Messing Up with BART: Error Generation for Evaluating Data-Cleaning Algorithms

Patricia C. Arocena[*]
University of Toronto, Canada

Boris Glavic
Illinois Inst. of Technology, US

Giansalvatore Mecca
University of Basilicata, Italy

Renée J. Miller [*]
University of Toronto, Canada

Paolo Papotti
QCRI Doha, Qatar

Donatello Santoro
University of Basilicata, Italy

ABSTRACT

We study the problem of introducing errors into clean databases for the purpose of benchmarking data-cleaning algorithms. Our goal is to provide users with the highest possible level of control over the error-generation process, and at the same time develop solutions that scale to large databases. We show in the paper that the error-generation problem is surprisingly challenging, and in fact, NP-complete. To provide a scalable solution, we develop a correct and efficient greedy algorithm that sacrifices completeness, but succeeds under very reasonable assumptions. To scale to millions of tuples, the algorithm relies on several non-trivial optimizations, including a new symmetry property of data quality constraints. The trade-off between control and scalability is the main technical contribution of the paper.

1. INTRODUCTION

We consider the problem of empirically evaluating data-cleaning algorithms. Currently, there are no openly-available tools for systematically generating data exhibiting different types and degrees of quality errors. This is in contrast to related fields, for example, *entity resolution*, where well-known datasets and generators of duplicated data exist and can be used to assess the performance of algorithms.

We allow the user to control of the data distribution by providing a clean database (DB) into which our error generator, BART[1], introduces errors. BART supports different kinds of random errors (including typos, duplicated values, outliers and missing/bogus values). Our main technical innovation, however, is related to the problem of generating errors in the presence of data quality rules. BART permits a user to specify a set of data quality rules in the powerful language of *denial constraints* [21]. Denial constraints (DC) are a rule-language that can express many data quality rules including functional dependencies (FD), conditional functional dependencies (CFD) [7], cleaning equality-generating dependencies [16], and fixing rules [24]. In addition, BART permits a user to declare parts of a database as *immutable*, and hence we can express editing rules [14] that use master data to determine a repair.

BART provides the highest possible level of control over the error-generation process, allowing users to choose, for example, the percentage of errors, whether they want a guarantee that errors are detectable using the given constraints, and even provides an estimate of how hard it will be to restore the database to its original clean state (a property we call *repairability*). This control is the main innovation of the generator and distinguishes it from previous error generators that control mainly for data size and amount of error. BART permits innovative evaluations of cleaning algorithms that reveal new insights on their (relative) performance when executed over errors with differing degrees of repairability.

Solving this problem proves surprisingly challenging, for two main reasons. First, we introduce two different variants of the error-generation problem, and show that they are both NP-complete. To provide a scalable solution, we concentrate on a polynomial-time algorithm that is correct, but not complete. Even so, achieving the desired level of scalability remains challenging. In fact, we show that there is a duality between the problem of injecting detectable errors in clean data, and the problem of detecting violations to database constraints in dirty data, a task that is notoriously expensive from the computational viewpoint. Finding the right trade-off between control over errors and scalability is the main technical problem tackled in this paper.

Example 1: [Motivating Example] Consider a database schema composed of two relations Emp and MD and the data shown in Figure 1. Suppose we are given a single data quality rule (a functional dependency) d_1 : Emp : Name $\rightarrow$ Dept.

We can introduce errors into this database in many ways. An example error is to change the Salary value of tuple t_4 to *"3000"*. This change (ch_0) creates an error that is not detectable using the given data quality rule d_1. On the contrary, a second change (ch_1) that changes the Name value of tuple t_2 to *"John"* introduces a *detectable error* since this change causes tuples t_1 and t_2 to agree on employee names, but not on department values. Of course, if other errors are introduced into the DB (for example, t_1.Dept is changed

[*]Supported in part by NSERC BIN.

[1]BART: Benchmarking Algorithms for data Repairing and Translation

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.

Proceedings of the VLDB Endowment, Vol. 9, No. 2
Copyright 2015 VLDB Endowment 2150-8097/15/10.

Emp

	Name	Dept	Salary	Mng
t_1 :	John	Staff	1000	Mark
t_2 :	Paul	Sales	1300	Frank
t_3 :	Jim	Staff	1000	Carl
t_4 :	Frank	Mktg	1500	Jack

MD

	Name	Dept	Mng
t_{m1} :	John	Staff	Mark
t_{m2} :	Frank	Mktg	Jack

Figure 1: Example Clean Database

to *"Sales"*), then ch_1 may no longer be detectable. BART permits the user to control not only the number of errors, but also how many errors are detectable and whether they are detectable by a single rule or many rules.

We not only strive to introduce detectable errors into I, we also want to estimate how hard it would be to restore I to its original, clean state. Consider the detectable error introduced by ch_1. This change removed the value Paul from the DB. Most repair algorithms for categorical values will use evidence in the DB to suggest repairs. If Paul is not in the active domain, then changing $t_2.Name$ to Paul may not be considered as a repair. BART lets the user control the *repairability* of errors by estimating how hard it would be to repair an error to its original value. □

Error-Generation Tasks. We formalize the problem of error generation using the notion of an *error-generation task*, **E**, that is composed of four key elements: (*i*) a database schema S, (*ii*) a set Σ of denial constraints (DCs) encoding data quality rules over S, (*iii*) a DB I of S that is clean with respect to Σ, and (*iv*) a set of configuration parameters Conf to control the error-generation process. These parameters specify, among other things, which relations are immutable, how many errors should be introduced, and how many of these errors should be detectable. They also let the user control the degree of repairability of the errors.

We concentrate on a specific update model, one in which the database is only changed by updating attribute values, rather than through insertions or deletions. This update model covers the vast majority of algorithms that have been proposed in the recent literature [3, 5, 6, 8, 14, 16, 20, 24, 25]. And importantly, it suggests a simple, flexible and scalable measure to assess the quality of repairs, a fundamental goal of our work, as we shall discuss in Section 7.

Contributions. Our main contributions are the following.
(*i*) We present the first framework for generating random and data-cleaning errors with fine-grained control over error characteristics. We allow users to inject a fixed number of detectable errors into a clean DB and to control repairability.
(*ii*) We introduce a new computational framework based on *violation-generation queries* for finding candidate cells (tuple, attribute pairs) into which detectable errors can be introduced. We study when these queries can be answered efficiently, and show that determining if detectable errors can be introduced is computationally hard.
(*iii*) We introduce several novel optimizations for violation-generation queries. We show that extracting tuple samples, along with computing cross-products and joins in main memory, brings considerable benefits in terms of scalability. We also identify a fragment of DCs called *symmetric constraints* that considerably extend previous fragments for which scalable detection techniques have been studied. We develop new algorithms for detecting and generating errors with symmetric constraints, and show that these algorithms have significantly better performance than the ones based on joins. Finally, we discuss the benefits of these optimizations when reasoning about error detectability and repairability.
(*iv*) We present a comprehensive empirical evaluation of our error generator to test its scalability. Our experiments show that the error-generation engine takes less than a few minutes to complete tasks that require the execution of dozens of queries, even on DBs of millions of tuples. In addition, we discuss the relative influence of the various activities related to error generation, namely identifying changes, applying changes to the DB, checking detectability and repairability.
(*v*) The generator provides an important service for data-cleaning researchers and practitioners, enabling them to more easily do robust evaluations of algorithms, and compare solutions on a level playing field. To demonstrate this, we present an empirical comparison of several data-repairing algorithms over BART data. We show novel insights into these algorithms, their features and relative performance that could not have been shown with existing generators.

Our error generator is open source and publicly available (`db.unibas.it/projects/bart`). We believe the system will raise the bar for evaluation standards in data cleaning.

The problem of error generation has points of contact with other topics in database research, such as the view-update problem [2], the problems of missing answers [19] and why-not provenance [18], and database testing [4]. However, our new algorithms and optimizations - specifically designed for the error-generation problem - are what allows us to scale to large error-generation tasks. These techniques have not previously been identified and will certainly have applicability to any of the related areas.

Organization of the Paper. We introduce notation and definitions in Sections 2 and 3. Section 4 formalizes the error-generation problem. Section 5 presents the violation-generation algorithm, followed by a number of important optimizations in Section 6. Use cases of the system are presented in Section 7. We empirically evaluate our techniques in Sections 8 and 9, and discuss related work in Section 10. We conclude in Section 11.

2. CONSTRAINTS AND VIOLATIONS

We assume the standard definition of a relational database schema S, instance I, and tuple t. In addition, we assume the presence of *unique tuple identifiers* in an instance. That is, t_{id} denotes the tuple with identifier (id) "*id*" in I. A *cell* in I is a tuple element specified by a tuple id and attribute pair $\langle t_{id}, A_i\rangle$. The *value* of a cell $\langle t_{id}, A_i\rangle$ in I is the value of attribute A_i in tuple t_{id}. As an alternative notation we use $t_{id}.A_i$ for a cell $\langle t_{id}, A_i\rangle$. A *relational atom* over a schema S is a formula of the form $R(\bar{x})$ where $R \in \mathsf{S}$ and $\bar{x}$ is a tuple of (not necessarily distinct) variables. A *comparison atom* is a formula of the form $v_1 \,\mathsf{op}\, v_2$, where v_1 is a variable and v_2 is either a variable or a constant, and op is one of the following comparison operators: $=, \neq, >, <, \leq, \geq$.

