is that it is useful to separate the decision of which attribute values need to be equivalent from the decision of exactly what value should be assigned to the eventually-produced equivalent set. Delaying value assignment allows poor local decisions to be improved—for example, consider a name that is sometimes spelled correctly and sometimes incorrectly. If the correct spelling is more frequent and/or has higher weight, then the accumulation of versions of the name in an equivalence class over time will allow the correct spelling to be chosen in many cases. Further, we believe that the equivalence class abstraction will be valuable

Procedure FD-RESOLVE-TUP (S, F)
Input: Set of tuples S that match on attribute set X,
FD $F = R[X] \rightarrow R[Y]$.
begin
1. **for each** attribute A in Y **do** {
2. $eq_A := \cup_{t \in S} \mathsf{eq}(t, A)$;
3. $\mathcal{E} := (\mathcal{E} - \{\mathsf{eq}(t, A) : t \in S\}) \cup \{eq_A\}$;
4. }

end

Figure 2: Resolving Set of Tuples S for FD F.

to a user who needs to check or modify a repair. The classes help expose the structure of data relationships, and if the user wants to override a value chosen by the repair algorithm, it can be accomplished on the *whole equivalence class in one step*.

An equivalence class eq is a set of tuple, attribute pairs (t, A) Our repair algorithm maintains a global set of equivalence classes $\mathcal{E}$ that covers $\mathcal{D}'$ (that is, the tuples in the original database $\mathcal{D}$ plus insertions). For a given pair (t, A), $\mathsf{eq}(t, A)$ returns the current equivalence class containing (t, A) in $\mathcal{E}$. Associated with each class eq is a "target value" $v = \mathsf{targ}(eq)$. The target value is fundamental to the construction of the database repair $\mathcal{D}'$, since $\mathcal{D}'(t, A)$ is defined as $\mathsf{targ}(\mathsf{eq}(t, A))$. Thus, all attributes in a class eq are assigned the value of $\mathsf{targ}(eq)$ in the repair.

Equivalence Class Cost. The cost of the equivalence class for a particular target value v is defined as the contribution of elements in the equivalence class to the cost of $\mathcal{D}'$ (ignoring the cost of inserts); that is, $\mathsf{cost}(eq, v) = \sum_{(t,A) \in eq} w(t) \cdot \mathsf{dis}(v, \mathcal{D}(t, A))$. Consistent with the goal of finding a low-cost repair, $v = \mathsf{targ}(eq)$ is chosen to minimize the cost of eq, and unless specified otherwise, $\mathsf{cost}(eq)$ is simply the minimum $\mathsf{cost}(eq, v)$ over some universe of potential v values, such as the values taken by elements of eq in $\mathcal{D}$. As an example, in the database shown in Fig. 1, for $eq = \{(t_1, \mathsf{phno}), (t_2, \mathsf{phno})\}$, $\mathsf{cost}(eq, \text{“}555 - 8145'') = 1 \cdot 1$ while $\mathsf{cost}(eq, \text{“}555 - 8195'') = 2 \cdot 1$. Thus, the value $\mathsf{targ}(eq)$ is "555-8145", and $\mathsf{cost}(eq) = 1$.

Merging Equivalence Classes. Whenever two equivalence classes are merged, this may result in additional attribute modifications in $\mathcal{D}'$, increasing its cost. For a subset E of equivalence classes from $\mathcal{E}$, we formalize this *increase in cost* as $\mathsf{mgcost}(E) = \mathsf{cost}(\cup_{eq \in E} eq) - \sum_{eq \in E} \mathsf{cost}(eq)$; that is, the difference between the cost of the merged class and the sum of the costs of the individual classes. For instance, referring back to Fig. 1, the cost of merging classes $eq_1 = \{(t_1, \mathsf{phno}), (t_2, \mathsf{phno})\}$ and $eq_2 = \{(t_7, \mathsf{phno})\}$ to form $eq_3 = \{(t_1, \mathsf{phno}), (t_2, \mathsf{phno}), (t_7, \mathsf{phno})\}$ is given by $\mathsf{mgcost}(\{eq_1, eq_2\}) = \mathsf{cost}(eq_3) - (\mathsf{cost}(eq_1) + \mathsf{cost}(eq_2)) = 2 - (1 + 0) = 1$.

4.2 Repairing Violations

We next discuss how individual constraint violations are repaired by *resolving* tuples.

Repairing an FD Violation. We say that a tuple t is *resolved w.r.t. an FD $F = R[X] \rightarrow R[Y]$* if, for all other tuples $t' \in R$, either $\mathcal{D}'(t, A) \neq \mathcal{D}'(t', A)$ for some $A \in X$, *or* for every $B \in Y$, $\mathsf{eq}(t, B) = \mathsf{eq}(t', B)$. Note that if t is resolved, it is not part of a violation in $\mathcal{D}'$, but the converse need not hold since for some $B \in Y$, (t, B) might have the same target value as (t', B) without (t, B) and (t', B) being in the same equivalence class. Clearly, a tuple $t \in R$ can *become unresolved* w.r.t. F due to a change in the target value of an attribute in X for some other tuple in R, a fact which we refer to as the *collision property* of FD resolution. This might happen, for example, due to changes in target values when equivalence classes merge.

While a violation can be explained in terms of pairs of tuples, we define the act of *resolving* a tuple t w.r.t. F in terms of a set S of tuples from R. Here S includes t and all other tuples that agree with t on (target values of) attributes in X. The procedure FD-RESOLVE-TUP shown in Fig. 2 shows how to resolve such a set S by merging, for each attribute A in Y, the equivalence classes $\mathsf{eq}(t, A)$ for $t \in S$. Accordingly, $\mathsf{rescost}(S, F)$, the *merge cost of resolving* S w.r.t. F, is the sum, for each attribute A in Y, of $\mathsf{mgcost}(\{\mathsf{eq}(t, A) : t \in S\})$. For example, in Fig. 1, the tuple set $\{t_1, t_2\}$ is resolved w.r.t. FD (4) by merging the classes $eq_1 = \{(t_1, \mathsf{phno})\}$ and $eq_2 = \{(t_2, \mathsf{phno})\}$. Thus, $\mathsf{rescost}(\{t_1, t_2\}, 4) = \mathsf{mgcost}(\{eq_1, eq_2\}) = 1$.

Procedure IND-RESOLVE-TUP (t, target, I)
Input: Tuple $t \in R_1$ to resolve, a target target which is either a
tuple $t' \in R_2$ or $\mathbf{new}$, IND $I = R_1[X] \subseteq R_2[Y]$.
begin
1. **if** $(\mathsf{target} = \mathbf{new})$ **then** {
2. $t' := \mathbf{new}$ null tuple in R_2 with 0 weight;
3. $\mathcal{E} := \mathcal{E} \cup \{\{(t', A)\} : A \in \mathsf{attr}(R_2)\}$;
4. }
5. **for each** attribute A in X and corresponding B in Y **do**
6. $\mathcal{E} := (\mathcal{E} - \{\mathsf{eq}(t, A), \mathsf{eq}(t', B)\}) \cup \{\mathsf{eq}(t, A) \cup \mathsf{eq}(t', B)\}$;

end

Figure 3: Resolving Tuple t for IND I.

