Let's Make it Dirty with BART!

Donatello Santoro[1], Patricia C. Arocena[2], Boris Glavic[3], Giansalvatore Mecca[1], Renée J. Miller[2], and Paolo Papotti[4]

[1]Università della Basilicata - Italy, [2]University of Toronto - Canada
[3]Illinois Inst. of Technology - USA, [4]Eurecom - France

(Discussion Paper)

Abstract. In the last few years many automatic or semi-automatic data-repairing algorithms have been proposed in order to improve the quality of a given database. Due to the richness of research proposals, it is important to conduct experimental evaluations to assess each tool's potential. BART is an open-source error-generation system conceived to support thorough experimental evaluations of these data-repairing systems. In this paper we discuss how generating errors in data is a complex problem, with several facets. We introduce the important notions of detectability and repairability of an error, that stand at the core of BART. Then, we show how, by changing the features of errors, it is possible to influence quite significantly the performance of the tools. Finally, we concretely put to work five data-repairing algorithms on dirty data of various kinds generated using BART, and discuss their performance.

1 Introduction

Data quality is a very important concern in data management. To date, many (disparate) automatic and semi-automatic data-cleaning algorithms have been proposed in the database community [13][12][8][5][6][4]. These algorithms come from different inspirations. Most of them are *constraint-based*: they assume that the target database comes with a set of data-quality rules – for example, functional dependencies (FDs) or conditional functional dependencies (CFDs) – and data is repaired to remove violations to these constraints. Others, on the contrary, do not rely on constraints, but rather on statistics-based approaches to identify suspect values or outliers and try to repair them.

Due to the richness of research proposals, it is important to conduct thorough and fair experimental evaluations to assess each tool's potential. In fact, other fields like *entity resolution*, *record linkage*, *schema mapping* and *data exchange* have worked to develop consolidated tools and benchmarks for empirically evaluating algorithms [14] [1] [3]. Thorough evaluation of data-cleaning systems requires systematic control over the amount of errors in a dataset, and over how hard these errors are to repair. Dirty datasets must be paired with a

SEBD 2018, June 24-27, 2018, Castellaneta Marina, Italy. Copyright held by the author(s).

ground-truth clean dataset to enable evaluation of the quality of a repair produced by a cleaning algorithm. To support rigorous empirical evaluations, an error-generation system must be able to generate multiple dirty versions of a dataset with low user effort, and scale to large datasets.

Bart. Bart [2][17] is the first error-generation tool explicitly conceived to support empirical evaluations of data-repairing algorithms as per the requirements outlined above. It takes as input a clean database and a set of data-quality rules, and injects errors into the database. Rules are expressed using the powerful language of *denial constraints* [15] and errors can be of several kinds, such as typos, duplicated values, nulls, and outliers. We show the major components of the system in Figure 1. A user interacts with Bart by creating error-generation tasks, using a GUI or CL interface. These tasks are then interpreted by Bart's error-generation engine to create dirty versions of a clean database.

The system provides the highest possible level of control over the error-generation process. Among other parameters, it allows users to choose 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. Bart is open-source: its codebase is available on GitHub and can be further extended by the community to develop new features and functionalities.

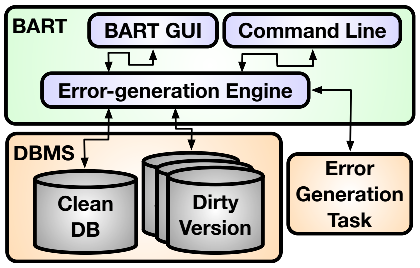


Fig. 1. Bart System Overview

Evaluation Overview. The empirical evaluation convey three primary insights about the system.

(*i*) First, we will discuss how the fine-level control over the features of errors distinguishes Bart from previous error-generating techniques used in evaluating data-repairing systems.

(*ii*) Then, we will discuss how the characteristics of errors may significantly influence the quality of repairs generated by a system.

(*iii*) Finally, we will demonstrate five different algorithms in action on dirty data generated using Bart, to reveal new insights on their (relative) performance as the characteristics of the errors are varied by Bart.