We use the language of denial constraints (DCs) [21] to specify cleaning rules. We introduce a *normal form* for DCs:

Definition 1: Denial Constraint – A *denial constraint* (DC) in normal form over schema S is a formula of the form

$$\forall \bar{x_1}, \ldots, \bar{x_n} : \neg(R_1(\bar{x_1}), \ldots, R_n(\bar{x}_n), \bigwedge_{i=1}^{m}(v_1^i \,\mathsf{op}\, v_2^i))$$

such that (*i*) $R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$ are relational atoms, where variables are not reused within or between atoms, and (*ii*)

$\bigwedge_{i=1}^{m}(v_1^i \,\mathsf{op}\, v_2^i)$ is a conjunction of comparison atoms, each of the form $x_k \,\mathsf{op}\, x_l$ or $x_k \,\mathsf{op}\, c$, where $c \in \mathsf{Consts}$. ☐

Example 2: Continuing with Example 1, we introduce the following sample declarative constraints.
(*i*) We already discussed FD d_1 : Emp : Name → Dept.
(*ii*) The FD Name → Mng holds over table Emp, but only for those employees in *"Sales"*. We specify this as a conditional FD (CFD) d_2 : Emp : Dept[*"Sales"*], Name → Mng.
(*iii*) A *standardization rule* over table Emp states that all employees working in department *"Staff"* must have a salary of one-thousand dollars. We model this using a (single-tuple) CFD d_3 : Emp : Dept[*"Staff"*] → Sal[*"1000"*].
(*iv*) An *editing rule* [14] d_4 prescribes how to repair inconsistencies over Emp based on master-data from MD: *"when Emp and MD agree on Name and Dept, change Emp.Mng to the value taken from MD.Mng"*. This rule requires that MD can not be changed (the MD table is *immutable*).
(*v*) Finally, we consider an *ordering constraint* d_5 stating that any manager should have a salary that is not lower than the salaries of his or her employees.

Next we encode our sample constraints d_1–d_5 as a set of DCs (universal quantifiers are omitted).

$\mathsf{dc}_1 : \neg(\mathsf{Emp}(n,d,s,m), \mathsf{Emp}(n',d',s',m'), n = n', d \neq d')$
$\mathsf{dc}_2 : \neg(\mathsf{Emp}(n,d,s,m), \mathsf{Emp}(n',d',s',m'), n = n', d = d',$
$\qquad d = \textit{"Sales"}, m \neq m')$
$\mathsf{dc}_3 : \neg(\mathsf{Emp}(n,d,s,m), d = \textit{"Staff"}, s \neq \textit{"1000"})$
$\mathsf{dc}_4 : \neg(\mathsf{Emp}(n,d,s,m), \mathsf{MD}(n',d',m'), n = n', d = d', m \neq m')$
$\mathsf{dc}_5 : \neg(\mathsf{Emp}(n,d,s,m), \mathsf{Emp}(n',d',s',m'), m = n', s' < s)$ ☐

To formalize the semantics of a DC, we introduce the notion of an *assignment* to the variables of a logical formula $\varphi(\bar{x})$. DCs in normal form have the nice property that each variable has a unique occurrence within a relational atom. We can therefore define an *assignment* as a mapping m of the variables $\bar{x}$ to the cells of I. We formalize the notion of a *violation* for a constraint using assignments.

Definition 2: Violation – Given a constraint $\mathsf{dc} : \forall\, \bar{x} : \neg(\phi(\bar{x}))$ over schema S, and an instance I of S. We say that dc is *violated* in I if there exists an assignment m such that $I \models \phi(m(\bar{x}))$. For a set Σ of DCs, $I \models \Sigma$ if none of the constraints in Σ is violated in I. ☐

Consider constraint dc_1 in Example 2. Assume two tuples are present in the instance of Emp: t_1 = Emp(John, Staff, 1000, Mark), t_6 = Emp(John, Sales, 1000, Mark). These tuples are a violation of dc_1 according to assignment m that maps variables to cells and atoms to tuples as follows (we omit s, s', m, m'):

$m(n) = t_1.\mathsf{Name}\quad m(d) = t_1.\mathsf{Dept}\quad m(\mathsf{Emp}(n,d,s,m)) = t_1$
$m(n') = t_6.\mathsf{Name}\quad m(d') = t_6.\mathsf{Dept}\quad m(\mathsf{Emp}(n',d',s',m')) = t_6$

To determine if a constraint is violated, we must examine values assigned to variables used in comparison atoms. To emphasize this, we call the *context variables* of a constraint dc in normal form those variables that appear in comparison atoms (in our example, $\{n, n', d, d'\}$). A *context* for a violation is the set of cells associated with context variables (in our example, $\{t_1.\mathsf{Name}, t_1.\mathsf{Dept}, t_6.\mathsf{Name}, t_6.\mathsf{Dept}\}$).

Definition 3: Violation Context – Given a constraint $\mathsf{dc} : \forall\, \bar{x} : \neg(\phi(\bar{x}))$ over schema S in normal form, and an instance I of S, assume I contains a violation of dc according to assignment m. The *violation context* for dc and m, denoted by $\textit{vio-context}_{\mathsf{dc}}(m)$, is the set of cells that are the image according to m of the context variables of dc. ☐

3. PROPERTIES OF ERRORS

Our primary concern is to provide users with fine-grained control over the error-generation process. To do so, we concentrate on two important properties of the changes we make to the clean instance, namely *detectability* and *repairability*.

Detectability. We define precisely when a cell change ch introduces an error that is detectable using Σ. We say that cell $t_i.A$ is *involved in a violation* with dc in instance I if there exists an assignment m such that I violates dc according to m, and $t_i.A \in \textit{vio-context}_{\mathsf{dc}}(m)$.

Definition 4: Detectable Errors – Consider a set of cell changes $\mathcal{C}h$ to instance I that yield a new instance $I_d = \mathcal{C}h(I)$. We say that a cell change $\mathsf{ch} = \langle t_i.A := v\rangle$ in $\mathcal{C}h$ introduces a *detectable error* for constraint dc if: (*i*) cell $t_i.A$ is *involved in a violation* with dc in instance I_d and (*ii*) cell $t_i.A$ was not involved in a violation with dc in instance $I' = \mathcal{C}h'(I)$, where $\mathcal{C}h' = \mathcal{C}h - \{\mathsf{ch}\}$. ☐

The cell-change $ch_1 = \langle t_2.\mathsf{Name} := \textit{"John"}\rangle$ from Example 1 introduces a detectable error for dc_1. Notice that it does not introduce any violation to dc_2-dc_5. Hence, we say that ch_1 is detectable by *exactly-one* constraint in $\Sigma = \{dc_1, ...dc_5\}$. This is not always the case. For example, the change $ch_2 = \langle t_4.\mathsf{Mgr} = \textit{"John"}\rangle$ is detectable using both dc_5 (managers must have a greater salaries than their employees), and dc_4 (employees present in the master-data table, must have the manager indicated by the master data). We say that ch_2 is *at-least-one* detectable, or simply detectable.

Reasoning about detectability is important. There are repair algorithms [8, 20] that exploit simultaneous violations to multiple constraints to select among repairs. Given a set of constraints Σ, we want our error-generation method to control the detectability of errors, and whether they are *exactly-one* or *at-least-one* detectable.

Repairability. The notion of *repairability* provides an estimate of how hard it is to restore a DB with errors to its original, clean state. It is clear that such an estimate depends on the actual algorithm used to repair the data. To propose a practical notion, we concentrate on a specific class of repair algorithms. First, we restrict our attention to repair algorithms that use *value modification*, i.e., they fix violations by modifying one or more values in the dirty DB. Second, we assume that these algorithms rely on a *minimality principle*, according to which repairs that minimally modify the given DB are to be preferred. We notice that the vast majority of algorithms in the recent literature [3, 5, 6, 8, 14, 16, 20, 21, 24] fall in this category.

Consider again Example 1, and assume instance I_d only has the following tuples t_{10}-t_{14}:

t_{10} : Emp(John, Staff, ...) $\quad t_{11}$: Emp(John, Sales, ...)
t_{12} : Emp(John, Sales, ...) $\quad t_{13}$: Emp(John, Mktg, ...)
t_{14} : Emp(John, Mktg, ...)

there are eight violation contexts for constraint dc_1, each one using two tuples with equal name and different department. However, an algorithm that is inspired by the principle of minimal repairs would be primarily interested in knowing that these 8 contexts reduce to a group of 10 cells (the name and dept cells of tuples 10-14). Variants of this notion, initially called *equivalence class* [6], have been formalized under the names of *homomorphism class* [16] and *repair context* [8]. We call this a *context equivalence class*.

Definition 5: Context Equivalence Class – Given a constraint dc over schema S in normal form, and an instance I of S, we say that two cells of I *share a context* if they share a violation context for dc and I. A *context equivalence class* $\mathcal{E}$ for dc and I is an equivalence class of cells induced by the transitive closure of the relation *"share a context"*. □

It is natural to reason about the repairability of a violation based on its context equivalence class. To formalize the notion of repairability, we start from a cell change $\mathsf{ch} = \langle t.A := v_d \rangle$ introducing a detectable error to dc. We assume we have identified the context equivalence class $\mathcal{E}$ for cell $t.A$ and dc. From this, we derive a bag of candidate values, denoted by *candidate-values*$(t.A, \mathcal{E}, \mathsf{dc})$. Then, we define the repairability as the probability of restoring $t.A$ to its original value by uniformly at random picking a value from *candidate-values*$(t.A, \mathcal{E}, \mathsf{dc})$:

Definition 6: Repairable Error and Repairability – Given a cell change $\mathsf{ch} = \langle t.A := v_d \rangle$ that changes the cell $t.A$ in I from a value v_c to v_d and assume $t.A$ belongs to a violation context for constraint dc, call $\mathcal{E}$ the context equivalence class of cell $t.A$ and dc. Let $\mathcal{V}$ = *candidate-values*$(t.A, \mathcal{E}, \mathsf{dc})$ be the bag of *candidate repair values* from $\mathcal{E}$ for $t.A$ and dc. We say ch is a *repairable error* if $v_c \in \mathcal{V}$. The *repairability* of ch is computed by dividing the number of occurrences of v_c in $\mathcal{V}$ by the size of $\mathcal{V}$. □

The definition of function *candidate-values*$(t.A, \mathcal{E}, \mathsf{dc})$ is elaborate but quite standard in data repairing. For the sake of readability, we introduce it by means of examples.

