Big Data Cleaning

Nan Tang

Qatar Computing Research Institute, Doha, Qatar
ntang@qf.org.qa

Abstract. Data cleaning is, in fact, a lively subject that has played an important part in the history of data management and data analytics, and it still is undergoing rapid development. Moreover, data cleaning is considered as a main challenge in the era of big data, due to the increasing volume, velocity and variety of data in many applications. This paper aims to provide an overview of recent work in different aspects of data cleaning: error detection methods, data repairing algorithms, and a generalized data cleaning system. It also includes some discussion about our efforts of data cleaning methods from the perspective of big data, in terms of volume, velocity and variety.

1 Introduction

Real-life data is often dirty: Up to 30% of an organization's data could be dirty [2]. Dirty data is costly: It costs the US economy $3 trillion+ per year [1]. These highlight the importance of data quality management in businesses.

Data cleaning is the process of identifying and (possibly) fixing data errors. In this paper, we will focus on discussing dependency based data cleaning techniques, and our research attempts in each direction [5].

Error detection. There has been a remarkable series of work to capture data errors as violations using integrity constraints (ICs) [3, 9, 10, 18, 20, 21, 25, 27] (see [17] for a survey). A violation is a set of data values such that when putting together, they violate some ICs, thus considered to be wrong. However, ICs cannot tell, in a violation, which values are correct or wrong, thus fall short of guiding dependable data repairing. Fixing rules [30] are proposed recently that can precisely capture which values are wrong, when enough evidence is given.

Data repairing. Data repairing algorithms have been proposed [7,8,12–15,23,24, 26,28,31]. Heuristic methods are developed in [6,8,13,26], based on FDs [6,27], FDs and INDs [8], CFDs [18], CFDs and MDs [23] and denial constraints [12]. Some works employ confidence values placed by users to guide a repairing process [8,13,23] or use master data [24]. Statistical inference is studied in [28] to derive missing values, and in [7] to find possible repairs. To ensure the accuracy of generated repairs, [24,28,31] require to consult users. Efficient data repairing algorithms using fixing rules have also been studied [30].

Data cleaning systems. Despite the increasing importance of data quality and the rich theoretical and practical contributions in all aspects of data cleaning,

	name	country	capital	city	conf
r_1:	George	China	Beijing	Beijing	SIGMOD
r_2:	Ian	China	Shanghai (Beijing)	Hongkong (Shanghai)	ICDE
r_3:	Peter	China (Japan)	Tokyo	Tokyo	ICDE
r_4:	Mike	Canada	Toronto (Ottawa)	Toronto	VLDB

Fig. 1. Database D: an instance of schema Travel

there is a lack of *end-to-end* off-the-shelf solution to (semi-)automate the detection and the repairing of violations *w.r.t.* a set of heterogeneous and ad-hoc quality constraints. NADEEF [14, 15] was presented as an extensible, generalized and easy-to-deploy data cleaning platform. NADEEF distinguishes between a *programming interface* and a *core* to achieve generality and extensibility. The programming interface allows the users to specify multiple types of data quality rules, which uniformly define *what* is wrong with the data and (possibly) *how* to repair it through writing code that implements predefined classes.

Organization. We describe error detection techniques in Section 2. We discuss data repairing algorithms in Section 3. We present a commodity data cleaning system, NADEEF, in Section 4, followed by open research issues in Section 5.

2 Dependency-based Error Detection

In this section, we start by illustrating how integrity constraints work for error detection. We then introduce fixing rules.

Example 1. Consider a database D of travel records for a research institute, specified by the following schema:

$$\mathsf{Travel}\ (\mathsf{name}, \mathsf{country}, \mathsf{capital}, \mathsf{city}, \mathsf{conf}),$$

where a Travel tuple specifies a person, identified by name, who has traveled to conference (conf), held at the city of the country with capital. One Travel instance is shown in Fig. 1. All errors are highlighted and their correct values are given between brackets. For instance, $r_2[\mathsf{capital}] = \texttt{Shanghai}$ is wrong, whose correct value is Beijing.