Overall, the attendees will learn how the availability of a tool like Bart may help to level the field and raise the bar for evaluation standards in data cleaning.

The paper is organized as follows. Section 2 introduces the main motivation for the system, and the notions of detectability and repairability of errors. Sec-

Bart: Benchmarking Algorithms for data Repairing and Translation
`https://github.com/dbunibas/BART`

Player	Name	Season	Team	Stadium	Goals
t_1 :	Giovinco	2013-14	Juventus	Juv.Stadium	3
t_2 :	Giovinco	2014-15	Toronto	BMO Field	23
t_3 :	Pirlo	2014-15	Juventus	Juv.Stadium	5
t_4 :	Pirlo	2015-16	N.Y.City	Yankee St.	0
t_5 :	Vidal	2014-15	Juventus	Juv.Stadium	8
t_6 :	Vidal	2015-16	Bayern	Allianz Ar.	3

Fig. 2. Example Clean Database

tion 3 provides an overview of the system and of its main use cases. Finally, Section 4 discusses the empirical evaluation, and the main lessons that can be learned from it.

2 Concept and Motivation

Assume we are given a database about soccer players, shown in Figure 2, and we want to assess the performance of repair algorithms according to a few data-quality rules.

(*i*) A first FD stating that Name and Season are a key for the table: d_1 : Name, Season $\rightarrow$ Team, Stadium, Goals.

(*ii*) And, a second FD stating that Team implies Stadium: d_2 : Team $\rightarrow$ Stadium.

We specify these rules in BART using the language of *denial constraints*. Denial constraints are a very expressive language, capable of capturing most data-quality rules used for data-repairing, including FDs, CFDs, cleaning equality-generating dependencies, editing rules, fixing rules, and ordering constraints [13]. For the sake of simplicity, here we omit the technical details about the syntax and semantics of denial constraints, and show example data-quality rules in the more familiar syntax of FDs.

To evaluate data-repair systems, we proceed as follows.

(*i*) We start with a clean instance I, like the one in Figure 2, and the set of constraints $\Sigma = \{d_1, d_2\}$ discussed above.

(*ii*) We inject errors by applying a set of *cell changes*; each cell change ch $= \langle t_i.A := v \rangle$ updates the value of attribute A in tuple t_i to a new value v, e.g, $\langle t_1.\text{Season} := \text{2011-12} \rangle$. By applying a set of cell changes $\mathcal{C}h$ to I, we obtain a new instance $I_d = \mathcal{C}h(I)$, named the *dirty* instance.

(*iii*) We run a data-repairing algorithm over I_d to obtain a repaired instance I_{rep}. We measure the quality of the algorithm using precision and recall. Intuitively, we count how many changes in $\mathcal{C}h$ have been restored to their original values in I_{rep}. Further details are in the full paper [2].

We want now to emphasize how different ways to change the cells of the clean instance may lead to errors that show completely different features when considered from the perspective of a data-repairing tool.

2.1 Detectability

When evaluating a constraint-based repair algorithm, we want to make sure the errors we inject are detectable by the system in question. After all, an error that cannot be detected, cannot be repaired. To reason about detectability, we need a notion for determining whether a cell change is involved in a constraint violation. Consider the following cell change: $\mathsf{ch}_1 = \langle t_1.\mathsf{Season} := \mathsf{2012\text{-}13} \rangle$ that updates tuple t_1 as follows:

Player	Name	Season	Team	Stadium	Goals
t_1 :	Giovinco	**2012-13**	Juventus	Juv.Stadium	3

This change does not introduce a violation to any of the constraints in Σ, i.e., after the cell change the modified database instances does fulfill all the constraints. Therefore, any data-repairing tool that relies on the constraints to detect dirtiness in the database will not be able to detect the change. We call this an *undetectable change*.