(*i*) Consider our sample equivalence class for tuples t_{10}-t_{14}. For FD dc_1 in Example 2 and change $t_{11}.\mathsf{Dept}$ from *"Sales"* to *"Staff"*, we select all values from cells of the Dept attribute in the equivalence class: $\{\mathsf{Staff}, \mathsf{Staff}, \mathsf{Sales}, \mathsf{Mktg}, \mathsf{Mktg}\}$. The error is repairable, and the repairability is $1/5 = 0.2$.

(*ii*) For a CFD, like dc_3 in Example 2 (employees of the staff department need to have salaries of \$1000), and change $t_1.\mathsf{Salary}$ from 1000 to 2000, the only candidate value is dictated by the constant, and is exactly 1000. Therefore, the repairability is 1. This is similar to fixing rules. Editing rules like dc_4 in Example 2 use master-data tuples, i.e., immutable cells, and are treated accordingly: candidate values are taken from the master-data cells only.

(*iii*) Finally, consider ordering constraints like dc_5 in Example 2 (managers have higher salaries than their employees). Assume we change Paul's salary to 2000 in t_2 to make it higher than his manager's salary. It is easy to see that there are infinite real values that can be used to repair the violation. In this case, the repairability is 0.

Notice that a change may be detectable by several constraints, and thus have different repairability values. In this case, we consider the maximum of these values.

4. FORMALIZATION OF THE PROBLEM

Recall that, given an instance I of S and a set Σ of DCs, to detect violations we find assignments that generate violation contexts. This can be done by running a *violation-detection query* for dc and I. Recall that *id* is an attribute storing the tuple id of a relation. For a constraint dc with multiple relation atoms, we abuse notation by using $\bar{id}$ in queries to denote a vector of variables bound to all tuple ids from these atoms. Given a DC of the form $\mathsf{dc} : \neg(\phi(\bar{id}, \bar{x}))$, we denote the context variables (i.e., variables that appear in a comparison atom) of dc by $\bar{x}_c$, and the rest by $\bar{x}_{nc}$.

Definition 7: Vio-Detection Query – The *violation-detection (vio-detection) query* for dc is a conjunctive query with free variables $\bar{id}, \bar{x}_c$ of the form: $DQ_{\mathsf{dc}}(\bar{id}, \bar{x}_c) = \phi(\bar{id}, \bar{x}_c, \bar{x}_{nc})$. □

Consider our example dc_1, its vio-detection query is the following (notice how we add the predicate $\mathsf{id} < \mathsf{id'}$ to avoid returning every pair of tuples twice):

$$DQ_{\mathsf{dc}_1}(\mathsf{id}, \mathsf{id'}, \mathsf{n}, \mathsf{n'}, \mathsf{d}, \mathsf{d'}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}, \mathsf{d}, \mathsf{s}, \mathsf{m}),\ \mathsf{Emp}(\mathsf{id'}, \mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}), \mathsf{n} = \mathsf{n'}, \mathsf{d} \neq \mathsf{d'}, \mathsf{id} < \mathsf{id'}$$

Injecting detectable errors into a clean DB requires the identification of cells that can be changed in order to trigger violations. To find cells with this property, we notice that each comparison in the normal form of a DC suggests a different strategy to generate errors.

Consider $\mathsf{dc}_1 : \neg(\mathsf{Emp}(\mathsf{n}, \mathsf{d}, \mathsf{s}, \mathsf{m}), \mathsf{Emp}(\mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}), \mathsf{n} = \mathsf{n'}, \mathsf{d} \neq \mathsf{d'})$ stating that there cannot exist two employees with equal names and different departments. Given a clean DB, there are two ways to change it to violate this constraint.

(*i*) *Enforce the inequality*: we may find tuples with equal names, and equal departments ($\mathsf{n} = \mathsf{n'}, \mathsf{d} = \mathsf{d'}$), and change one of the departments in such a way that they become different (d becomes different from $\mathsf{d'}$).

(*ii*) *Enforce the equality*: we may find tuples with different names, and different departments ($\mathsf{n} \neq \mathsf{n'}, \mathsf{d} \neq \mathsf{d'}$) and change one of the names in such a way that n becomes equal to $\mathsf{n'}$.

Based on this intuition, we introduce the notion of a *violation-generation query* (vio-gen query in the following). These queries are obtained from a violation-detection query for a DC by negating one of the comparisons.

Definition 8: Vio-Gen Queries – Given a constraint dc in normal form, consider its vio-detection query, DQ_{dc}. A *vio-gen query* $GQ_{\mathsf{dc},i}$ for dc is obtained from DQ_{dc} by changing a comparison atom of the form $v_1^i \,\mathsf{op}\, v_2^i$ into its negation, $\neg(v_1^i \,\mathsf{op}\, v_2^i)$. The latter is called a *target comparison.* □

In our example, we have two vio-gen queries, as follows (target comparisons are enclosed in brackets):

$$GQ_{\mathsf{dc}_1,1}(\mathsf{id}, \mathsf{id'}, \mathsf{n}, \mathsf{n'}, \mathsf{d}, \mathsf{d'}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}, \mathsf{d}, \mathsf{s}, \mathsf{m}),\ \mathsf{Emp}(\mathsf{id'}, \mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}), \mathsf{n} = \mathsf{n'}, (\mathbf{d} = \mathbf{d'}), \mathsf{id} < \mathsf{id'}$$

$$GQ_{\mathsf{dc}_1,2}(\mathsf{id}, \mathsf{id'}, \mathsf{n}, \mathsf{n'}, \mathsf{d}, \mathsf{d'}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}, \mathsf{d}, \mathsf{s}, \mathsf{m}),\ \mathsf{Emp}(\mathsf{id'}, \mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}), (\mathbf{n} \neq \mathbf{n'}), \mathsf{d} \neq \mathsf{d'}, \mathsf{id} < \mathsf{id'}$$

Definition 9: Vio-Gen Cell – Given an instance I, a *vio-gen cell* for vio-gen query $GQ_{\mathsf{dc},i}$ is a cell in the result of $GQ_{\mathsf{dc},i}$ over I that is associated with a variable in the target comparison of $GQ_{\mathsf{dc},i}$. □

Consider our DB in Example 1. The two vio-gen queries for dc_1 identify cells $t_4.\mathsf{Dept}$ and $t_1.\mathsf{Name}$ that can be used to inject errors, as follows:

$GQ_{\mathsf{dc}_1,1}$ result	*change*	*after change*
t_4 : Emp(Frank, Mktg, ...)	t_4.Dept	t_4 : Emp(Frank, **xxx**, ...)
t_5 : Emp(Frank, Mktg, ...)	:= *"xxx"*	t_5 : Emp(Frank, Mktg, ...)

$GQ_{\mathsf{dc}_1,2}$ result	*change*	*after change*
t_1 : Emp(John, Staff, ...)	t_1.Name	t_1 : Emp(**Paul**, Staff, ...)
t_2 : Emp(Paul, Sales, ...)	:= *"Paul"*	t_2 : Emp(Paul, Sales, ...)

Query $GQ_{\mathsf{dc}_1,1}$ identifies cells $t_4.\mathsf{Dept} = t_5.\mathsf{Dept} =$ *"Mktg"*. By making these cells different, we are guaranteed to introduce a violation (the tuples have equal names). Similarly, query $GQ_{\mathsf{dc}_1,2}$ captures the fact that we can introduce a detectable violation by equating cells t_1.Name = *"John"*, t_3.Name = *"Paul"* (since they have different departments).

A few remarks are in order. Vio-gen queries also identify a context for each vio-gen cell, i.e., a set of cells that are the image of variables in comparison atoms. For example, the context of cell t_4.Dept is composed of cells $\{t_4.\mathsf{Dept}, t_4.\mathsf{Name}, t_5.\mathsf{Dept}, t_5.\mathsf{Name}\}$. After we have identified a vio-gen cell, then we need to identify an appropriate value to generate the actual cell change and update the DB. Together, a vio-gen cell and a context determine which values can be used to update the cell. We need to keep track of contexts to avoid new changes that are accidentally repairing other violations. These aspects will be discussed in the next section.

We now define the *error-generation problem* for an error-generation task $\mathbf{E} = \langle \mathsf{S}, \Sigma, I, \mathsf{Conf} \rangle$. For each constraint $\mathsf{dc} \in \Sigma$, a parameter $\epsilon(\mathsf{dc})$ in Conf determines the number of detectable errors that should be introduced in I.

Definition 10: Error-Generation Problem – Given an *error-generation task* $\mathbf{E} = \langle \mathsf{S}, \Sigma, I, \mathsf{Conf} \rangle$, find $\epsilon(\mathsf{dc})$ *at-least-one* (resp. *exactly-one*) detectable errors in I for each $\mathsf{dc} \in \Sigma$. □

It turns out that both variants of the error-generation problem are NP-complete.