Repairing an IND Violation. For an IND $I = R_1[X] \subseteq R_2[Y]$, a tuple t is said to be *resolved* with respect to I if there is some tuple $t' \in R_2$ such that (t, A) and (t', B) are in the same equivalence class for *every pair* of corresponding attributes $A \in X$ and $B \in Y$. It is easy to see that, in contrast to FDs, a tuple resolved w.r.t. an IND I will not become unresolved, a fact we refer to as the *permanency property* of IND resolution. Thus, by resolving all tuples w.r.t. INDs, we can ensure that no INDs are violated.

Tuple t is resolved by "covering" it with either a new or existing tuple $t' \in R_2$. Here a new tuple t' consists of null, i.e., $\mathcal{D}'(t', A) = \mathsf{null}$ for each attribute A of t'. This is accomplished by procedure IND-RESOLVE-TUP shown in Fig. 3. This procedure creates t' if required, and merges $\mathsf{eq}(t, A)$ with $\mathsf{eq}(t', B)$ for corresponding attributes A and B from X and Y respectively. Accordingly, the *cost of resolving t w.r.t. I using t'*, $\mathsf{rescost}(t, t', I)$, is the sum of the attribute-wise costs, $\mathsf{mgcost}(\{\mathsf{eq}(t, A), \mathsf{eq}(t', B)\})$ for corresponding attributes A and B, plus the insert cost of t' if it is new. (Note that if t' is new, then $\mathsf{rescost}(t, t', I)$ is simply $\mathsf{inscost}(R_2)$ since t' is assigned 0 weight.) For example, in Fig. 1, tuple t_7 can be resolved w.r.t. IND (1) by merging the classes $eq_1 = \{(t_1, \mathsf{phno})\}$ and $eq_2 = \{(t_7, \mathsf{phno})\}$. Thus, $\mathsf{rescost}(t_7, t_1, 1) = \mathsf{mgcost}(\{eq_1, eq_2\}) = 1$.

5. Repair Algorithms

In this section, provide detailed descriptions of equivalence-class-based constraint repair. We present a general heuristic framework that guarantees termination, and develop two specific heuristic methods GREEDY-REPAIR and GREEDY-REPAIR-FDFIRST. Finally, we discuss optimizations and extensions.

At a high level, our repair algorithm begins by putting each tuple, attribute pair in its own equivalence class. It then greedily merges the equivalence classes of (t, A) pairs until all constraints in $\mathcal{C}$ are satisfied. To illustrate, consider tuples t_1 and t_2 for Bob Jones in Fig. 1. In order to satisfy FD (4), we group tuples t_1 and t_2 on phone number to form the equivalence class

Procedure GEN-REPAIR ($\mathcal{D}, \mathcal{C}$)
Input: Database $\mathcal{D}$, constraint set $\mathcal{C}$.
Output: Database repair $\mathcal{D}'$.
begin
1. $\mathcal{E} := \{\{(t, A)\} : t \in R, A \in \mathsf{attr}(R)\}$;
2. Initialize unResolved sets for FDs and INDs;
3. **while** (unResolved is not empty) {
4. $(t, \mathsf{target}, C) :=$ PICKNEXT ();
5. **if** (C is an FD) **then**
6. FD-RESOLVE-TUP (target, C);
7. **else**
8. IND-RESOLVE-TUP (t, target, C);
9. Process unResolved sets affected by resolution step for C ;
10. }
11. **return** $\mathcal{D}'$; /* Obtained by inserting new tuples into $\mathcal{D}$ and assigning each (t, A) the value for $\mathsf{eq}(t, A)$, i.e., $\mathsf{targ}(\mathsf{eq}(t, A))$ */
end

Figure 4: Generic Equivalence-Class Based Repair Procedure.

$\{(t_1, \mathsf{phno}), (t_2, \mathsf{phno})\}$. Next, to ensure that IND (1) holds, we form two equivalence classes:

- $eq = \{(t_1, \mathsf{phno}), (t_2, \mathsf{phno}), (t_6, \mathsf{phno})\}$. This ensures that tuple t_5 in equip is covered by t_1 and t_2 in cust.
- $eq = \{(t_1, \mathsf{phno}), (t_2, \mathsf{phno}), (t_6, \mathsf{phno}), (t_7, \mathsf{phno}), (t_8, \mathsf{phno})\}$. This ensures that tuples t_7 and t_8 are also covered by t_1 and t_2.

Thus, in the final repaired database, all tuples for Bob Jones: $t_1, t_2, t_6 - t_8$ will have identical phone number values; as a result, these tuples will satisfy constraints FD (4) and IND (1). (Additional equivalence classes involving the other attributes will be needed to satisfy the remaining constraints – we list these in Example 3). We now present our heuristic algorithms in detail.

Tracking Unresolved Tuples. Our overall approach is to resolve (unresolved) tuples one at a time, until no unresolved tuples remain. While not strictly required for correctness, an important efficiency optimization is to keep track of *potentially unresolved* tuples for each dependency in $\mathcal{C}$. To accomplish this, we maintain a data structure $\mathsf{unResolved}(C)$ which maps each constraint $C \in \mathcal{C}$ to a set of tuples. Our repair algorithms ensure that the maintained unResolved sets satisfy the following two invariants: (1) If t is unresolved w.r.t. $I = R_1[X] \subseteq R_2[Y]$, $t \in \mathsf{unResolved}(I)$, and (2) If t is unresolved w.r.t. $F = R[X] \rightarrow R[Y]$, then *some tuple* t' which matches t on attributes in X is guaranteed to be in $\mathsf{unResolved}(F)$; here t' serves as a proxy for t, and when it is resolved, t will also be resolved.

Our maintenance algorithms perform the following actions on the unResolved sets that can be shown to preserve the above-mentioned invariants:

- *Initialization:* For each IND $I = R_1[X] \subseteq R_2[Y]$, $\mathsf{unResolved}(I)$ is initially set to $\{t : t \in R_1\}$. For each FD $F = R[X] \rightarrow R[Y]$, $\mathsf{unResolved}(F)$ is initialized to contain all the tuples in R.
- *After each resolution step:* When a tuple t is resolved w.r.t. a constraint C, the following actions are taken: 1) t is removed from $\mathsf{unResolved}(C)$, 2) a newly inserted tuple into table R is added to $\mathsf{unResolved}(C)$ if C is an FD on table R or an IND of the form $R[_] \subseteq _[_]$, and 3) if resolution causes equivalence class merging, such that $\mathsf{targ}(t, A)$ changes due to the merge, then we add (t, A) to $\mathsf{unResolved}(F)$ for any $F = R[X] \rightarrow R[Y]$ where $t \in R$ and $A \in X$.

Procedure PICKGREEDY ()
Output: The constraint to repair next, and the tuples to resolve for the constraint.
begin
1. $\mathsf{bestCost} := \infty$;
2. **foreach** FD $F = R[X] \rightarrow R[Y], t \in \mathsf{unResolved}(F)$ **do** {
3. $S := \{t' \in R : \mathcal{D}'(t', X) = \mathcal{D}'(t, X)\}$;
4. **if** rescost $(S, F) < \mathsf{bestCost}$ **then**
5. $\mathsf{bestFix} := (t, S, F)$; $\mathsf{bestCost} := \mathsf{rescost}(S, F)$;
6. }
7. /* **if** (FDFirst **and** bestCost $< \infty$) **then return** bestFix; */
8. **for each** IND $I = R_1[X] \subseteq R_2[Y], t \in \mathsf{unResolved}(I)$ **do** {
9. Let t^* be $t' \in R_2$ with minimum $c := \mathsf{rescost}(t, t', I)$;
10. **if** ($c < \mathsf{bestCost}$) **then**
11. $\mathsf{bestFix} := (t, t^*, I)$; $\mathsf{bestCost} := c$;
12. **if** ($\mathsf{inscost}(R_2) < \mathsf{bestCost}$) **then**
13. $\mathsf{bestFix} := (t, \mathbf{new}, I)$; $\mathsf{bestCost} := \mathsf{inscost}(R_2)$;
14. }
15. **return** bestFix;
end

Figure 5: Greedy Selection of the Lowest-Cost Resolution.