More formally, a cell change $\mathsf{ch} = \langle t_i.A := v \rangle$ in $\mathcal{C}h$ introduces a *detectable error* in I 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. Here "involved in a violation" is defined based on the fact that a constraint can be associated with a query that returns sets of cells that cause a violation. We call this type of queries *violation-detection* queries. A cell is involved in a violation with a constraint dc if it is in the result of the violation-detection query for dc [2]. For example, the violation-detection query for d_2 is $Q_{d_2}(i, i', t, a, a') = \mathsf{Player}(i, n, s, t, a, g),\ \mathsf{Player}(i', n', s', t, a', g'),\ a \neq a',\ i \neq i'$. This query returns the Team and $\mathsf{Stadium}$ attributes of pairs of Player tuples with the same team, but different stadiums. Using the tuple *id*'s, sets of cells involved in violations can be determined based on the result of this query.

BART allows users to control whether changes to the clean database are detectable when using the constraints in Σ. BART may be configured to generate random changes, that do not need to be detectable, or errors that are guaranteed to be detectable using the constraints. Note this requires BART to reason efficiently and holistically about a set of changes to ensure that they are detectable using a given set of constraints.

Interestingly, this latter requirement significantly increases the complexity of the error-generation process. In fact, generating a given number of errors in a clean database that are detectable using a set of constraints Σ is an NP-complete problem [2]. To deal with this complexity BART implements several novel optimizations [2] that balance the need for control over the nature of errors and scalability.

We assume every tuple has a unique identifier that per convention is the first attribute. Queries are expressed in a notation similar to Datalog.

2.2 Repairability

An alternative change that indeed introduces a detectable error is the following: $\mathsf{ch}_2 = \langle t_1.\mathsf{Season} := \mathsf{2014\text{-}15} \rangle$. After this update, tuples t_1 and t_2 violate FD d_1, which states that Name and Season are a key for the table:

Player	Name	Season	Team	Stadium	Goals
t_1 :	Giovinco	**2014-15**	Juventus	Juv.Stadium	3
t_2 :	Giovinco	2014-15	Toronto	BMO Field	23

This change is easily detected using the constraints. Still, it is quite difficult for an automatic data-repairing algorithm to restore the database to its clean state. Notice, in fact, that after this change, the original value 2013-14 has been removed from the active domain of the dirty database. A correct repair cannot be found by any repair algorithm that uses the values in the database as the candidates for repair. Bart uses the notion of *repairability* of an error to characterize this aspect. In the case above, it would assign repairability 0 to change ch_2. Different detectable changes may have quite different repairability values. As an example, consider now change $\mathsf{ch}_3 = \langle t_1.\mathsf{Stadium} := \mathsf{Delle\ Alpi} \rangle$. The change is detectable using FD d_2. In addition, the redundancy in the dirty database may be used to repair the database:

Player	Name	Season	Team	Stadium	Goals
t_1 :	Giovinco	2013-14	Juventus	**Delle Alpi**	3
t_3 :	Pirlo	2014-15	Juventus	Juv.Stadium	5
t_5 :	Vidal	2014-15	Juventus	Juv.Stadium	8

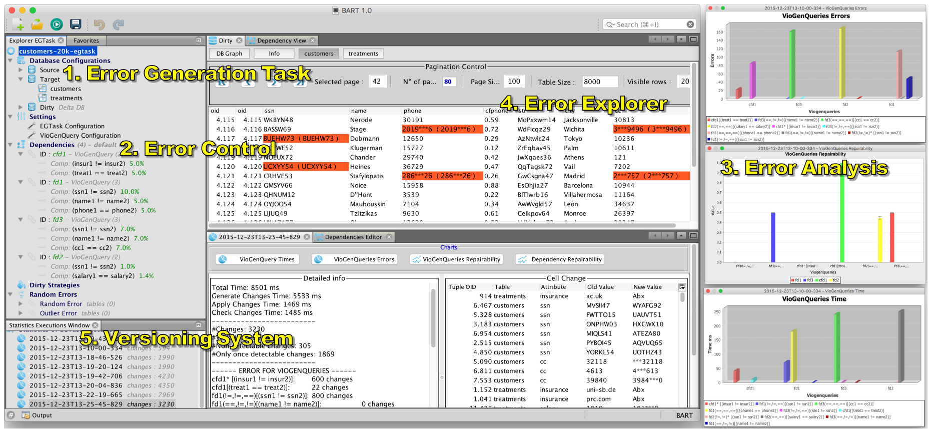
The new, dirty tuple t_1 is involved in two violations to d_2, one with t_3, another with t_5. In both cases, the change is in violation with Juv.Stadium. By a straightforward probabilistic argument, Bart would calculate a 2/3 repairability for this error, and rank it as a medium-repairability error.