A variety of ICs have been used to capture errors in data, from traditional constraints (*e.g.,* functional and inclusion dependencies [8,10]) to their extensions (*e.g.,* conditional functional dependencies [18]).

Example 2. Suppose that a functional dependency (FD) is specified for the Travel table as:

$$\phi_1\colon \mathsf{Travel}\ ([\mathsf{country}] \rightarrow [\mathsf{capital}])$$

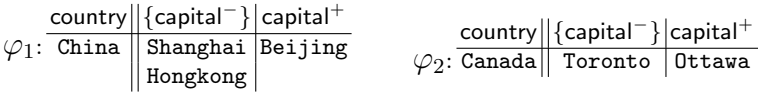


Fig. 2. Example fixing rules

which states that country uniquely determines capital. One can verify that in Fig. 1, the two tuples (r_1, r_2) violate ϕ_1, since they have the same country but carry different capital values, so do (r_1, r_3) and (r_2, r_3).

Example 2 shows that although ICs can detect errors (*i.e.,* there must exist errors in detected violations), it reveals two shortcomings of IC based error detection: (*i*) it can neither judge which value is correct (*e.g.,* t_1[country] is correct), nor which value is wrong (*e.g.,* t_2[capital] is wrong) in detected violations; and (*ii*) it cannot ensure that *consistent* data is correct. For instance, t_4 is consistent with any tuple *w.r.t.* ϕ_1, but t_4 cannot be considered as correct.

Fixing rules. Data cleaning is not magic; it cannot guess something from nothing. What it does is to make decisions from evidence. Certain *data patterns* of semantically related values can provide evidence to precisely capture and rectify data errors. For example, when values (`China, Shanghai`) for attributes (country, capital) appear in a tuple, it suffices to judge that the tuple is about `China`, and `Shanghai` should be `Beijing`, the capital of `China`. In contrast, the values (`China, Tokyo`) are not enough to decide which value is wrong.

Motivated by the observation above, fixing rules were introduced [30]. A fixing rule contains an *evidence pattern*, a set of *negative patterns*, and a *fact* value. Given a tuple, the evidence pattern and the negative patterns of a fixing rule are combined to precisely tell which attribute is wrong, and the fact indicates how to correct it.

Example 3. Figure 2 shows two fixing rules. The brackets mean that the corresponding cell is multivalued.

For the first fixing rule φ_1, its evidence pattern, negative patterns and the fact are `China`, {`Shanghai, Hongkong`}, and `Beijing`, respectively. It states that for a tuple t, if its country is `China` and its capital is either `Shanghai` or `Hongkong`, capital should be updated to `Beijing`. For instance, consider the database in Fig. 1. Rule φ_1 detects that r_2[capital] is wrong, since r_2[country] is `China`, but r_2[capital] is `Shanghai`. It will then update r_2[capital] to `Beijing`.

Similarly, the second fixing rule φ_2 states that for a tuple t, if its country is `Canada`, but its capital is `Toronto`, then its capital is wrong and should be `Ottawa`. It detects that r_4[capital] is wrong, and then will correct it to `Ottawa`.

After applying φ_1 and φ_2, two errors, r_2[capital] and r_4[capital], can be repaired.

Notation. Consider a schema R defined over a set of attributes, denoted by attr(R). We use $A \in R$ to denote that A is an attribute in attr(R). For each attribute $A \in R$, its domain is specified in R, denoted as dom(A).

Syntax. A *fixing rule* φ defined on a schema R is formalized as $((X, t_p[X]), (B, T_p^-[B])) \to t_p^+[B]$ where

1. X is a set of attributes in $\mathsf{attr}(R)$, and B is an attribute in $\mathsf{attr}(R) \setminus X$ (*i.e.,* B is not in X);
2. $t_p[X]$ is a pattern with attributes in X, referred to as the *evidence pattern* on X, and for each $A \in X$, $t_p[A]$ is a constant value in $\mathsf{dom}(A)$;
3. $T_p^-[B]$ is a finite set of constants in $\mathsf{dom}(B)$, referred to as the *negative patterns* of B; and
4. $t_p^+[B]$ is a constant value in $\mathsf{dom}(B) \setminus T_p^-[B]$, referred to as the *fact* of B.