It is easy to see that (1-3) above preserve the two invariants on unResolved sets, since this follows directly from the *permanency* and *collision* properties of INDs and FDs, respectively.

5.1 Repair with Equivalence Classes

In Fig. 4, we present GEN-REPAIR, the overall driver for all of our constraint repair procedures. It is abstracted in terms of a function PICKNEXT, which selects the next tuple t to be resolved w.r.t. a constraint C. If C is an FD $R[X] \rightarrow R[Y]$, then PICKNEXT also returns the target set of tuples to resolve—this set essentially consists of tuples in R that agree with t on attributes in X. On the other hand, if C is an IND, then the target returned by PICKNEXT is either another tuple t' or $\mathbf{new}$ to indicate that t should be covered by a newly-created tuple. Note that at line 9 of GEN-REPAIR, unResolved is maintained as described earlier. The proposed repair $\mathcal{D}'$ is produced by inserting new tuples and replacing (t, A) values in $\mathcal{D}$ with $\mathsf{targ}(\mathsf{eq}(t, A))$. The arbitrary selection of what tuple and constraint to address represents the degree of freedom for designing an equivalence-class-based technique, and we present two intuitive greedy approaches in the next subsection.

Correctness. Clearly, the same tuple may enter and leave $\mathsf{unResolved}(F)$ for an FD F many times. Nevertheless, we now argue that PICKNEXT selects a tuple from unResolved to resolve next and returns only a bounded number of $\mathbf{new}$ tuples to fix IND constraints, and that consequently GEN-REPAIR terminates and produces a repair $\mathcal{D}'$ of $\mathcal{D}$.

Theorem 2: *The number of tuple inserts is bounded for* PICKNEXT*, and* GEN-REPAIR *terminates and produces a repair $\mathcal{D}'$ of $\mathcal{D}$ that satisfies the constraints in $\mathcal{C}$.* □

Proof Sketch: The number of inserts by PICKNEXT is bounded because (1) the number of equivalence classes determined by the original database $\mathcal{D}$ is bounded by $|\mathcal{D}| \cdot \alpha$, where α is the maximum number of attributes in any table, (2) for each unique pattern of such equivalence classes at most one new tuple is inserted, (3) for each attribute A in a table *no* new atomic value (resp. new equivalence class) is added, at any time, and (4) an inserted tuple consists of only data from $\mathcal{D}$ and null. From these it follows that the number of equivalent classes containing data from $\mathcal{D}$ and newly inserted tuples is bounded by $O(|\mathcal{D}| \cdot \alpha)$ at any time, and thus the number of newly inserted tuples is bounded.

Termination of GEN-REPAIR follows from the following points. (1) Every iteration removes at least one tuple from unResolved. (2) Tuples are only added to unResolved when tuples are inserted (whose number is bounded) or equivalence classes are merged. (3) The number of merge events is bounded by the number of equivalence classes, which is at most $O(|\mathcal{D}| \cdot \alpha)$. Thus GEN-REPAIR invokes PICKNEXT at most $n \cdot \alpha$ times, where n is the number of tuples in the database (including new tuples). The correctness follows from the fact that unResolved is empty when GEN-REPAIR terminates, and thus, due to the invariants maintained on unResolved, all tuples are resolved w.r.t. constraints in $\mathcal{C}$ at termination. □

5.2 Two Flavors of Greedy Repair

In this subsection, we build our two proposed algorithms for constraint repair in this paper, GREEDY-REPAIR and GREEDY-REPAIR-FDFIRST by making simple changes to PICKNEXT. Our first algorithm, GREEDY-REPAIR, is built from GEN-REPAIR by replacing PICKNEXT with PICKGREEDY shown in Fig. 5. This routine picks and returns an unresolved tuple to repair with the minimum rescost. In the case of an IND constraint, it also returns the lowest cost target of the resolution, which may be a tuple or **new** if an insert in the target relation is the lowest cost step.

To motivate our next algorithm, GREEDY-REPAIR-FDFIRST, we observe that there is a fundamental difference in the manner in which FDs and INDs are repaired in our framework. FD repair, in many respects, is more rigid than IND repair. For an FD $F = R[X] \rightarrow R[Y]$, and a pair of tuples t, t' that violate the FD, repair involves modifying tuple attribute values so that t and t' match on Y. While we have some flexibility in the tuples we choose to modify (t or t') for each non-matching attribute A in Y, the only choice we have is between the values appearing in t and t', which may or may not be similar. In contrast, IND repair is much more flexible. For an IND $I = R_1[X] \subseteq R_2[Y]$, and a tuple $t \in R_1$ that violates the IND, repair can be achieved by considering any tuple t' in R_2, and modifying attribute values so that t and t' match on the corresponding attributes in X and Y. Essentially, any tuple t' in R_2 can be considered as the covering tuple for t in order to repair the IND.

Due to the rigidity of FD repair, we consider the FDFirst variant which gives precedence to fixing FDs. This is accomplished by uncommenting line 7 of PICKGREEDY, so that an unresolved tuple for an FD will be returned if available. If not, the lowest cost tuple for an IND repair is returned.

Example 3: Consider the cust and equip tables depicted in Fig. 1. We trace the sequence of resolution steps performed by the FDFirst variant of our greedy heuristic when it is run on tuples $t_1, t_2, t_6 - t_8$ for Bob Jones. In the following, the "target value" of an attribute A of tuple t refers to $\mathcal{D}'(t, A) = \mathsf{targ}(eq(t, A))$. We only list below the steps that result in new classes due to merges.

1. Resolve tuples t_1, t_2 w.r.t. FD (4), since these two tuples match on name, street, and zip ($\mathsf{rescost}(\{t_1, t_2\}, 4) = 1$). This results in the equivalence class $\{(t_1, \mathsf{phno}), (t_2, \mathsf{phno})\}$ with target value "555-8145". Also, since the phone number in t_2 changes, it is added to $\mathsf{unResolved}(2)$.

2. Resolve tuples t_1, t_2 w.r.t. FD (2) since t_1 and t_2 now match on phno ($\mathsf{rescost}(\{t_1, t_2\}, 1) = 1$). This causes classes for t_1 and t_2 to be merged for every cust attribute, and t_2's target state value to be updated to "ny".

3. Resolve tuples t_1 and t_6 w.r.t. IND (1) ($\mathsf{rescost}(t_1, t_6, 1) = 0$). This yields the class $\{(t_1, \mathsf{phno}), (t_2, \mathsf{phno}), (t_6, \mathsf{phno})\}$ with value "555-8145".

4. Resolve tuples t_2 and t_7 w.r.t. IND (1) ($\mathsf{rescost}(t_2, t_7, 1) = 1$). This causes the phone numbers of tuples t_1, t_2, t_6 and t_7 to be merged into the same equivalence class with value "555-8145". Since the target value of the phone number attribute for t_7 changes to "555-8145", it is added to $\mathsf{unResolved}(6)$.