Theorem 1: *The* exactly-one *and* at-least-one *error-generation problems for a task* $\mathbf{E}$ *are NP-complete.*

Before we present algorithms and optimizations to solve the error-generation problem, we observe that the vio-gen queries for a constraint dc are nothing else than vio-detection queries for DCs that can be considered as being *"dual"* to dc. Consider constraint dc_1. The two vio-gen queries correspond to the detection queries of constraints $\mathsf{dc}_1^1, \mathsf{dc}_1^2$:

$$\begin{aligned}
\mathsf{dc}_1 &: \neg(\mathsf{Emp}(\mathsf{n},\mathsf{d},\mathsf{s},\mathsf{m}), \mathsf{Emp}(\mathsf{n'},\mathsf{d'},\mathsf{s'},\mathsf{m'}), \mathsf{n}=\mathsf{n'}, \mathsf{d}\neq\mathsf{d'})\\
\mathsf{dc}_1^1 &: \neg(\mathsf{Emp}(\mathsf{n},\mathsf{d},\mathsf{s},\mathsf{m}), \mathsf{Emp}(\mathsf{n'},\mathsf{d'},\mathsf{s'},\mathsf{m'}), \mathsf{n}=\mathsf{n'}, \mathbf{d}=\mathbf{d'})\\
\mathsf{dc}_1^2 &: \neg(\mathsf{Emp}(\mathsf{n},\mathsf{d},\mathsf{s},\mathsf{m}), \mathsf{Emp}(\mathsf{n'},\mathsf{d'},\mathsf{s'},\mathsf{m'}), \mathbf{n}\neq\mathbf{n'}, \mathsf{d}\neq\mathsf{d'})
\end{aligned}$$

Since DCs are closed with respect to the negation of comparison atoms, this is a general property: any vio-gen query for constraint dc coincides with the vio-detection query of a constraint dc′. This highlights the *duality* between queries for injecting new errors and those for detecting errors.

5. THE VIO-GEN ALGORITHM

We now develop a *vio-gen* algorithm to solve the error-generation problem. To efficiently generate large erroneous datasets, we aim for a solution that is in PTIME and scales to large databases. Our algorithm is a correct, but not complete, solution for the *at-least-one* error-generation problem. When it succeeds, it returns a solution to the input task.

We use vio-gen queries to identify cells to change; also, we avoid interactions between cell changes by enforcing that two cell changes cannot share a context. Thus, our algorithm may fail to find a solution if there is no solution without shared contexts. However, as long as there are sufficient vio-gen cells and non-overlapping contexts available, our algorithm will succeed. Intuitively, whether this is the case depends mainly on the requested error ratios (and to a lesser extent on the constraints); in practice, ratios are typically ∼1%-10% of the DB size, and therefore the probability of success is very high. The main tasks are as follows.

Task 1: Finding Vio-Gen Cells and Contexts. Given dc and I, we generate the vio-gen queries for dc. Then, we execute each vio-gen query over I to identify a set of vio-gen cells (possibly all), and the associated violation contexts.

Task 2: Value Whitelists and Blacklists. For a vio-gen cell and one of its contexts, we need to find a set of values that would inject errors into the cell. That is, our algorithm first determines the context and then determines the actual changes that are admissible in this context. Here we make use of a *value whitelist* (candidate values) and *value blacklist* (values that cannot be used). Admissible values for a cell are obtained by the set difference between the cell's whitelist and blacklist. These depend on the constraint, the context, and the target comparison. In our previous examples:

(i) The target comparison of query $GQ_{\mathsf{dc}_1,1}$ is an equality ($d = d'$), and we want to generate changes that falsify the equality; therefore, once we have identified cell t_4.Dept and its value, *"Mktg"*, the whitelist contains any string – denoted by *"*"* – and the blacklist is the single value {*"Mktg"*}.

(ii) The target comparison of query $GQ_{\mathsf{dc}_1,2}$ is a not-equal ($n \neq n'$), and our changes should falsify it; therefore, once we have identified cell t_1.Name with value *"John"*, and the value *"Paul"* in another cell t_2.Name which shares a violation context with t_1.Name, the whitelist for cell t_1.Name contains {*"Paul"*}, the blacklist contains the original value {*"John"*}.

The process of identifying values to change vio-gen cells has some additional subtleties. Consider query $GQ_{\mathsf{dc}_2,3}$ for constraint dc_2 in our running example:

$$\begin{aligned}GQ_{\mathsf{dc}_2,3}(\mathsf{id},\mathsf{id'},\ldots) = \mathsf{Emp}(\mathsf{id},\mathsf{n},\mathsf{d},\mathsf{s},\mathsf{m}), \mathsf{Emp}(\mathsf{id'},\mathsf{n'},\mathsf{d'},\mathsf{s'},\mathsf{m'}),\\ \mathsf{n}=\mathsf{n'}, (\mathbf{d}\neq\mathbf{d'}), \mathsf{d} = \textit{“Sales”}, \mathsf{m}\neq\mathsf{m'}\end{aligned}$$

Here, the value of variable d in the target comparison is constrained to be *"Sales"*. As a consequence, we cannot change its value. This general rule is necessary for detectability: any change we make to the DB for a vio-gen query must break the target comparison but preserve all other comparisons within the query. Therefore, we can only change the value of d' and make it equal to *"Sales"*, that is, the whitelist of vio-gen cells for d' will only contain that value.

Overall, we determine valid values for vio-gen cells by computing equivalence classes based on equality comparisons and use an interval algebra for ordering constraints.

Task 3: Handling Interactions. The purpose of this step is to avoid possible interference among different changes to the DB. To see the problem, following our discussion of dc_1, suppose vio-gen query $GQ_{\mathsf{dc}_1,1}$ suggests a change to cell t_4.Dept to make it equal to *"xxx"*. Assume after that query $GQ_{\mathsf{dc}_1,2}$ suggests a change to the Name cell of the same tuple, to make it equal to *"Paul"*. If we apply both changes to the DB, we get the following instance:

clean DB	*change*	*after changes*
t_4: Emp(Frank, Mktg, . . .)	t_4.Dept := *"xxx"*	t_4: Emp(**Paul**, **xxx**, . . .)
t_5: Emp(Frank, Mktg, . . .)	t_4.Name := *"Paul"*	t_5: Emp(Frank, Mktg, . . .)

It is easy to see that the instance obtained after these two changes is not dirty for dc_1, since the second change is removing the detectable error introduced by the first one.

Our algorithm, while generating changes also stores their violation contexts in main memory. We discard a candidate change that may remove violations introduced by previous changes. More precisely, a vio-gen cell is changed only if: (i) the cell itself does not belong to the context of any other previous change; (ii) none of the cells in its context have been changed by previous changes.

Task 4: Generating Random Changes. Bart generates the desired percentage of detectable errors for a constraint and an attribute in the task configuration. Recall that error-percentages are expressed wrt the size of the DB table, e.g., 1% of errors in the cells of attribute A in a table R of 100K tuples corresponds to 1000 errors.

A crucial requirement is that, during this process, vio-gen cells that may be needed later on are not discarded. In fact, we do not know if the clean DB contains a sufficient number of non-interacting vio-gen cells. Suppose we ask for 1000 errors for an attribute, but there are fewer vio-gen cells. In this case, Bart will generate as many changes as possible, and log a warning about the impossibility of meeting the configuration parameter. To handle this, we resorted to a main-memory sampling algorithm. The algorithm works as follows. Assume we want to generate n changes for vio-gen query GQ and attribute R.A:

(*i*) Before executing GQ, we select a random probability p_{GQ} within a range that can be configured by the user (usually 10% to 50%), and a random offset o_{GQ}, again randomly generated in a range from 1% to 10% of the size of R; these parameters will be used to sample cells at query execution.

(*ii*) Then, we execute the GQ, and scan its result set; for each tuple t, we extract the vio-gen cell and vio-gen context, and check if it overlaps with the ones generated so far; in this case, we discard the tuple; otherwise, we consider the tuple as a candidate to generate a detectable change.

(*iii*) A candidate tuple can be either processed or skipped; notice however that tuples that are skipped are not lost, but rather stored in a queue of candidate tuples, in case we need them later on; first, we skip the first o_{GQ} tuples that would generate valid vio-gen contexts; after this, for each candidate tuple we draw a random number *random*(t): we process the tuple and generate a new change whenever $random(t) < p_{GQ}$; otherwise, we store the tuple in the candidate queue.

(*iv*) The process stops as soon as we have generated n changes; if, however, no more tuples are available in the query result before this happens, then we go back to the queue of candidate tuples, and iterate the process.

We now state a correctness result. The proof and pseudocode for the algorithm are in our technical report [1].

Theorem 2: *Given an error-generation task* $\mathbf{E} = \langle \mathsf{S}, \Sigma, I, \mathsf{Conf}\rangle$*, call* $\mathcal{C}h$ *the output of the vio-gen algorithm over* $\mathbf{E}$*. If the algorithm succeeds, then* $\mathcal{C}h$ *is an at-least-one solution for task* $\mathbf{E}$.

The cost of our *vio-gen* algorithm is dominated by query execution. It is in PTIME in the instance size and the number of constraints and is NP-hard in the size of constraints.

Theorem 3: *Let* $\mathbf{E} = \langle \mathsf{S}, \Sigma, I, \mathsf{Conf}\rangle$ *be an error-generation task. The* vio-gen *algorithm runs in PTIME in* $\|I\|$ *and* $\|\Sigma\|$ *and is NP-hard in the maximal size (number of atoms) of constraints in* Σ.

Note that this is the worst-case complexity of the algorithm. As we show in Section 8, the algorithm performs very well in practice, because using sampling we seldom have to fully evaluate a vio-gen query and the size of constraints is typically limited to a few relational atoms.