Intuitively, the evidence pattern $t_p[X]$ of X, together with the negative patterns $T_p^-[B]$ impose the condition to determine whether a tuple contains an error on B. The fact $t_p^+[B]$ in turn indicates how to correct this error.

Note that condition (4) enforces that the correct value (*i.e.,* the fact) is different from known wrong values (*i.e.,* negative patterns) relative to a specific evidence pattern.

We say that a tuple t of R *matches* a rule $\varphi : ((X, t_p[X]), (B, T_p^-[B])) \to t_p^+[B]$, denoted by $t \vdash \varphi$, if (i) $t[X] = t_p[X]$ and (ii) $t[B] \in T_p^-[B]$. In other words, tuple t matches rule φ indicates that φ can identify errors in t.

Example 4. Consider the fixing rules in Fig. 2. They can be formally expressed as follows:

φ_1: (([country], [China]), (capital, {Shanghai, Hongkong})) → Beijing
φ_2: (([country], [Canada]), (capital, {Toronto})) → Ottawa

In both φ_1 and φ_2, X consists of country and B is capital. Here, φ_1 states that, if the country of a tuple is China and its capital value is in {Shanghai, Hongkong}, its capital value is wrong and should be updated to Beijing. Similarly for φ_2.

Consider D in Fig. 1. Tuple r_1 does not match rule φ_1, since r_1[country] = China but r_1[capital] $\notin$ {Shanghai, Hongkong}. As another example, tuple r_2 matches rule φ_1, since r_2[country] = China, and r_2[capital] $\in$ {Shanghai, Hongkong}. Similarly, we have r_4 matches φ_2.

Semantics. We next give the semantics of fixing rules.

We say that a fixing rule φ is *applied* to a tuple t, denoted by $t \to_\varphi t'$, if (i) t matches φ (*i.e.,* $t \vdash \varphi$), and (ii) t' is obtained by the update $t[B] := t_p^+[B]$.

That is, if $t[X]$ agrees with $t_p[X]$, and $t[B]$ appears in the set $T_p^-[B]$, then we assign $t_p^+[B]$ to $t[B]$. Intuitively, if $t[X]$ matches $t_p[X]$ and $t[B]$ matches some value in $T_p^-[B]$, it is evident to judge that $t[B]$ is wrong and we can use the fact $t_p^+[B]$ to update $t[B]$. This yields an updated tuple t' with $t'[B] = t_p^+[B]$ and $t'[R \setminus \{B\}] = t[R \setminus \{B\}]$.

Example 5. As shown in Example 3, we can correct r_2 by applying rule φ_1. As a result, r_2[capital] is changed from Shanghai to Beijing, *i.e.,* $r_2 \to_{\varphi_1} r_2'$ where r_2'[capital] = Beijing and the other attributes of r_2' remain unchanged.

Similarly, we have $r_4 \to_{\varphi_2} r_4'$ where the only updated attribute value is r_4'[capital] = Ottawa.

Fundamental problems associated with fixing rules have been studied [30].

Termination. The *termination problem* is to determine whether a rule-based process will stop. We have verified that applying fixing rules can ensure the process will terminate.

Consistency. The *consistency problem* is to determine, given a set Σ of fixing rules defined on R, whether Σ is consistent.

Theorem 1. *The consistency problem of fixing rules is* PTIME.

We prove Theorem 1 by providing a PTIME algorithm for determining if a set of fixing rules is consistent in [30].

Implication. The *implication problem* is to decide, given a set Σ of consistent fixing rules, and another fixing rule φ, whether Σ implies φ.

Theorem 2. *The implication problem of fixing rules is coNP-complete. It is down to* PTIME *when the relation schema R is fixed.*

Please refer to [30] for a proof.

Determinism. The *determinism problem* asks whether all terminating cleaning processes end up with the same repair.

From the definition of consistency of fixing rules, it is trivial to get that, if a set Σ of fixing rules is consistent, for any t of R, applying Σ to t will terminate, and the updated t' is deterministic (*i.e.,* a unique result).