5. Resolve tuples t_6 and t_7 w.r.t. FD (6) since their target values now match on phno, ($\mathsf{rescost}(\{t_6, t_7\}, 6) = 1$). This results in the equivalence class $\{(t_6, \mathsf{serno}), (t_7, \mathsf{serno})\}$ with value "L55001". Since the target value of the serial number attribute for t_7 changes to "L55001", it is added to $\mathsf{unResolved}(5)$.

6. Resolve tuples t_6 and t_7 w.r.t. FD (5) since they now match on serno ($\mathsf{rescost}(\{t_6, t_7\}, 5) = 2$). This causes classes for t_1 and t_2 to be merged for every equip attribute, and t_7's install date value to be updated to "Jan-03".

7. Resolve tuples t_2 and t_8 w.r.t. IND (1). The phone numbers for tuples t_1, t_2, t_6, t_7, t_8 are merged into the same class with value "555-8145". The value of the phone number attribute for t_8 is changed to "555-8145".

Thus, in the final repair, all of Bob Jones' tuples have identical phone number values of "555-8145". Further, the state in tuple t_2 is modified to "ny", and the serial number and installation date in t_7 are modified to "L55001" and "Jan-03", respectively. Thus, the tuples $t_1, t_2, t_6 - t_8$ satisfy all the FD and IND constraints. Similarly, for Alice Smith's tuples, our heuristic will correct the name and street in tuple t_4, and the phone number in tuple t_9. □

Looking Ahead for FD Costs. In the algorithm GREEDY-REPAIR-FDFIRST, we perform FD repair first to limit the effect of cross-tuple merging on the larger equivalence classes created by IND repair. Another approach to minimizing undue effects from IND repairs is to attempt to avoid bad repairs (that cause many FD violations) by adding some degree of lookahead to the cost model. In order to do this, we modify the rescost of tuples to include an approximation of the cost of resolving tuples added to the unResolved set of Functional Dependencies. Note that this is similar in spirit to the idea of preferring 1-1 entity matching [17].

5.3 Improving Performance

We now analyze the running time of GREEDY-REPAIR and GREEDY-REPAIR-FDFIRST in an attempt to predict their practical behavior, and introduce three important heuristic optimizations.

Running Time. With very complicated overlapping constraints, a finite but exponential number of new tuples could be inserted in the course of a constraint repair. However, our experiments (which will be presented in the next section) demonstrate that when inscost is correctly set, the number of tuples inserted is far fewer than the number of tuples in the original database. Accordingly, we assume below that the number of inserts is linear in the original database size, and for simplicity we use the parameter n as the number of original tuples in the database *plus all the inserts*.

Recall that α is the maximum number of attributes of any table in $\mathcal{D}$, and that $|\mathcal{C}|$ is the number of constraints. Another important parameter is meq, the largest size of any equivalence class encountered during a run. Of course, this is $O(n)$ in the worst case, but is practically much smaller. Given these parameters, the **while** loop of GEN-REPAIR (Figure 4) repeats at most $\alpha \cdot n$ times, since one equivalence class is removed by each pass. In the worst case, in PICKGREEDY, finding the lowest cost target for each tuple in $\mathsf{unResolved}(C)$ for an IND C can take $O(n)$ steps, where each step takes $O(\alpha \cdot \mathsf{meq})$ time to compute the cost of the new equivalence classes. There are potentially $n \cdot |\mathcal{C}|$ unresolved tuples. The cost of resolving the tuple is not trivial since it includes the cost of adding any new tuples to unResolved, but is not excessive since an index can be maintained on the set of attributes involved in the

LHS of each FD. Thus, the time complexity of PICKGREEDY is $O(n^2 \cdot |\mathcal{C}|^2 \cdot \mathsf{meq})$, and the overall running time of GEN-REPAIR, in the worst case, is $O(n^3 \cdot |\mathcal{C}|^2 \cdot \mathsf{meq})$ (α is omitted since it is a constant when the schema is fixed).

This clearly indicates that, while technically tractable, neither GREEDY-REPAIR nor GREEDY-REPAIR-FDFIRST will scale well to large data sets. We now introduce three optimizations which, as will be shown in the next section, make a substantial improvement in scalability without greatly affecting quality.

Redundant Computation. As mentioned above, the most expensive part of $PickGreedy$ is the search for covering tuples for INDs in lines 8-13 of Fig. 5, since the computation of S at line 3 can be assisted by a hash table. We observe that most cost evaluations between one execution of PICKGREEDY and the next are redundant; this optimization seeks to avoid this redundant computation. We now define some notation: Let ts be a global timestamp which is incremented before any equivalence class change. Let $\mathsf{best}(t, I)$ be the $\mathsf{bestFix}$ value computed for a tuple t with respect to IND I in PICKGREEDY. Assume that $\mathsf{lastcompute}(t)$ represents the timestamp at which the last evaluation of PICKGREEDY for this tuple took place. Let $\mathsf{changed}(s)$ be the set of tuples for which the equivalence class of *some attribute* has changed at a timestamp greater than s.

Now consider how to find the new $\mathsf{bestFix}$ for t with respect to constraint I on the next call to PICKGREEDY. When neither t or $\mathsf{best}(t, I)$ has changed since $\mathsf{lastcompute}(t)$, we argue that the new $\mathsf{bestFix}$ for t with respect to i is either $\mathsf{best}(t, I)$, or it inolves a tuple from $\mathsf{changed}(\mathsf{lastcompute}(t))$. This follows directly from the definitions, since $\mathsf{best}(t, I)$ was optimal at time $\mathsf{lastcompute}(t)$ and no other costs have changed. If an equivalence class of t or $\mathsf{best}(t, I)$ has changed, however, then we must scan the entire target table. This optimization does not affect quality, and is always used.

For a given tuple t, the equivalence class of t can only change meq times, potentially reducing the n^3 term to $n^2\mathsf{meq}$. However, extreme cases can be constructed in which for every step, 1) some tuple t' appears as $\mathsf{best}(t, I)$ for *every* unresolved tuple t and IND I, and 2) t' is modified at that step entering $\mathsf{changed}(\mathsf{lastcompute}(t))$. Thus, though the optimization is practically important, the worst-case running time is unchanged.

Nearby Tuples. To further improve running time, when trying to satisfy inclusion dependencies, we limit the number of target tuples considered for each source tuple, based on techniques from duplicate elimination [11, 15, 21]. For each attribute A in relation R appearing on the right-hand side of some IND, we produce a set of indexes of R based on different features abstracted from the attribute [15]. In particular, we keep one list sorted by attribute value; and we create another list by first sorting the characters in the attribute and then using this attribute to sort the list. When looking for target tuples with which to resolve t w.r.t. IND $I = R_1[X] \subseteq R_2[Y]$, we probe values from t's X attributes to access each sorted list for each attribute in Y. We then examine tuples, starting with the best-matching attribute according to the attribute-level similarity metric employed. This produces a candidate set of tuples, which we then order on our tuple-cost metric, and return the first k. We refer to the resulting optimization as NEARBY(k). When combined with the last optimization, we intersect the NEARBY list with the recently changed list. This optimization improves the worst-case running time to $O(n^2 \cdot |\mathcal{C}|^2 \cdot \mathsf{meq} \cdot k \cdot \lg n)$.