Errors may have higher repairability, even 1 in some cases. Consider, for example, an additional rule d_3: $\mathsf{Team}[Juventus]$, $\mathsf{Season}[2013-14] \rightarrow \mathsf{Stadium}[Juv.Stadium]$. This CFD rule states unequivocably that Juventus has played its home games for season 2013–14 in the Juventus Stadium. Since this knowledge is part of the constraint, the dirty cell can easily be restored to its original, clean state.

2.3 Other Kinds of Errors

To conclude this discussion about the features of errors, we notice that the notions of detectability and repairability, that are centered around detecting violations to constraints, are not the only ones supported by Bart.

Consider, for example, change $\mathsf{ch}_4 = \langle t_1.\mathsf{Goals} := 123 \rangle$. This change is not detectable using the constraints. However, it might be detected by a statistics-based data-repairing algorithm, because it introduces an *outlier* into the distribution of values of the Goals attribute. Bart can be configured in order to generate changes of this kind as well.



```
customers-20k-egtask
  Database Configurations
    Source
    Target
      customers
      treatments
    Dirty  Delta DB
  Settings
    EGTask Configuration
    VioGenQuery Configuration
  Dependencies (4) - default
    ID : cfd1 - VioGenQuery (2)
      Comp: (insur1 != insur2)  5.0%
      Comp: (treat1 == treat2)  5.0%
    ID : fd1 - VioGenQuery (3)
      Comp: (ssn1 != ssn2)  10.0%
      Comp: (name1 != name2)  5.0%
      Comp: (phone1 == phone2)  5.0%
    ID : fd3 - VioGenQuery (3)
      Comp: (ssn1 != ssn2)  7.0%
      Comp: (name1 != name2)  7.0%
      Comp: (cc1 == cc2)  7.0%
    ID : fd2 - VioGenQuery (2)
      Comp: (ssn1 != ssn2)  1.0%
      Comp: (salary1 == salary2)  1.4%
  Dirty Strategies
  Random Errors
    Random Error  tables (0)
    Outlier Error  tables (0)
```

```
2015-12-23T1
2015-12-23T13-10-00-334  changes : 594
2015-12-23T13-18-46-526  changes : 1990
2015-12-23T13-19-20-124  changes : 1990
2015-12-23T13-19-42-706  changes : 4230
2015-12-23T13-20-04-836  changes : 4350
2015-12-23T13-22-19-665  changes : 7969
2015-12-23T13-25-45-829  changes : 3230
```

Selected page : 42 N° of pa... 80 Page Si... 100 Table Size : 8000 Visible rows : 20

oid	oid	ssn	name	phone	cfphone	str	city	cc
4.115	4.115	WKBYN48	Nerode	30191	0.59	MoPxxwm14	Jacksonville	30813
4.116	4.116	BASSW69	Stage	2019***6 (2019***6)	0.72	WdFicqz29	Wichita	3***9496 (3***9496)
4.117	4.117	BUEHW73 (BUEHW73)	Dobmann	12650	0.29	AzNtwlc24	Tokyo	10236
4.118	4.118	WE52	Klugerman	15727	0.12	ZrEqbav45	Palm	10611
4.119	4.119	NOEUX72	Chander	29740	0.42	JwXqaes36	Athens	121
4.120	4.120	UCXYY54 (UCXYY54)	Heines	36729	0.47	QqTagsk72	Vail	7202
4.121	4.121	CRHVE53	Stafylopatis	286***26 (286***26)	0.26	GwCsgna47	Madrid	2***757 (2***757)
4.122	4.122	GMSYV66	Noice	15958	0.88	EsOhjia27	Barcelona	10944
4.123	4.123	QHNUM12	D''Hont	3539	0.22	BITlwrb16	Villahermosa	11164
4.124	4.124	OYJOO54	Mauboussin	7104	0.34	AwWvgld57	Leon	34637
4.125	4.125	LIJUQ49	Tzitzikas	9630	0.61	Celkpov64	Monroe	26397