3 Data Repairing Algorithms

In this section, we will discuss several classes of data repairing solutions. We will start by the most-studied problem: computing a consistent database (Sectioin 3.1). We then discuss user guided repairing (Section 3.2) and repairing data with precomputed confidence values (Section 3.3). We will end up this section with introducing the data repairing with fixing rules (Section 3.4).

3.1 Heuristic Algorithms

A number of recent research [4,8,12] have investigated the data cleaning problem introduced in [3]: repairing is to find another database that is consistent and minimally differs from the original database. They compute a consistent database by using different cost functions for value updates and various heuristics to guide data repairing.

For instance, consider Example 2. They can change r_2[capital] from `Shanghai` to `Beijing`, and r_3[capital] from `Tokyo` to `Beijing`, which requires two changes. One may verify that this is a repair with the *minimum* cost of two updates. Though these changes correct the error in r_2[capital], they do not rectify r_3[country]. Worse still, they mess up the correct value in r_3[capital].

	country	capital
s_1:	China	Beijing
s_2:	Canada	Ottawa
s_3:	Japan	Tokyo

Fig. 3. Data D_m of schema Cap

3.2 User Guidance

It is known that heuristic based solutions might introduce data errors [22]. In order to ensure that a repair is dependable, users are involved in the process of data repairing [22, 29, 31].

Consider a recent work [24] that uses editing rules and master data. Figure 3 shows a master data D_m of schema Cap (country, capital), which is considered to be correct. An editing rule eR_1 defined on two relations (Travel, Cap) is:

$\mathsf{eR}_1 : ((\mathsf{country}, \mathsf{country}) \rightarrow (\mathsf{capital}, \mathsf{capital}), t_{p1}[\mathsf{country}] = ())$

Rule eR_1 states that: for any tuple r in a Travel table, if r[country] is correct and it matches s[country] from a Cap table, we can update r[capital] with the value s[capital] from Cap. For instance, to repair r_2 in Fig. 1, the users need to ensure that r_2[country] is correct, and then match r_2[country] and s_1[country] in the master data, so as to update r_2[capital] to s_1[capital]. It proceeds similarly for the other tuples.

3.3 Value Confidence

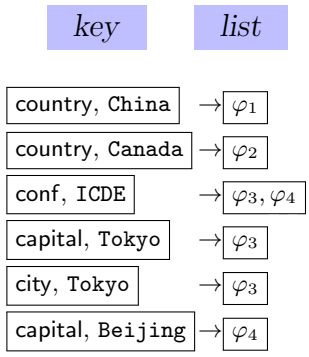
Instead of interacting with users to ensure the correctness of some values or to rectify some data, some work employs pre-computed or placed confidence values to guide a repairing process [8, 13, 23]. The intuition is that the values with high confidence values should not be changed, and the values with low confidence values are mostly probably to be wrong and thus should be changed. These information about confidence values will be taken into consideration by modifying algorithms *e.g.,* those in Section 3.1.

3.4 Fixing Rules

There are two data repairing algorithms using fixing rules that are introduced in Section 2. Readers can find the details of these algorithms in [30]. In this paper, we will give an example about how they work.

Example 6. Consider Travel data D in Fig. 1, rules φ_1, φ_2 in Fig 2, and the following two rules.

φ_3: (([capital, city, conf], [Tokyo, Tokyo, ICDE]), (country, {China})) → Japan
φ_4: (([capital, conf], [Beijing, ICDE]), (city, {Hongkong}) → Shanghai



(a) Inverted lists

r_1: $\Gamma = \{\varphi_1\}$, c(φ_1) = 1
itr1: $r'_1 = r_1$, $\Gamma = \emptyset$

r_2: Γ={φ_1}, c($\varphi_1, \varphi_3, \varphi_4$)=1;
itr1: r'_2[capital]= Beijing
c(φ_3)=1, c(φ_4)=2, Γ={φ_4};
itr2: r'_2[city]= Shanghai , $\Gamma = \emptyset$

r_3: Γ={φ_3}, c(φ_3)=3;
itr1: r'_3[country]= Japan , $\Gamma = \emptyset$

r_4: Γ={φ_2}, c(φ_2)=1;
itr1: r'_4[capital]= Ottawa , $\Gamma = \emptyset$

Fig. 4. A running example

Rule φ_3 states that: for t in relation Travel, if the conf is ICDE, held at city Tokyo and capital Tokyo, but the country is China, its country should be updated to Japan.