Relaxing Greedy. Our third optimization is to relax the PICKGREEDY routine so that it is much more efficient, but does not always choose the lowest cost merge to do next. To this end, we initially create a queue of triples, (t, t^*, I), where $t \in \mathsf{unResolved}(I)$ for some IND I, and t^* is the best fix found by PICKGREEDY (i.e. t^* is either $\mathbf{new}$ or an existing potentially covering tuple). The tuples in the queue are sorted by the cost of resolving t with t^*. Whenever the GEN-REPAIR calls $PickNext$, the lowest-cost unresolved tuple t in the queue is considered, and the following steps are taken: 1) t's best repair and repair cost with respect to I is *recomputed*, using NEARBY if this option is in force. 2) If t's cost is unchanged or reduced, it is chosen without examining other tuples. 3) If t's cost has increased, it is resorted into the queue, and the process continues with the next tuple. This technique is referred to as QUEUE. Since at step 2 there may be some entry later in the queue with a better cost, the lowest cost greedy step is not always found. However, QUEUE does ensure that when resolving a tuple t w.r.t. an IND I, the currently-lowest cost tuple from the target table with which to cover t is used.

Construction of the queue takes at most $O(n \cdot \lg n \cdot |\mathcal{C}|^2)$ time. As with the first optimization, if every tuple's best match is the same tuple, and that tuple changes on each step, this optimization does not improve running time. However, if we limit each step to resorting a constant number of tuples, then in conjunction with NEARBY the worst case running time is in $O(n \cdot |\mathcal{C}|^2 \cdot \mathsf{meq} \cdot \lg^2 n)$. As we will see in the next section, this optimization is also practically very important.

6. Experimental Evaluation

In this section, we present an experimental study of our constraint repair techniques. We investigate the *utility*, *scalability* and *sensitivity to noise* of our low-cost constraint repair heuristics on synthetic TPC-H data and a collection of real datasets.

6.1 Experimental Setting

We perform our experiments with 1) artificial datasets generated by the TPC-H [26] benchmark and 2) real-life datasets containing DVD information. All the experiments are run on similar machines, powered by either 933 MHz or 1 GHz Pentium 3 processors.

TPC Data. For TPC data, we create a clean TPC-H instance $\mathcal{D}^+$ by using the TPC dbgen program with different scaling factors. We refer to this dataset by its size. For example, using a scaling factor of .0002, yielding approximately 2,000 tuples, is referred to as "TPC.2k". We then introduce *noise* to each attribute involved in an IND or appearing on the left-hand side of an FD with probability p_{noise} to produce $\mathcal{D}$. When noise is introduced, with probability p_{cf} (the *confusion metric*) the value of the attribute is replaced with another value found in the same column. Otherwise, the noise introduced is a textual error guaranteed not to cause a collision (insertion of a "!" at a random location in the string). Note that noise may only be applied to those attributes that may cause constraint violations in $\mathcal{D}$. Finally, one of our algorithms is used to repair constraint violations in $\mathcal{D}$, producing a proposed repair $\mathcal{D}'$.

The TPC model includes 10 FDs and 5 INDs generalizing the keys and foreign keys in the dataset. An FD on the supplier table is, e.g., *the supplier key* $\rightarrow$ *name, address, phone number, region.*

DVD Data. For DVD datasets, we scrape information from three e-commerce web sites, AMAZON, DVD EMPIRE and DIGITAL EYES. Each dataset is obtained by searching each site for a single keyword, and filling in a table of information about the DVDs found. We consider 19 arbitrary search words to generate 19 different datasets. Each generated table has the following columns: title, DVD release year, theatrical release year, aspect ratio, length, rating, anamorphic ('y' or 'n'), and a unique ID where possible (AMAZON and DVD EMPIRE). The Title and DVD year columns

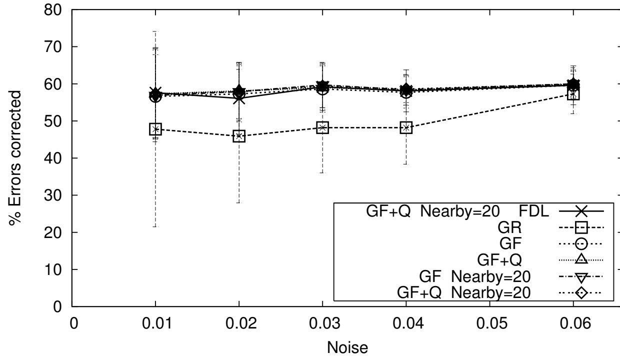

Figure 6: TPC Quality, $p_{cf} = 0.1$.

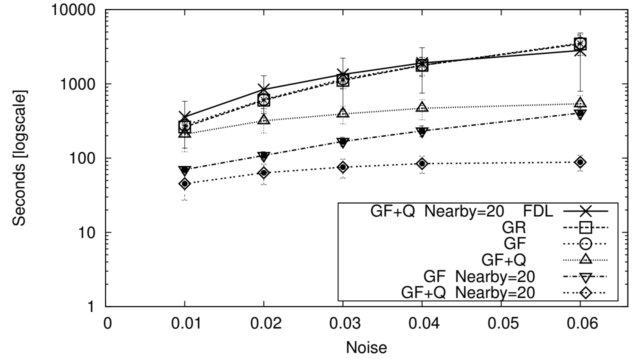

Figure 7: TPC Performance, $p_{cf} = 0.1$.

form the key. Because there were many natural discrepancies between the three sites, no artificial errors were introduced.

For each table we define an FD from its key to all of its other fields. Furthermore, we define an IND from the two key fields in each table to another table, such that the only way to satisfy all of the dependencies is for each table to have an identical set of movies.

Algorithms. We have implemented prototypes in Java of the GREEDY-REPAIR (GR) and GREEDY-REPAIR-FDFIRST (GF) algorithms, with the redundant computation optimizations mentioned in Section 5.3. We also implemented the NEARBY and QUEUE optimizations, which can be used independently or together, as well as LOOKAHEAD FOR FD COSTS (FDL).

Cost Models. We now show the cost function used in these experiments. We start with a string edit distance function SE from Damerau-Levenshtein. For TPC data, we use a modified SE that excludes character replacement operations, allowing only inserts and deletes. The distance between '!3' and '93' therefore increases from 1 to 2. We make this change because the key data in TPC consists of densely populated integers. Without the change, an n-digit string with a '!' noise character inserted (e.g. '!3') would have the same edit distance of 1 to many $(n+1)$-digit numbers (e.g. '13', '23', '33', etc.) as it would to the original string. Even with this change, however, '3' is just as close to '13', '23', etc., as it is to '!3', so errors are still often hard to correct.

We then define the attribute-level distance function dis as

$$\mathsf{dis}(s_1, s_2) = \max\left(1, \frac{SE(s_1, s_2)}{\max(10, \min(|s_1|, |s_2|)} + p\right)$$

where $p = .1$ when $s_1 \neq s_2$, $p = 0$ otherwise, to ensure that different strings can never have a vanishingly small cost. Longer strings with a 1-character difference are closer than shorter strings with a 1-character difference, but a 1-character difference in short strings does not cause a disproportionately high cost. The rescost used in PICKGREEDY (shown in Fig. 5) is then defined to be the sum of dis across all attributes being merged, multiplied by the percentage of attributes being merged that are not exact matches.