2015-12-23T13-25-45-829 Dependencies Editor

Detailed info

```
Total Time: 8501 ms
Generate Changes Time: 5533 ms
Apply Changes Time: 1469 ms
Check Changes Time: 1485 ms
-------------------------
#Changes: 3230

#Non detectable changes: 305
#Only once detectable changes: 1869
-------------------------
------ ERROR FOR VIOGENQUERIES ------
cfd1* [(insur1 != insur2)]:       600 changes
cfd1[(treat1 == treat2)]:         22 changes
fd1(!=,!=,==)[(ssn1 != ssn2)]:  800 changes
fd1(==,!=,!=)[(name1 != name2)]:          0 changes
```

Cell Change

Tuple OID	Table	Attribute	Old Value	New Value
914	treatments	insurance	ac.uk	Abx
6.467	customers	ssn	MVSII47	WYAFG92
5.328	customers	ssn	FWTTO15	UAUVT51
3.183	customers	ssn	ONPHW03	HXGWX10
6.954	customers	ssn	MIQLS41	ATEZA80
2.515	customers	ssn	PYBOI45	AQVUQ65
4.850	customers	ssn	YORKL54	UOTHZ43
5.090	customers	cc	32118	***32118
6.811	customers	cc	4613	4***613
7.553	customers	cc	39840	3984***0
1.152	treatments	insurance	uni-sb.de	Abx
1.041	treatments	insurance	prc.com	Abx

Fig. 3. User interface

3 Overview of the System

BART provides users with the graphical user interface shown in Figure 3 to handle error-generation tasks. An *error-generation task*, **E** 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*) an instance I of S that is clean with respect to Σ; (*iv*) a set of configuration parameters Conf (shown in Figure 3.1) to control the error-generation process. These parameters specify, among other things, which relations can be changed, 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.

Based on this, BART supports several uses cases. The main one consists of generating a desired degree of detectable errors for each constraint. In addition, users may also specify a range of repairability values for each constraint; BART will estimate the repairability of changes, and only generate errors with estimated repairability within that range. 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', '***'). Random errors may be freely mixed with constraint-induced ones. Finally, BART can introduce outliers in numerical attributes. BART provides sophisticated features to analyze the characteristics of errors that are introduced in the data. It generates charts to analyze the number of errors detected by each constraint, and their estimated repairability (shown in Figure 3.3). It also offers a versioning system, that allows users to generate different dirty databases for the given scenario, and compare the characteristics of their errors.

Finally, BART offers a flexible set of metrics to measure the quality of the repairs generated by a data-repairing tool. In fact, different algorithms can repair data in different ways. For example, some algorithms can produce repairs that mark dirty cells using a variable, while others always restore the dirty instance with a constant value. Different metrics have been proposed and are implemented in Bart to uniformly evaluate these heterogenous changes in the data [4, 11].

4 Empirical Evaluation

We conduct [17] an empirical evaluation of several data-repairing algorithms over dirty data generated using Bart.

We used two publicly available tools, Llunatic [11] and Nadeef [8], to run four rule-based data-repairing algorithms: (*i*) Greedy [5, 7]; (*ii*) Holistic [6]; (*iii*) Llunatic [10]; and (*iv*) Sampling [4]. In addition, we evaluated (*v*) SCARE [18], a statistics-based tool.