Task 5: Checking Detectability and Repairability. Theorem 2 guarantees that the *vio-gen* algorithm generates detectable changes. Our algorithm only guarantees *at-least-one* detectability, but may generate changes that violate multiple constraints. If requested by a user, the system will compute how many constraints are violated by a change. This can be used to output *exactly-one* detectable changes. In addition, it may also compute repairability and filter solutions to guarantee a level of repairability if specified in the task configuration (Definition 6). This is done as follows.

(*i*) After changes have been generated and applied to yield the dirty DB I_d, we run the detection query on I_d for each constraint dc in Σ to find all violation contexts for dc and I_d; recall that this task is essentially the same as Task 1 (vio-gen queries and vio-detection queries are structurally identical); we keep counters for each cell change to know how many violation contexts the cell change occurs in, and how many constraints it violates; notice that the size of the final set of *exactly-one* changes is typically lower than the original number of *at-least-one* detectable changes. Therefore, in some cases the system may need to generate a larger number of changes in order to output a desired number of *exactly-one* detectable changes, as discussed in Section 8.

(*ii*) We compute repairability based on the context equivalence classes (optimizations are discussed in Section 6.2).

6. OPTIMIZATIONS

We now discuss the scalability of the *vio-gen* algorithm. The algorithm is designed to solve all subtasks of the error-generation problem, i.e., find n changes with *at-least-one* detectability, and compute detectability and repairability measures. The main cost factor is related to executing vio-gen queries (in turn, vio-detection queries when measuring detectability and repairability). These techniques do not scale to large DBs, for two main reasons.

Problem 1: Computing Cross-Products. Some constraints may result in vio-generation queries with no equalities. Such queries would end up being executed as cross-products, and therefore have inherent quadratic cost. As an example, consider query $GQ_{\mathsf{dc}_5,1}$ for constraint dc_5:

$$GQ_{\mathsf{dc}_5,1}(\mathsf{id}, \mathsf{id'}, \ldots) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}, \mathsf{d}, \mathsf{s}, \mathsf{m}), \mathsf{Emp}(\mathsf{id'}, \mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}), (\mathbf{m} \neq \mathbf{n'}), \mathsf{s'} < \mathsf{s}$$

This query has a very common pattern, one we have already observed in the vio-gen queries of constraint dc_1.

Problem 2: Computing Expensive Joins. Even when equalities are present, and joins are performed, this may be slow. In fact, the distribution of values in the DB might lead to join results of very large cardinality. Consider an FD d : Emp : Dept → Mngr, stating that department names imply manager names. One of the vio-gen queries for d would be:

$$GQ_{d,1}(\mathsf{id}, \mathsf{id'}, \ldots) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}, \mathsf{d}, \mathsf{s}, \mathsf{m}), \mathsf{Emp}(\mathsf{id'}, \mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}), (\mathsf{d} = \mathsf{d'}), (\mathbf{m} = \mathbf{m'}), \mathsf{id} < \mathsf{id'}$$

As we execute this query on a clean instance I, where equal departments correspond to equal managers, the result of this join may be much larger than the size of table Emp (e.g., when there are few departments).

These problems illustrate that the evaluation of vio-gen queries to generate errors may not scale well to large DBs. We introduce two important optimizations to greatly improve performance. We first exploit the fact that we only have to produce a certain percentage of errors by avoiding the full execution of vio-gen queries involving cross-products. Second, we identify a class of DCs, called symmetric constraints, for which the vio-gen queries can be significantly simplified.

6.1 Sampling Cross-Products

For testing data-cleaning algorithms, often we want to generate a set of errors that is (much) smaller than all errors that could potentially be introduced. Thus, we consider sampling tables, and then computing cross-products over these samples in main memory as an alternative to computing the cross-product using a declarative query. To understand our intuition, consider the typical vio-gen query with inequalities for an FD, for example dc_1:

$$GQ_{\mathsf{dc}_1,2}(\mathsf{id}, \mathsf{id'}, \ldots) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}, \mathsf{d}, \mathsf{s}, \mathsf{m}), \mathsf{Emp}(\mathsf{id'}, \mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}), \\ (\mathsf{n} \neq \mathsf{n'}), \mathsf{d} \neq \mathsf{d'}, \mathsf{id} < \mathsf{id'}$$

We search for tuples with different names and departments. In any DB with a sufficiently large number of tuples, we should have a good probability of finding tuples that differ from each other. Whenever we need to generate n changes:

(*i*) We scan the tables in the query to extract a sample of $c \cdot n$ tuples (c being a configuration parameter), and materialize them in memory.

(*ii*) We compute the cross-product in memory, filter results according to the comparisons $((\mathsf{n} \neq \mathsf{n'}), (\mathsf{d} \neq \mathsf{d'}))$, and use the results for identifying vio-gen cells and their contexts; we stop as soon as n contexts have been found.

(*iii*) If we are not able to find n non-overlapping contexts, then we repeat the process, i.e., we choose a random offset, re-sample the tables and iterate; an iteration limit is used to stop the process before it becomes too expensive.

This strategy has a worst-case cost that is comparable (or even worse) to that of computing the whole cross-product. To avoid this cost, we limit the number of iterations. This may only happen when the cross-product is empty, or has very few results, both very unlikely cases. Our experiments confirm that typically, our optimized version runs orders of magnitude faster than computing the cross-product using the DBMS, even when using the LIMIT clause to reduce the number of results.

6.2 Symmetric Constraints

It is possible to find tuples that violate CFDs by running join-free queries using the group-by operator [5, 12]. A similar technique was used earlier for scalable consistent query answering over schemas containing keys [15]. In this section, we build on these results, and extend them in several ways:

(*i*) We generalize Bohannon et al.'s [5] treatment of CFDs by formalizing the notion of *symmetric denial constraints*; this significantly enlarges the class of constraints for which this optimization can be adopted.

(*ii*) We develop a general algorithm to optimize the execution of symmetric queries with at least one equality and arbitrary inequalities, and show how we can efficiently compute both violation contexts and context equivalence classes.

(*iii*) We conduct an extensive experimental evaluation to show the benefits of this optimization.

We now blur the distinction between DCs, vio-gen queries, and detection queries, and speak simply about their logical formulas. Consider a DC $d : \neg(\phi(\bar{x}))$ in normal form; assume that $\phi(\bar{x})$ has no ordering comparisons $(>, <, \leq, \geq)$, and contains at least one equality. Intuitively, d is *symmetric* if it can be *"broken up"* in two isomorphic subformulas. To formalize this idea, we use the following definition.

Definition 11: Formula Graph – Given $\neg(\phi(\bar{x}))$ in normal form, its *formula graph*, $\mathcal{G}(\phi(\bar{x}))$, is defined as follows:

(*i*) It has a node for every relational atom, comparison atom, variable and constant in $\phi(\bar{x})$.

(*ii*) A node for relational atom $R_i(\bar{x}_i)$ has label R_i; a node for comparison $v \, \mathsf{op} \, v'$ has label op; a node for variable x (constant c) has label x (c, respectively).

(*iii*) There is an edge between the node for atom $R_i(\bar{x}_i)$ and each node for a variable $x \in \bar{x}_i$; if x appears within attribute A, the edge is labeled by $R_i.A$.

(*iv*) There is an edge between the node for atom $v \, \mathsf{op} \, v'$ (v, v' either variable or constant) and the nodes for v, v'. □

We are interested in subformulas that are isomorphic to each other according to some mapping h of the variables. Thus, given h, we use it to break up the graph, and look for isomorphic connected components. Given a mapping of the variables $h : \bar{x} \rightarrow \bar{x}$, we define the *reduced formula graph* for h as the graph obtained from $\mathcal{G}(\phi(\bar{x}))$ by removing all nodes corresponding to comparisons of the form $x \, \mathsf{op} \, h(x)$.

Definition 12: Symmetric Formula – A formula in normal form $\neg(\phi(\bar{x}))$ with no ordering comparisons and at least one equality is *symmetric* with respect to mapping h if its reduced connection graph contains exactly two connected components, and these are isomorphic to each other according to h, i.e., (*i*) a variable label v can be mapped to $h(v)$, (*ii*) every other label is preserved. □

In our example, both dc_1 and dc_2 are symmetric constraints (with mapping $\mathsf{n} \rightarrow \mathsf{n'}, \mathsf{d} \rightarrow \mathsf{d'}, \mathsf{s} \rightarrow \mathsf{s'}, \mathsf{m} \rightarrow \mathsf{m'}$). The formula graph for dc_2 is depicted in Figure 2. Notice how, for the purpose of testing symmetry, we add the implied comparison atom $d' =$ *"Sales"*.

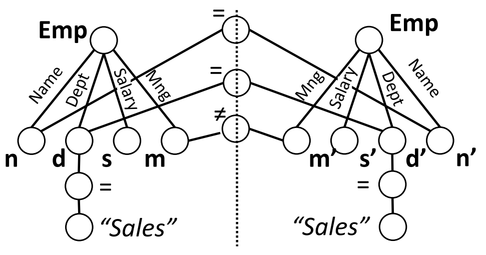

Figure 2: Formula Graph for dc_2

It is easy to see that DCs encoding FDs are all symmetric. This is also true for CFDs, with the exception of single-tuple constraints which do not require joins, like dc_3 in our example. Editing rules, like dc_4, and ordering constraints, like dc_5, on the contrary, are not symmetric.

The vio-detection query for a symmetric constraint is always symmetric. However, not all vio-gen queries for a symmetric constraint need to be symmetric as some may not have equalities. For example, we know dc_1 is symmetric $(\mathsf{n} = \mathsf{n'}, \mathsf{d} \neq \mathsf{d'})$, and among its vio-gen queries, one is symmetric $(\mathsf{n} = \mathsf{n'}, \mathsf{d} = \mathsf{d'})$, the second one is not $(\mathsf{n} \neq \mathsf{n'}, \mathsf{d} \neq \mathsf{d'})$.