Another parameter that must be set is the insert cost inscost. For TPC, no entities are missing, so inscost for all tables is set to infinity. For the movie data, however, missing data is common. We find that a generally good setting for inscost is slightly more than $\mathsf{dis}(s_1, s_2)$, where s_1 and s_2 are short strings that differ by 1 character. We therefore use $\mathsf{inscost} = 0.225$ for the DVD experiments.

Measuring Repair Quality. To measure repair quality, one approach is to consider the cost of the repair found by the algorithm, and compare this cost to some reference, such as the optimal (minimum cost) repair. Unfortunately, we know of no effective algorithm to find the optimal repair for non-trivial data sizes. Furthermore, since our cost metric is new, showing that a low cost solution is found does not prove that a *good* solution is found.

Since the TPC-H data is synthetically generated, we have an original correct instance $\mathcal{D}^+$ available (though it is not seen by the algorithm, of course). Thus we can measure the number of *errors corrected* as the *errors* (the number of attribute-level differences between $\mathcal{D}^+$ and $\mathcal{D}$) minus the *errors remaining* (the difference between $\mathcal{D}'$ and $\mathcal{D}^+$). The *quality* of a TPC solution refers to the ratio of errors corrected to errors.

For movie data, measuring quality is more problematic. One approach would be to hand-merge each data set to determine an ideal merge. However, this is impractical for the larger data sets. We chose to approximate the hand-merge by building a Perl script to evaluate the merges done to reach $\mathcal{D}'$. This script has a set of "expert" rules (created by studying the evaluated movie data) which determines if two movies are the same. Given a repair, this script counts the number of bad merges between movies that are not the same. It then calculates an "adjusted size" (ADJSZ), which is $mov + penalty * bad$, where mov is the number of movies, and bad is the number of bad merges. Lower values of ADJSZ mean that the repair created a more compact, accurate database. So, for example, if a database should have 2 tuples, the correct result would have an ADJSZ of 2, with 0 bad merges. But if a repair merged those two tuples into one incorrectly, the ADJSZ would be $1 + penalty * 1$. In order for the bad repair to have a worse ADJSZ than the correct repair, we have set $penalty$ to be 2 in our experiments. We compare our algorithms with a $naive$ repair, which simply merges any two DVDs when the title and year are exact matches, and report the quality as the improvement in ADJSZ of the given algorithm over the naive repair.

6.2 Experimental Results

TPC Result Quality. These experiments quantify how much our algorithms improve the quality of the data. For the TPC data, each experiment was repeated 30 times with different random seeds for error introduction (except for the scaling experiments, which were done 3 times each). We test p_{cf} with values ranging between 0% and 40%. The results with $p_{cf} = 10\%$ on TPC.2k are shown in Fig. 6. We see that GREEDY-REPAIR performs worse than the GREEDY-REPAIR-FDFIRST-based algorithms. We also find that larger p_{cf} increases the difference between GREEDY-REPAIR and GREEDY-REPAIR-FDFIRST (not shown).

The large error bars show that GREEDY-REPAIR occasionally makes very bad repairs. This happens when an expensive FD is deferred while many mainly similar equivalence classes are merged due to INDs, only for them all to ultimately be merged together. (GREEDY-REPAIR-FDFIRST avoids this problem by never deferring FDs.) Interestingly, as noise increases, this problem happens *less* frequently, as large merges of mainly similar equivalence classes become less common.

Figure 6 also shows that the optimization techniques do not de-

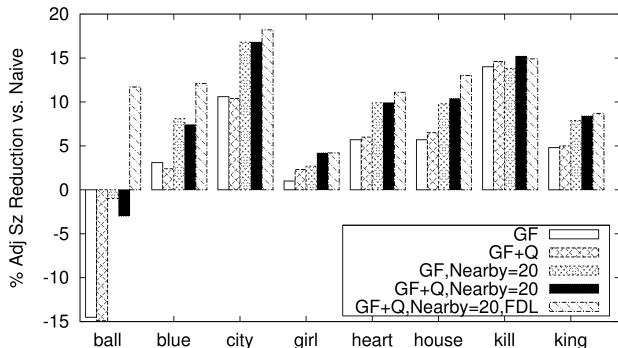

Figure 8: Repair Quality on Movie Dataset

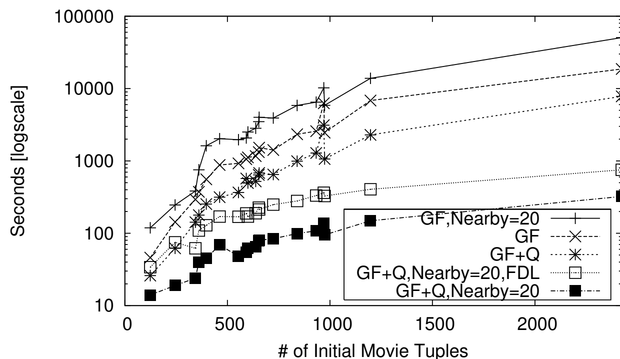

Figure 9: Performance on DVD data

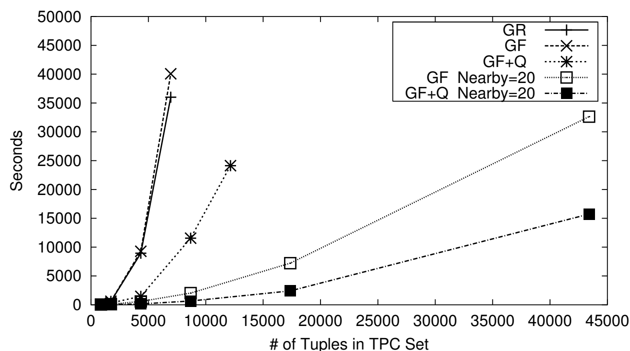

Figure 10: Scalability of GREEDY-REPAIR-FDFIRST

grade the quality of the results for this dataset and dependency set. Meanwhile, Figure 7 shows that the optimization techniques do have a significant effect on runtime (user-time).

DVD Result Quality. Figure 8 shows the percentage decrease in ADJSZ over naive for each algorithm for a subset of the databases we test. Whereas GREEDY-REPAIR was inferior to GREEDY-REPAIR-FDFIRST on TPC data with non-zero p_{cf}, it is the same on the DVD dataset, and is not shown.

In these results, once again the QUEUE and NEARBY optimizations do not degrade the results, compared to GREEDY-REPAIR-FDFIRST (hollow bar). In fact, the NEARBY optimization *improves* the results, because for matches at equal edit distances, it prefers the match which is closer alphabetically. (This is consistent with research [28] that it is beneficial to include the length of the common prefix of two strings in a string matching score.)

For some movie datasets, our string matching function does poorly matching titles in the generated database. For example, the query “ball” generates many long similar titles that however refer to different movies, such as “Dragon Ball Z: Garlic Jr. - Vanquished” and “Dragon Ball Z: Garlic Jr. Saga”. On such a data set, the results are in general poor, as is shown in the figure. Adding the FD lookahead heuristic, however, enables the algorithm to avoid many of the bad merges, greatly improving the quality of the result.

Figure 9 shows the performance of the different algorithms. NEARBY and QUEUE are needed to scale, and adding FD Lookahead degrades performance, but not dramatically.