The tools were tested with several repair tasks, based on synthetic and real datasets, some of them constraint-based and some statistics-based. We briefly list them here: (*i*) Employees is a synthetic scenario in the full paper [2]; (*ii*) Customers is a synthetic scenario from Geerts et al. [10]; (*iii*) Tax is a synthetic scenario from Fan et al. [9]; (*iv*) Bus is a real-world scenario from Dallachiesa et al. [8]; and (*v*) Hospital is a real-world scenario used in several data-repairing papers (e.g., [8, 10]).

Datasets and constraints have been chosen to exhibit different characteristics. Some have high redundancy in their data. Others contain numerical attributes, and constraints containing ordering ($<$, $>$) comparisons. Some datasets have *master-data* [16] and CFDs, while others have only FDs. All these differences help to validate our techniques and the tools under exam.

Notice the purpose of the evaluation was not to assess the quality of the repair algorithms, rather to show how Bart can be used to uncover new insights into the data-repairing process. Some important insights are the following.

Lesson 1: Data-repairing is not yet Mature. We expect that a wide degree of variability in quality among all algorithms will emerge from our evaluations. This variability does not clearly emerge from evaluations reported in the literature, suggesting there is no definitive data-repairing algorithm yet.

Lesson 2: Repairability Matters. We observe different trends with respect to repairability. Typically, repair algorithms return very good repairs when sufficient information is available (i.e., high repairability); however, their quality tends to degrade quickly as repairability decreases.

A key observation is that repairability has a strong correlation with the quality of the repairs. In this respect, we believe it nicely captures the *"hardness"* of the data-repairing problem and it helps in getting a concrete intuition of the power and the limits of existing solutions.

Lesson 3: We Need to Document Our Dirty Data. We may conclude that tools exhibit quite different performance on data-repairing problems of different nature, and the repairability is a natural proxy to characterize how *"difficult"* a data-repairing problem is.

In light of this and to level the field, we believe it is crucial to have at our disposal systematic error-generation tools and to properly document the characteristics of the dirty data used in empirical evaluations of data-repairing solutions. BART is a concrete step in this direction.

Lesson 4: Generating Errors is Hard. The problem of systematically generating errors, however, is not an easy one. We will show how different configurations of the error-generation task affect the overall scalability of the system, and discuss the main optimizations that BART relies on in order to tame the complexity of the process.

References

1. B. Alexe, W. Tan, and Y. Velegrakis. Comparing and Evaluating Mapping Systems with STBenchmark. *PVLDB*, 1(2):1468–1471, 2008.
2. P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing-Up with BART: Error Generation for Evaluating Data Cleaning Algorithms. *PVLDB*, 9(2):36–47, 2015.
3. M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *PODS*, pages 37–52. ACM, 2017.
4. G. Beskales, I. F. Ilyas, and L. Golab. Sampling the Repairs of Functional Dependency Violations under Hard Constraints. *PVLDB*, 3(1):197–207, 2010.
5. 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.
6. X. Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations into Context. In *ICDE*, pages 458–469, 2013.
7. G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving Data Quality: Consistency and Accuracy. In *VLDB*, pages 315–326, 2007.
8. 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.
9. W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional Functional Dependencies for Capturing Data Inconsistencies. *ACM TODS*, 33, 2008.
10. F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, pages 232–243, 2014.
11. F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's All Folks! LLUNATIC Goes Open Source. *PVLDB*, 7(13):1565–1568, 2014.
12. J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning. In *SIGMOD*, pages 893–907, 2016.
13. I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.
14. H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *VLDB*, 3(1-2):484–493, 2010.
15. A. Lopatenko and L. Bravo. Efficient Approximation Algorithms for Repairing Inconsistent Databases. In *ICDE*, pages 216–225, 2007.
16. D. Loshin. *Master Data Management.* Knowl. Integrity, Inc., 2009.
17. D. Santoro, P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, and P. Papotti. BART in action: Error generation and empirical evaluations of data-cleaning systems. In *SIGMOD*, pages 2161–2164, 2016.
18. 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.