Symmetric constraints significantly reduce the number of joins required to execute the corresponding queries. The intuition is that we can only consider one of the isomorphic subqueries, and avoid redundant work. We represent these subcomponents of the original queries by means of relational atoms with *adornments*. An *adornment* for a variable $x \in \bar{x}_i$ is a superscript of the form $=$ or $\neq$, denoted by $x^{=}$ or $x^{\neq}$. We use adornments to track the constraints that are imposed over variables within the original symmetric query.

Given a symmetric formula $\phi(\bar{x})$ in normal form, to optimize its execution we generate a *reduced formula with adornments*, denoted by $reduce(\phi(\bar{x}))$. Following are the reduced formulas for the symmetric vio-gen queries of $\mathsf{dc}_1, \mathsf{dc}_2$ (we use boldface to mark the variables that were involved in the target comparison of the original query):

$reduce(GQ_{\mathsf{dc}_1,1}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}^=, \mathbf{d}^=, \mathsf{s}, \mathsf{m})$
$reduce(GQ_{\mathsf{dc}_2,1}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}^=, \mathsf{d}^=, \mathsf{s}, \mathbf{m}^=), \mathsf{d} = \text{“Sales”}$
$reduce(GQ_{\mathsf{dc}_2,2}) = \mathsf{Emp}(\mathsf{id}, \mathbf{n}^{\neq}, \mathsf{d}^=, \mathsf{s}, \mathsf{m}^{\neq}), \mathsf{d} = \text{“Sales”}$
$reduce(GQ_{\mathsf{dc}_2,3}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}^=, \mathbf{d}^=, \mathsf{s}, \mathsf{m}^{\neq}), \mathsf{d} \neq \text{“Sales”}$

Intuitively, we use $reduce(GQ_{\mathsf{dc}_1,1})$ to derive a (join-free) SQL query that will give us all tuples t in Emp such that there exists a tuple t' with a different tuple id, equal name and equal department. Similarly for $reduce(GQ_{\mathsf{dc}_2,1})$ (equal names, equal departments both *“Sales”*, equal managers). Reduced formulas are constructed using the following algorithm (ids are treated separately).

(i) Start with $\phi(\bar{x})$ and variable mapping h; consider the formula $\phi'(\bar{x}')$ corresponding to one of the connected components of the reduced formula graph for $\phi(\bar{x})$ and h.
(ii) For each comparison $x\,\mathsf{op}\,h(x)$ in $\phi(\bar{x})$, add to variable x in $\phi'(\bar{x}')$ adornment op.

Following is a more complex constraint. It states that an FD dc_6 : Emp : Name $\rightarrow$ Manager holds for tuples of Emp that correspond to master-data tuples on Name and Dept:

$\mathsf{dc}_6 : \neg(\mathsf{Emp}(n, d, s, m), \mathsf{MD}(n', d', m'), n = n', d = d',$
$\mathsf{Emp}(n'', d'', s'', m''), \mathsf{MD}(n''', d''', m'''), n'' = n''', d'' = d''',$
$n = n'', m \neq m'')$

The constraint is symmetric and this is the reduced formula of one of its symmetric vio-gen queries (as usual, we explicitly mention id attributes in the formula) :

$reduce(GQ_{\mathsf{dc}_6,1}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}^{\neq}, \mathsf{d}, \mathsf{s}, \mathsf{m}^{\neq}), \mathsf{MD}(\mathsf{id'}, \mathsf{n'}, \mathsf{d'}, \mathsf{s'}, \mathsf{m'}),$
$\mathsf{n} = \mathsf{n'}, \mathsf{d} = \mathsf{d'}$

6.3 The Benefits of Symmetry

Reduced formulas suggest an alternative query execution strategy that is based on a limited use of joins and favors group-by clauses. This strategy was previously used to find tuples involved in violations of (multi-tuple) CFDs [5]. We extend those algorithms to a larger class of constraints. Our work can handle multiple inequality adornments, while the original technique only considered one inequality at a time.

We first summarize the main ideas behind the use of group-by clauses for formulas with at most one inequality, then discuss the general case.

Simple Case: At Most One Inequality Adornment. Context equivalence classes for symmetric queries with at most one inequality can be computed by grouping tuples. Consider constraint dc_2 and the reduced formula for its symmetric vio-gen query:

$reduce(GQ_{\mathsf{dc}_2,1}) = \mathsf{Emp}(\mathsf{id}, \mathsf{n}^=, \mathsf{d}^=, \mathsf{s}, \mathsf{m}^{\neq}), \mathsf{d} = \text{“Sales”}$

In this case, a vio-gen context consists of two tuples with equal names and departments (equal to *“Sales”*), and different managers. A context equivalence class is composed of all tuples that have equal values of names and departments (*“Sales”*), such that there exist at least two different managers associated with them. Notice that this is a general property. We call the set of variables with equality adornments in a reduced formula its *equality variables* (corresponding to variables equated in original formula).

Property 1: *Given a symmetric constraint* dc *with at most one inequality, and instance I, two cells c_1, c_2 of I belong to the same context equivalence class for* dc *and I if and only if: (i) they belong to contexts vc_1, vc_2 for* dc *and I; (ii) vc_1, vc_2 have equal values for the equality variables of* dc*.* □

To construct contexts and equivalence classes, we generate an SQL query with aggregates, denoted by sym-sql($reduce(Q)$), from the reduced formula. In our example:

```
SELECT id, name, dept, mngr FROM emp WHERE name, dept IN
  (SELECT name, dept FROM emp WHERE dept = 'Sales'
   GROUP BY name, dept HAVING COUNT(DISTINCT mngr) > 1)
ORDER BY name, dept
```

Contexts and equivalence classes are built as follows:
(i) We run query sym-sql($reduce(Q)$).
(ii) We group tuples in the result with equal name and dept attribute values, yielding all context equivalence classes.
(iii) To construct the actual contexts, tuples within an equivalence class are combined in all possible ways in memory.

The General Case. Let us now consider a generic formula with adornments, with at least one equality and multiple inequalities, like the following:

$reduce(GQ_{\mathsf{dc}_2,2}) = \mathsf{Emp}(\mathsf{id}, \mathbf{n}^{\neq}, \mathsf{d}^=, \mathsf{s}, \mathsf{m}^{\neq}), \mathsf{d} = \text{“Sales”}$

Here, we are looking for pairs of tuples that have equal departments (with value *“Sales”*), and *both* different names, *and* different managers. Handling both inequalities makes the construction of contexts more complex. The intuition behind the algorithm is to execute multiple aggregates within our SQL query, one for each inequality, to identify cells that are candidates to generate violation contexts. In our example, sym-sql($reduce(GQ_{\mathsf{dc}_2,2})$) would be the following query:

```
SELECT id, name, dept, mngr FROM emp
WHERE dept, name IN (SELECT dept, name FROM emp
    GROUP BY dept, name HAVING COUNT(DISTINCT name) > 1)
  AND dept, mngr IN (SELECT dept, mngr FROM emp
    GROUP BY dept, mngr HAVING COUNT(DISTINCT mngr) > 1)
ORDER BY dept
```

Notice, however, that Property 1 does not hold in this case. Indeed, belonging to the result of this query is a necessary but not sufficient condition for a cell to be in a violation context for $GQ_{\mathsf{dc}_2,2}$. In fact, we have no guarantee that two tuples from the result of the SQL query above satisfy all inequalities. To select the cells that actually belong to contexts, we need to build the actual contexts, and keep only those in which all inequalities are satisfied.

A crucial optimization, here, is to select a relatively small set of candidate tuples for each context equivalence class. To do this, we group the tuples in the result of the query on the values of the equality variable (dept in our example). Then, we combine the candidate tuples within each set to generate the actual violation contexts.

Once the contexts have been generated, building the context equivalence classes is straightforward: it suffices to hash contexts based on the values of the equality variables, and compute equivalence classes from the cells in each bucket. We show experimentally in Section 8 that this strategy performs very well even for large DBs.

Since symmetry guarantees good performance and helps us avoid extensive sampling, whenever this is compatible with the configuration parameters, our algorithm tries to favor symmetric vio-gen queries over non-symmetric ones.

7. USE CASES OF THE TOOL

Use-Case 1: Generate Detectable Errors. The main

purpose of BART is to generate *constraint-induced errors*. We control an error-generation task **E** by a set of configuration parameters Conf. The main parameters are:
(*i*) *Authoritative sources*: names of DB relations that are to be considered *immutable* (empty by default).
(*ii*) *Error percentages*: desired degree of detectable errors for each vio-gen query. We specify percentages with respect to the number of tuples in a table (e.g., 1% errors in a table of $100K$ tuples means errors are introduced in 1000 cells).
(*iii*) *Repairability range*: users may also specify a range of repairability values for each vio-gen query; BART will estimate the repairability of changes, and only generate errors with estimated repairability within that range. This simplifies the generation of configurations with controlled repairability (e.g., low), as shown in our experiments.

Use-Case 2: Generate Random Errors. In addition to detectable errors, BART may also generate random errors of several kinds: *typos* (e.g., 'databse'), *duplicated values*, *bogus* or *null values* (e.g., '999', '***'), and *outliers*. Random errors may be freely mixed with constraint-induced ones.

Use-Case 3: Computing Repairability. Given a set of constraints Σ and a dirty DB I_d, BART can be used to compute the repairability of the violations in I_d with respect to Σ, as per Definition 6. This can also be done in the case in which I_d was not generated by the tool itself.