Furthermore, *GF* + NEARBY*(*20*)* is actually slower than *GF* by itself. The reason for this is that the Redundant Computation optimization is very effective when the set of changed tuples that needs reevaluation is small. When only one or two tuples have changed, the time needed to use NEARBY to determine if those tuples should be considered is greater than the time needed to simply recompute for those tuples. So, for small change sets, NEARBY actually degrades performance. For GREEDY-REPAIR-FDFIRST, when all tuples are considered after each change, only one new equivalence class can be created in a column between computations, and thus the change set is limited by the size of an equivalence class. (With the QUEUE optimization, multiple equivalence classes are created between computations on a given tuple, and thus change sets are bigger.) For DVD data, the equivalence classes are always small and so *GF*+NEARBY*(*20*)* performs poorly.

Scalability. In Fig. 10 we vary the size of the TPC instance from 1000 to 44,000 tuples, and compare the running time of GREEDY-REPAIR-FDFIRST alone with NEARBY, QUEUE, and with both NEARBY and QUEUE. Because the equivalence classes are larger than those in the DVD data, the NEARBY optimization is very effective, and is critical for the algorithm to scale with this dataset size.

Impact of Constraint Sets. An advantage of the algorithms discussed in this paper is that they give the user flexibility in specifying the integrity constraints that need to be met. There are often many constraint sets with equivalent semantics. In Figure 11, using GREEDY-REPAIR-FDFIRST with QUEUE and NEARBY(20), we compare two runs on the DVD data. In one run, we use a dependency set that consists of INDs across all of the attributes of all of the tables (instead of just the key fields). In the other run, for each table we add FDs from the key fields to all the other fields. Either dependency set could be reasonably used for matching up DVDs between the different databases. As the figure shows, much more consistent and better results are obtained with the FD presence.

Varying NEARBY**.** We also experiment with GREEDY-REPAIR-FDFIRST and QUEUE with 2% noise, while varying NEARBY. The results are shown in Fig. 12. (The right-hand Y-axis, used for the two movie datasets, is inverted so that better, lower values of ADJSZ are higher.) It shows that for the DVD data, with real strings, even a nearby of 4 is sufficient for good results. For the TPC data, where the text data is actually integers, it shows that higher values of NEARBY are needed. In fact, as is shown for $Nearby < 8$ in TPC.4k, values of NEARBY that are too low may cause the algorithm to fail dramatically, finding a repair with a very high cost. If the nearby string set frequently fails to include the correct choice, a good repair cannot be found. Moreover, because our NEARBY sort orders do not perform well on the integer keys seen in TPC, larger values of NEARBY are needed. For TPC.2k, NEARBY of approximately 8 is sufficient, whereas for TPC.4k, NEARBY of 20 is required to get optimal results.

This graph also shows that for the same amount of noise in TPC data, a higher percentage of errors can be corrected in a larger database. This is because a larger TPC database has more redundant values, and hence more errors are correctable.

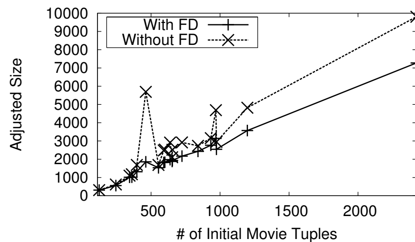

Figure 11: Constraint choices

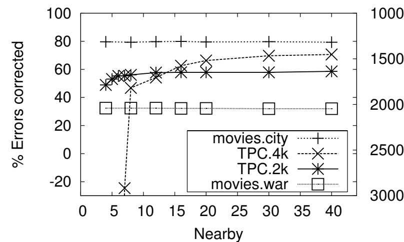

Figure 12: Vary Nearby (2% noise)

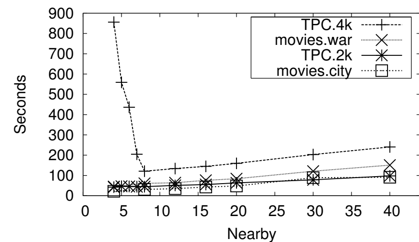

Figure 13: Nearby Runtime

Figure 13 shows that excluding the failed repair, performance scales linearly with NEARBY, as expected.

Summary. We have presented several results from our experimental study of cost-based constraint repair. First, we find that our equivalence-class-based constraint repair heuristics, in almost all cases, substantially improve the quality of an inconsistent TPC or DVD data set. Second, we compare several heuristics and find that GREEDY-REPAIR-FDFIRST with NEARBY and QUEUE scales much better than other combinations yet produces repairs of similar quality. Finally, we note that the inclusion of FDs and FD Lookahead can significantly improve repair quality. While our experiments on the DVD dataset perform essentially a data-linkage task, we do not compare ourselves with other data linkage systems. These systems have rule-bases for matching and carefully tuned comparison metrics, and will presumably achieve better results if applied to matching movie titles. Instead, our goal is to show that a general constraint-repair facility can reasonably perform this well-known task, and that our cost-based model affords the opportunity to integrate techniques from record linkage, like NEARBY, into a practical constraint repair system. Finally, it is easy to see that constraint repair *generalizes* data linkage, since constraint repair 1) has the added burden of suggesting a repair once linkage is accomplished, and 2) handles complex data models and can generalize to additional kinds of constraints.

7. Related Work

Data cleaning systems described in the research literature include the AJAX system [12] which provides users with a declarative language for specifying data cleaning programs, and the Potter's Wheel system [25] that extracts structure for attribute values and uses these to flag discrepancies in the data. Most commercial ETL tools for data warehouses have little built-in data cleaning capabilities covering mainly data transformation needs such as data type conversions, string functions, etc. [24] presents a comprehensive survey of commercial data cleaning tools, as well as a taxonomy of current approaches to data cleaning. While a constraint repair facility will logically become part of the cleaning process supported by these systems, we are not aware of analogous functionality currently in any of the systems mentioned.

Most closely related to our work is the line of research on inconsistent databases (e.g., [1, 3, 4, 5, 7, 10, 13, 14, 27]), i.e., databases that violate given integrity constraints. A semantic notion of minimal repair was first introduced in [1] in terms of symmetric difference of the original database and its repair under set containment. Consistent information is obtained from an inconsistent database following two approaches: *repair* is to find another database that is consistent and minimally differs from the original database [7, 10, 13, 14]; and *consistent query answer*, a notion also introduced by [1], is to find an answer to a given query in every repair of the original database [1, 3, 4, 5, 14, 27]. Most earlier work (except [10, 27]) either adopts tuple deletions as repair primitive and requires that a repair is a subset of the original database [7], assuming that the database is complete yet not necessarily correct, or allows both tuple insertions and deletions [1, 3, 4, 5, 13, 14], assuming that the database is neither sound nor complete. Recently tuple modifications were studied as repair actions for consistent conjunctive query answer [27] and census data repair [10]. In these settings, complexity results [1, 4, 7, 14, 27], algorithms [1, 4, 5, 7, 27], constraint rewriting techniques [14], representations of all repairs with logic programming [3, 10] or tableau [27], and constraint repair based on techniques from model-based diagnosis [13] were developed, for single database [1, 4, 7, 10, 13, 14, 27] and integration systems [3, 5, 14] (see recent surveys on consistent query answer [2] and on constraint repair [6]).