Rule φ_4 states that: for t in relation Travel, if the conf is ICDE, held at some country whose capital is Beijing, but the city is Hongkong, its city should be Shanghai. This holds since ICDE was held in China only once at 2009, in Shanghai but never in Hongkong.

Before giving a running example, we shall pause and introduce some indices, which is important to understand the algorithm.

Inverted lists. Each inverted list is a mapping from a *key* to a set Υ of fixing rules. Each key is a pair (A, a) where A is an attribute and a is a constant value. Each fixing rule φ in the set Υ satisfies $A \in X_\varphi$ and $t_p[A] = a$.

For example, an inverted list *w.r.t.* φ_1 in Example 4 is as:

$$\boxed{\text{country, China}} \rightarrow \boxed{\varphi_1}$$

Intuitively, when the country of some tuple is China, this inverted list will help to identify that φ_1 might be applicable.

Hash counters. It uses a hash map to maintain a counter for each rule. More concretely, for each rule φ, the counter c(φ) is a nonnegative integer, denoting the number of attributes that a tuple agrees with $t_p[X_\varphi]$.

For example, consider φ_1 in Example 4 and r_2 in Fig. 1. We have c(φ_1) = 1 *w.r.t.* tuple r_2, since both r_2[country] and t_{p_1}[country] are China. As another example, consider r_4 in Fig. 1, we have c(φ_1) = 0 *w.r.t.* tuple r_4, since r_4[country] = Canada but t_{p_1}[country] = China.

Given the four fixing rules φ_1–φ_4, the corresponding inverted lists are given in Fig. 4(a). For instance, the third key (conf, ICDE) links to rules φ_3 and φ_4, since conf $\in X_{\varphi_3}$ (*i.e.*, {capital, city, conf}) and t_{p_3}[conf] = ICDE; and moreover,

conf $\in X_{\varphi_4}$ (*i.e.,* {capital, conf}) and t_{p_4}[conf] = ICDE. The other inverted lists are built similarly.

Now we show how the algorithm works over tuples r_1 to r_4, which is also depicted in Fig. 4. Here, we highlight these tuples in two colors, where the green color means that the tuple is clean (*i.e.,* r_1), while the red color represents the tuples containing errors (*i.e.,* r_2, r_3 and r_4).

r_1: It initializes and finds that φ_1 may be applied, maintained in Γ. In the first iteration, it finds that φ_1 cannot be applied, since r_1[capital] is Beijing, which is not in the negative patterns {Shanghai, Hongkong} of φ_1. Also, no other rules can be applied. It terminates with tuple r_1 unchanged. Actually, r_1 is a clean tuple.

r_2: It initializes and finds that φ_1 might be applied. In the first iteration, rule φ_1 is applied to r_2 and updates r_2[capital] to Beijing. Consequently, it uses inverted lists to increase the counter of φ_4 and finds that φ_4 might be used. In the second iteration, rule φ_1 is applied and updates r_2[city] to Shanghai. It then terminates since no other rules can be applied.

r_3: It initializes and finds that φ_3 might be applied. In the first iteration, rule φ_3 is applied and updates r_3[coutry] to Japan. It then terminates, since no more applicable rules.

r_4: It initializes and finds that φ_2 might be applied. In the first iteration, rule φ_2 is applied and updates r_4[capital] to Ottawa. It will then terminate.

At this point, we see that all the fours errors shown in Fig. 1 have been corrected, as highlighted in Fig. 4.