Use-Case 4: Measuring Repair Quality. Our ultimate goal is to benchmark data-repairing algorithms over BART data. Suppose we run some algorithm A over a dirty instance I_d as generated by BART, and we obtain a repaired instance $I_{rep,A}$ by a set $\mathcal{C}h_A$ of cell changes, i.e., $I_{rep,A} = \mathcal{C}h_A(I_d)$. The tool adopts a natural strategy to measure the performance of A over a task **E**.

We call $\mathcal{C}h^{-1}$ the set of cell changes that are needed to bring I_d back to its original state, I. Since we assume that I, I_d and $I_{rep,A}$ all have the same set of tuple ids, we define the *quality* of A over **E** as the *F-Measure* of the set $\mathcal{C}h_A$, measured with respect to $\mathcal{C}h^{-1}$. That is, we compute the precision and recall of A in fixing the errors introduced in the original clean instance I. The higher the *F-measure*, the closer $I_{rep,A}$ is to the original clean instance I.

Since data-repairing algorithms have used different metrics to measure the quality of repairs, BART has been designed to be flexible in this respect. Precision and recall may be computed using the following measures:
(*i*) Value: we count the number of cells that have been restored to their original values.
(*ii*) Cell-Var: in addition to cells restored to their original values, we count (with 0.5 score) the cells that have been correctly identified as erroneous, and changed to a variable.
(*iii*) Cell: we count the cells that have been identified as erroneous, regardless of the value assigned by the algorithm.

8. EXPERIMENTAL RESULTS

We describe a detailed evaluation of the BART Java prototype over synthetic and real-world datasets. We ran experiments on a machine with 8GB RAM, 2.6 GHz Intel Core i7, MacOS 10.10, and PostgreSQL 9.3.

Tasks. We tested five tasks, based on synthetic and real datasets (full details are reported in our technical report [1]). We briefly list them here: (*i*) Employees is the running example used in the paper; (*ii*) Customers is a synthetic scenario from Geerts et al. [16]; (*iii*) Tax is a synthetic scenario from Fan et al. [13]; (*iv*) Bus is a real-world scenario from Dallachiesa et al. [11]; and (*v*) Hospital is a real-world scenario used in several data repairing papers (e.g., [11, 14, 16]).

Note that all datasets have different characteristics. Hospital and Bus have higher redundancy in their data. Tax and Employees are the only datasets with constraints containing ordering (<, >) comparisons. Some datasets have master-data and CFDs, while others have only FDs. All these differences help to validate our techniques.

Settings. To measure the scalability of error generation, we focus on execution times. Each task has been run five times and we report the average execution time. Because our focus is on error generation for testing data-cleaning algorithms, we report our study on different % of errors. We first study the range (1% - 10%), in which most cleaning algorithms can perform reasonably well, and then (25% - 75%). To show the effectiveness of our optimizations for symmetric constraints, we compare them against the standard execution.

Scalability. Figures 3.a-c report BART's execution times on synthetic data sets, over an increasing number of tuples for 1%, 5%, and 10% injected errors, respectively. As expected, execution times increase both with the size of the input (number of tuples) and the number of changes (% of required errors). Our system is very fast, taking at most 6.6 minutes for one million tuples in the worst scenario (10% errors). The same observations apply for real-world data, as we show in the first three pairs of bars in Figure 3.d. Notice that, without our optimizations, the execution of all scenarios exceeds the time threshold we set (30 minutes). For this reason we do not report the data.

Symmetric Queries. For each scenario, we extracted all *symmetric* vio-gen queries and executed them with and without our optimization (we report the number of symmetric queries per scenario in Figure 4). Figure 3.e and f report the execution times over an increasing number of tuples and 5% injected errors for the synthetic datasets, with and without the symmetric optimization, respectively. Figure 3.d reports results for real datasets in the last two pairs of bars, using a 10 minutes timeout for each symmetric vio-gen query. Note how the symmetric strategy scales linearly with the size of the data, while the same observation does not hold for the standard join-based strategy (the DBMS adopts a nested-loop execution plan). Our symmetric optimization runs up to two orders of magnitude faster on these datasets.

Success Rate. Our algorithm trades completeness for scalability. It may fail to return the required number of changes, even if these exists, because it non-optimally uses violation contexts. To gain more insight on this aspect, we studied the behavior of the algorithm when 25%, 50%, and 75% of the size of the DB is required to be made dirty. In Figure 3.g we report the fraction of changes that have been generated. Figure 3.h reports the breakdown of the success rate for every constraint of Customers. To limit the incidence of the distribution of data on the experiment, for each constraint we first computed an estimate of the maximal number of detectable changes that can be generated. Then, when running the actual experiments, we made sure to never ask for a number of changes per constraint higher than this estimate. It is interesting to note that in all scenarios, the algorithm generates 100% of the required changes for scenarios with 25% error rate. This can be considered satisfactory for most error-generation applications. The percentage reduces progressively for 50% and 75% errors.

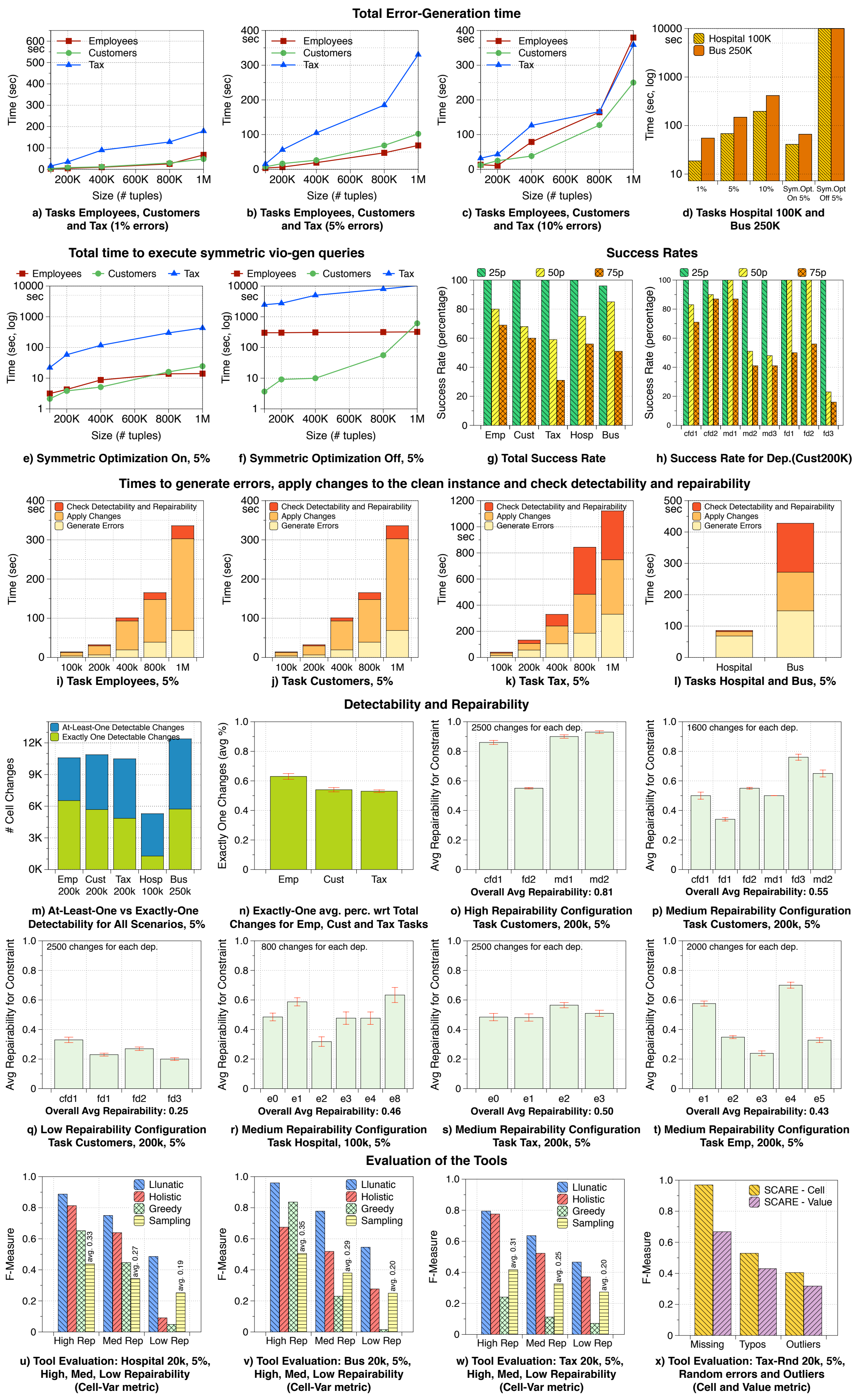

Figure 3: Experimental Results

Datasets	Type and Size	#Tables	#Attributes	#Constraints	#Vio-Gen Queries	#Symmetric Queries	#Non Symmetric Queries	#Queries with No Equalities
Employees	Synthetic, 100K to 1M	2	7	5	12	4	5	3
Customers	Synthetic, 100K to 1M	3	16	8	22	7	12	3
Tax	Synthetic, 100K to 1M	1	13	5	18	12	2	4
Bus	Real 250K	1	5	12	24	10	2	12
Hospital	Real, 100K	1	19	7	16	11	0	5

Figure 4: Datasets and Tasks

Total Execution Time. Figures 3.i-k report total execution times over an increasing number of tuples and 5% injected errors for datasets Employees, Customers, and Tax, respectively. Not surprisingly, most of the time is consumed by the updates to the DB (there are thousands of updates, which are known to be slow). Error generation does not dominate the total time and is stable over the different scenarios, while the computation of detectability and repairability vary significantly depending on the characteristics of the constraints. Again, Tax is the scenario with the most complex constraints, and this is reflected in the execution time for the detection (note the different scale in the y axis). Real-data in Figure 3.l is also consistent with synthetic datasets of the corresponding size in the distribution of the time for the different tasks. Hospital requires less time because of the size and the smaller set of constraints.