While our work was inspired by prior work on constraint repair, it differs from earlier approaches in the following. First, our repair model allows attribute values to be modified for restoring constraints and introduces a cost-based notion of minimal repairs. While [10, 27] also allow value modifications, the applicability of their techniques is restricted to specific databases or certain constraints. Indeed, in contrast to the generic setting of data cleaning in the presence of both FDs and INDs studied in this paper, [10] considers detecting and solving conflicts for specific census databases of a fixed schema, and [27] studies consistent answer of (conjunctive) queries in the presence of universal (full) constraints, which cannot express INDs. Furthermore, our model introduces the novel and practical notion of minimality for repairs based on costs measured by tuple weights and value similarity for each modification, which was not considered by any previous work. While most of prior models focus on tuple deletions/insertions [1, 3, 4, 5, 7, 13] as repair actions, we would like to emphasize that an attribute-modification cost model seems preferable as discussed in Section 3.1. However, computing repair cost in terms of value modification rather than tuple insertion and deletion significantly complicates the repair problemin the presence of FDs and INDs. For instance, with modifications, an uncovered tuple that violates an IND can be covered by modifying any of the tuples in the covering table (In Example 1, tuple t_9 that violates IND (1) can be covered by updating the phone number field in any of the tuples in the customer table to "949-2212".) Thus, with modifications, the search space for repairing constraint violations explodes, making the search techniques of the earlier work impractical. Second, our NP-completeness results (Section 3) extend the complexity results developed for constraint repair (and consistent query answer, due to the connection between the complexity of these two issues established in [7]: in the presence of foreign keys, the problem of constraint repair is logspace-reducible to the complement of the problem for consistent query answer). Indeed, our results show that in the presence of attribute value modifications as legitimate repair actions, the repair problem is intractable even when either INDs alone or only FDs are allowed, while in contrast, [7] shows that in a delete-only repair model, the corresponding problem is tractable, and it becomes Co-NP hard if arbitrary FDs and INDs are put together. Finally, to the best of our knowledge, our equivalence-class-based approach yields the first effective heuristic algorithm

for restoring the database to a consistent state, and leads to a practical tool for data reconciliation and data cleaning. In contrast, earlier approaches in the presence of value modifications require exponential time (combined complexity) for corrections [10], or expensive tableau construction [27] (the termination problem for its chase-based process is undecidable when it is generalized to deal with both INDs and FDs).

A related topic of interest in data cleaning is the elimination of *approximate duplicates*, also referred to as the object identity or merge/purge problem; see, for example, [8, 11, 15, 20, 21, 28]. This problem frequently occurs during the integration of disparate data sources, such as medical records, address lists or census records. Previous work has focused on the statistical foundation of feature matching [28], issues associated with string matching [8, 25] and the performance issues associated with avoiding pair-wise comparisons of every tuple in a large table. In [11], the authors present the AJAX system for cleaning which includes approximate matching with an SQL-like syntax. For performance, [15] proposes a method which considers multiple sorts of the database using different combinations of attributes, which inspires our NEARBY optimization. It should be mentioned that the algorithms developed for detecting approximate duplicates [15, 20, 21] compute clusters (transitive closures) of records, which can also be understood as computing "equivalence classes" of "tuples". Our algorithms are more involved than theirs as we need to compute equivalence classes of *attribute* and *tuple pairs* in order to repair FDs and INDs across multiple tables of related entities, rather than to find approximately duplicate records in one or two tables.

Finally, there has been a considerable amount of work in recent years on schema matching (see, e.g., [23] for a survey) in the context of schema integration. This, along with value reformatting, is a necessary pre-step to database reconciliation by constraint repair.

8. Concluding Remarks

In this paper, we observe that an important use of data integration is *database reconciliation*, that is, to correct errors introduced by source databases. We model this problem as one of finding low-cost repairs of constraint violations in an integrated database. Having shown the intractability of the problem, we introduce a heuristic approach based on equivalence classes of (tuple, attribute) pairs. In this context we consider specific algorithms and a number of performance optimizations. We demonstrate the utility of the approach and the scalability of our algorithms with experimental evaluation on synthetic and real data. Not only does this approach help to suggest reasonable repairs, but it allows the reuse of match metrics developed for record linkage in the context of constraint repair.

For future work, we intend to investigate extensions within the relational model, since SQL allows more general "NOT EXISTS" style constraints. Second, we are looking to extend our results to cover semi-structured data as well; specifically, we intend to develop schemes for repairing constraint violations in the context of XML data integration. Finally, we intend to more explicitly address the interactive formulation of the problem since we believe user involvement will be important in constraint repair, as in any other area of data cleaning. This will involve giving the user appropriate control over the repair process as well as the development of appropriate visualization tools for the proposed repairs.

Acknowledgements. The authors wish to thank Jan Chomiki for helpful discussions and comments on a previous draft of this paper.

9. References

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[2] L. Bertossi and J. Chomicki. Query Answering in Inconsistent Databases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*. Springer, 2003.

[3] L. Bravo and L. Bertossi. Logic programs for consistently querying data integration systems. In *IJCAI*, 2003.

[4] A. Cali, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, 2003.

[5] A. Cali, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *IJCAI*, 2003.

[6] J. Chomicki and J. Marcinkowski. On the Computational Complexity of Minimal-Change Integrity Maintenance in Relational Databases. In L. Bertossi, A. Hunter, and T. Schaub, editors, *Integrity Tolerance*. Springer, 2004.

[7] J. Chomicki and J. Marcinkowski. "Minimal-Change Integrity Maintenance Using Tuple Deletions". *Information and Computation*, in press.

[8] W. Cohen, P. Ravikumar, and S. Feinberg. A comparison of string-distance metrics for name-matching tasks. In *IIWeb*, 2003.

[9] M. G. Elfeky, V. S. Verykios, and A. K. Elmagarmid. TAILOR: A record linkage toolbox. In *ICDE*, 2002.

[10] E. Franconi, A. L. Palma, N. Leone, S. Perri, and F. Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *LPAR*, 2001.

[11] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. "AJAX: An Extensible Data Cleaning Tool". In *SIGMOD*, 2001.

[12] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. "Declarative Data Cleaning: Language, Model and Algorithms". In *VLDB*, 2001.

[13] M. Gertz and U. Lipeck. A diagnostic approach to repairing constraint violations in databases. In *DX*, 1995.

[14] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Eng*, 15(6):1389–1408, 2003.

[15] M. A. Hernandez and S. Stolfo. "Real-World Data is Dirty: Data Cleansing and the Merge/Purge Problem". *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[16] International Standard ISO/IEC 9075-2:2003(E). Information technology: Database languages, SQL Part 2 (Foundation, 2nd edition), 2003.

[17] M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa florida. *Journal of the American Statistical Association*, 89:414–420, 1989.

[18] Lavastorm. Making the Case for Automated Revenue Assurance Solutions. http://www.lavastormtech.com.

[19] Lucent Technologies. Revenue Assurance Products. http://www.lucent.com/solutions/revenue.html.

[20] A. Monge. Matching algorithm within a duplicate detection system. *IEEE Data Engineering Bulletin*, 23(4), 2000.

[21] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *DKMD*, 1997.

[22] Network Resource Management: Inventory Takes Stage. Rhk research, July 2002. RHK Market Report.

[23] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[24] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4), 2000.

[25] V. Raman and J. M. Hellerstein. "Potter's Wheel: An Interactive Data Cleaning System". In *VLDB*, 2001.

[26] Transaction Processing Performance Council. TPC-H Benchmark. http://www.tpc.org.

[27] J. Wijsen. Condensed representation of database repairs for consistent query answering. In *ICDT*, 2003.

[28] W. Winkler. Advanced methods for record linkage. Technical report, Statistical Research Division, U.S. Bureau of the Census., 1994.