4 NADEEF: A Commodity Data Cleaning Systems

Despite the need of high quality data, there is no *end-to-end off-the-shelf* solution to (semi-)automate error detection and correction *w.r.t.* a set of *heterogeneous* and *ad-hoc* quality rules. In particular, there is no commodity platform similar to general purpose DBMSs that can be easily customized and deployed to solve application-specific data quality problems. Although there exist more expressive logical forms (*e.g.,* first-order logic) to cover a large group of quality rules, *e.g.,* CFDs, MDs or denial constraints, the main problem for designing an effective holistic algorithm for these rules is the lack of *dynamic semantics*, *i.e.,* alternative ways about *how* to repair data errors. Most of these existing rules only have *static semantics*, *i.e., what* data is erroneous.

Emerging data quality applications place the following challenges in building a commodity data cleaning system.

Heterogeneity: Business and dependency theory based quality rules are expressed in a large variety of formats and languages from rigorous expressions (*e.g.,* functional dependencies), to plain natural language rules enforced by code embedded in the application logic itself (as in many practical scenarios). Such diversified semantics hinders the creation of one uniform system to accept het-

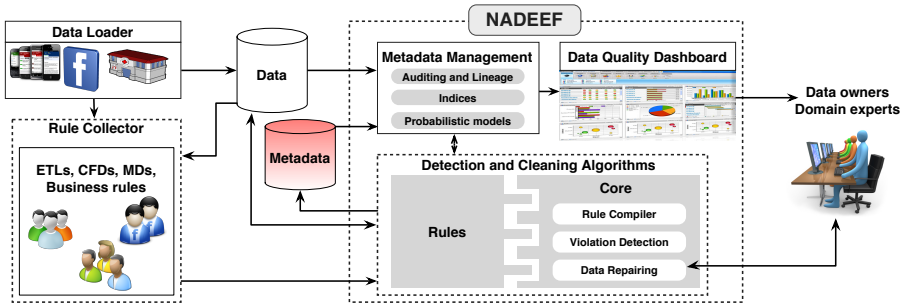


Fig. 5. Architecture of NADEEF

erogeneous quality rules and to enforce them on the data within the same framework.

Interdependency: Data cleaning algorithms are normally designed for one specific type of rules. [23] shows that interacting two types of quality rules (CFDs and MDs) may produce higher quality repairs than treating them independently. However, the problem related to the interaction of more diversified types of rules is far from being solved. One promising way to help solve this problem is to provide unified formats to represent not only the static semantics of various rules (*i.e.,* what is wrong), but also their dynamic semantics (*i.e.,* alternative ways to fix the wrong data).

Deployment and extensibility: Although many algorithms and techniques have been proposed for data cleaning [8, 23, 31], it is difficult to download one of them and run it on the data at hand without tedious customization. Adding to this difficulty is when users define new types of quality rules, or want to extend an existing system with their own implementation of cleaning solutions.

Metadata management and data custodians: Data is not born an orphan. Real customers have little trust in the machines to mess with the data without human consultation. Several attempts have tackled the problem of including humans in the loop (*e.g.,* [24, 29, 31]). However, they only provide users with information in restrictive formats. In practice, the users need to understand much more meta-information *e.g.,* summarization or samples of data errors, lineage of data changes, and possible data repairs, before they can effectively guide any data cleaning process.

NADEEF[1] is a prototype for an extensible and easy-to-deploy cleaning system that leverages the separability of two main tasks: (1) isolating rule specification that uniformly defines *what* is wrong and (possibly) *how* to fix it; and (2) developing a core that holistically applies these routines to handle the detection and cleaning of data errors.

[1] https://github.com/Qatar-Computing-Research-Institute/NADEEF

4.1 Architecture Overview

Figure 5 depicts of the architecture of NADEEF. It contains three components: (1) the *Rule Collector* gathers user-specified quality rules; (2) the *Core* component uses a rule compiler to compile heterogeneous rules into homogeneous constructs that allow the development of default holistic data cleaning algorithms; and (3) the *Metadata management* and *Data quality dashboard* modules are concerned with maintaining and querying various metadata for data errors and their possible fixes. The dashboard allows domain experts and users to easily interact with the system.

Rule Collector. It collects user-specified data quality rules such as ETL rules, CFDs (FDs), MDs, deduplication rules, and other customized rules.

Core. The core contains three components: *rule compiler*, *violation detection* and *data repairing*.