Detectability and Repairability. Figure 3.m shows how the ratio between *at-least-one* and *exactly-one* detectable changes depends on the constraints and the data. For example, Hospital has redundant data and overlapping rules, thus it is less likely to find *exactly-one* detectable changes.

To further study this relationship, we used 10 experiments with different DB sizes and error percentages over the three synthetic datasets. We report in Figure 3.n the average percentage of *exactly-one* detectable changes, with 95% confidence intervals. We observe that, given an error-generation task, it is fairly easy to estimate the number of *at-least-one* detectable changes to request in order to obtain a given number of *exactly-one* detectable changes.

Figures 3.o-q report the average repairability of errors in Customers per constraint, using three configurations (i.e., high, medium, and low repairability). Every constraint has the same required amount of errors (5%), and all configurations used the same clean instance of 200k tuples. Intuitively, a high repairability configuration involves mostly rules with master data and CFDs, while a low repairability one involves FDs. Results for medium-repairability configurations on the remaining scenarios are in Figures 3.r-t.

9. DATA REPAIRING TOOL EVALUATION

In this section, we present an empirical comparison of several data-repairing algorithms over BART data. We show a number of novel insights into these algorithms that could not have been shown with existing error generators.

Tools and Algorithms. We used two publicly available tools, namely Llunatic [17] and Nadeef [11], to run four data-repairing algorithms: (*i*) Greedy [6, 9]; (*ii*) Holistic [8]; (*iii*) Llunatic, the chase-based algorithm [16]; and (*iv*) Sampling [3]. In addition, we obtained a copy of the (*v*) SCARE [25] statistics-based tool from the authors.

Tasks. We tested four tasks, three of these constraint-based and one statistics-based (i.e., random errors). For testing constraint-based algorithms, we used (*i*) Hospital, (*ii*) Bus, and (*iii*) Tax. We restricted the set of DCs to FDs and CFDs as only these can be handled by the algorithms under consideration. We selected a clean instance of 20K tuples, and made it dirty with 5% errors and different repairability levels: High (approximately 0.8 rep.), Med (0.5 rep), and Low (0.25 rep.). For testing SCARE [25], we used task (*iv*) Tax-Rnd. We injected errors of different kinds in the Tax dataset: (*a*) 5% missing values, (*b*) 5% typos, and (*c*) 5% outliers over numerical attributes.

We only report results for the constraint-based algorithms over detectable errors (tasks (*i*)–(*iii*)), and for SCARE over random errors (task (*iv*)). As expected, the performance of the algorithms is quite poor when they are applied to errors that are outside of their scope.

Results. The purpose of this evaluation is not to assess the quality of repair algorithms, rather to show how BART can be used to uncover new insights into the data-repairing process. We measured the quality of repairs in terms of precision/recall using the Value, Cell-Var, and Cell metrics as explained in Section 7. When multiple repairs were returned, we chose the best one. We show the results in Figure 3.u–x.

(*a*) We notice a wide degree of variability in quality among all algorithms, from excellent repairs to low-quality ones (a trend observed in Figures 3.u–w). This variability does not clearly emerge from evaluations reported in the literature, an observation that suggests there is no definitive data-repairing algorithm yet.

(*b*) We observe different trends with respect to repairability: (*b*.1) some of the algorithms return very good repairs when sufficient information is available (i.e., high repairability); however, their quality tends to degrade quickly as repairability decreases; (*b*.2) on the contrary, the Sampling algorithm [3] – for which we return the best and average quality among a random sample of 500 repairs – is less affected by different levels of repairability.

(*c*) For Task (*iv*) (as shown in Figure 3.x), (*c*.1) different kinds of random errors pose challenges of different complexity, with missing values being the easiest ones to detect; (*c*.2) interestingly, these errors are more easily detected than fixed, as shown by the differences between the Cell and Value metrics (the first one only counts cells that have been correctly identified as erroneous, while the second one requires that they have been restored to their original value).

A key observation that emerges from this study is that repairability has a strong correlation with the quality of the repairs, thus capturing the *"hardness"* of the problem. Our study also shows the importance of having systematic error-generation tools for evaluating data-repairing solutions.

10. RELATED WORK

Many researchers have had to consider the problem of injecting errors into clean data in order to evaluate a cleaning method. Typically, these approaches inject random errors by scanning DB attributes involved in data-quality rules and changing attribute values with some fixed probability [3, 6, 8, 11, 16]. While often not described in detail, none of this work claims to control the properties of errors like their detectability or repairability. In some approaches, each (individual) injected error is (*at-least-one*) detectable [9], but no guaranteed is made that a set of X errors introduces X detectable errors. No claim is made that their error injection process would work for constraints beyond those considered in their experiments or that it is scalable.

Missing Answers and Why-Not Provenance. Given a query Q and a DB I, a solution to the missing answer problem is an explanation for why $Q(I)$ does not include one or more tuples. *Instance-based* approaches [18, 19] determine how the input instance can be changed to make the missing answer appear in the result. We view the problem of introducing errors into a clean instance as an instance-based missing answer problem, i.e., how to modify the DB to make vio-detection query results non-empty. As we strive to perform only updates of attributes values, approaches that insert or delete tuples to explain missing answers are inapplicable to our problem. While Huang et al.'s technique [19] supports updates, it is known to produce some incorrect answers (see [18]). How-To queries as supported by Tiresias [22] could also be used to encode error generation. Since these approaches rely on constraint solvers and on possible world semantics, they need to consider multiple solutions and minimize the presence of non-expected tuples in the queries (i.e., side-effects). In contrast, by applying *at-least-one* detectability we do not have to compute multiple solutions and minimize side-effects. Furthermore, we introduce a novel optimization for symmetric queries which has not been considered by missing answer approaches.

View Update. The missing answer problem and also ours, are essentially a new take on the view update problem [2]. The problem of introducing errors is equivalent to inserting tuples into views corresponding to the violation-detection queries. Here we only consider approaches that translate view insertions into updates to the base data. While an overview of this problem is beyond the scope of the present work, two examples are the work of Shu [23], who proposes to encode the problem as constraint satisfaction, and Cong et al. [10], who study the complexity of the problem and its relationship to annotation propagation. One major cost factor in view update is computing a translation that minimizes the side-effects on the view and/or the instance. We avoid that cost with our *at-least-one* detectability semantics.

11. CONCLUSIONS AND FUTURE WORK

We have presented the first scalable error-generation tool that provides a high degree of control over the generation process. BART generates errors by value modification and we are currently exploring the introduction of insertions and deletions. The main technical challenge to be solved is that this completely changes the nature of the quality metric and in its full generality, would require identifying tuple homomorphisms among two databases, a problem for which there are currently no scalable algorithms.

12. REFERENCES

[1] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Error Generation for Evaluating Data Cleaning Algorithms. Technical Report TR-01-2015 – *http://db.unibas.it/projects/bart/files/TR-01-2015.pdf*, Università della Basilicata, 2015.

[2] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM TODS*, 6(4):557–575, 1981.

[3] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the Repairs of Functional Dependency Violations under Hard Constraints. *PVLDB*, 3:197–207, 2010.

[4] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. In *ICDE*, pages 506–515, 2007.

[5] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional Functional Dependencies for Data Cleaning. In *ICDE*, pages 746–755, 2007.

[6] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *SIGMOD*, pages 143–154, 2005.

[7] L. Bravo, W. Fan, and S. Ma. Extending Dependencies with Conditions. In *VLDB*, pages 243–254, 2007.

[8] X. Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations into Context. In *ICDE*, pages 458–469, 2013.

[9] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving Data Quality: Consistency and Accuracy. In *VLDB*, pages 315–326, 2007.

[10] G. Cong, W. Fan, F. Geerts, J. Li, and J. Luo. On the Complexity of Annotation Propagation and View Update Analyses. *IEEE TKDE*, 24(3):506–519, 2012.

[11] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a Commodity Data Cleaning System. In *SIGMOD*, pages 541–552, 2013.

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

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

[14] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards Certain Fixes with Editing Rules and Master Data. *PVLDB*, 3(1):173–184, 2010.

[15] A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient Management of Inconsistent Databases. In *SIGMOD*, pages 155–166, 2005.

[16] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, pages 232–243, 2014.

[17] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's All Folks! LLUNATIC Goes Open Source. *PVLDB*, 7(13):1565–1568, 2014.

[18] M. Herschel and M. A. Hernández. Explaining Missing Answers to SPJUA Queries. *PVLDB*, 3(1):185–196, 2010.

[19] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the Provenance of Non-Answers to Queries over Extracted Data. *PVLDB*, 1(1):736–747, 2008.

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

[21] A. Lopatenko and L. Bravo. Efficient Approximation Algorithms for Repairing Inconsistent Databases. In *ICDE*, pages 216–225, 2007.

[22] A. Meliou and D. Suciu. Tiresias: the Database Oracle for How-To Queries. In *SIGMOD*, pages 337–348, 2012.

[23] H. Shu. Using Constraint Satisfaction for View Update. *J. Intelligent Inf. Sys.*, 15(2):147–173, 2000.

[24] J. Wang and N. Tang. Towards Dependable Data Repairing with Fixing Rules. In *SIGMOD*, pages 457–468, 2014.

[25] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid. Don't be SCAREd: Use SCalable Automatic REpairing with Maximal Likelihood and Bounded Changes. In *SIGMOD*, pages 553–564, 2013.