(i) Rule Compiler. This module compiles all heterogeneous rules and manages them in a unified format.

(ii) Violation Detection. This module takes the data and the compiled rules as input, and computes a set of data errors.

(iii) Data Repairing. This module encapsulates holistic repairing algorithms that take violations as input, and computes a set of data repairs, while (by default) targeting the minimization of some pre-defined cost metric. This module may interact with domain experts through the *data quality dashboard* to achieve higher quality repairs.

For more details of NADEEF, please refer to the work [14].

4.2 Entity Resolution Extension

Entity resolution (ER), the process of identifying and eventually merging records that refer to the same real-world entities, is an important and long-standing problem. NADEEF/ER [16] was an extension of NADEEF as a generic and interactive entity resolution system, which is built as an extension over NADEEF. NADEEF/ER provides a rich programming interface for manipulating entities, which allows generic, efficient and extensible ER. NADEEF/ER offers the following features: (1) *Easy specification* – Users can easily define ER rules with a browser-based specification, which will then be automatically transformed to various functions, treated as *black-boxes* by NADEEF; (2) *Generality and extensibility* – Users can customize their ER rules by refining and fine-tuning the above functions to achieve both effective and efficient ER solutions; (3) *Interactivity* – NADEEF/ER [16] extends the existing NADEEF [15] dashboard with summarization and clustering techniques to facilitate understanding problems faced by the ER process as well as to allow users to influence resolution decisions.

4.3 High-Volume Data

In order to be scalable, NADEEF has native support for three databases, PostgreSQL, mySQL, and DerbyDB. However, to achieve high performance for high-

volume data, a single machine is not enough. To this purpose, we have also built NADEEF on top of Spark[2], which is transparent to end users. In other words, users only need to implement NADEEF programming interfaces in logical level. NADEEF will be responsible to translate and execute user provided functions on top of Spark.

4.4 High-Velocity Data

In order to deal with high-velocity data, we have also designed new NADEEF interfaces for incremental processing of streaming data. By implementing these new functions, NADEEF can maximally avoid repeated comparison of existing data, hence is able to process data in high-velocity.

5 Open Issues

Data cleaning is, in general, a hard problem. There are many issues to be addressed or improved to meet practical needs.

Tool selection. Given a database and a wide range of data cleaning tools (*e.g.,* FD-, DC- or statistical-based methods), the first challenging question is which tool to pick for the given specific task.

Rule discovery. Although several discovery algorithms [11, 19] have been developed for *e.g.,* CFDs or DCs, rules discovered by automatic algorithms are far from clean themselves. Hence, often times, manually selecting/cleaning thousands of discovered rules is a must, yet a difficult process.

Usability. In fact, usability has been identified as an important feature of data management, since it is challenging for humans to interact with machines. This problem is harder when comes to the specific topic of data cleaning, since given detected errors, there is normally no evidence that which values are correct and which are wrong, even for humans. Hence, more efforts should be put to usability of data cleaning systems so as to effectively involve users as first-class citizens.

References

1. Dirty data costs the U.S. economy $3 trillion+ per year. http://www.ringlead.com/dirty-data-costs-economy-3-trillion/.
2. Firms full of dirty data. http://www.itpro.co.uk/ 609057/firms-full-of-dirty-data.
3. M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. *TPLP*, 2003.
4. L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT*, 2011.
5. G. Beskales, G. Das, A. K. Elmagarmid, I. F. Ilyas, F. Naumann, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, and N. Tang. The data analytics group at the qatar computing research institute. *SIGMOD Record*, 41(4):33–38, 2012.

[2] http://spark.apache.org

6. G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 2010.
7. G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David. Modeling and querying possible repairs in duplicate detection. In *VLDB*, 2009.
8. P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.
9. L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In *VLDB*, 2007.
10. J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 2005.
11. X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13), 2013.
12. X. Chu, P. Papotti, and I. Ilyas. Holistic data cleaning: Put violations into context. In *ICDE*, 2013.
13. G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.
14. 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*, 2013.
15. A. Ebaid, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. Nadeef: A generalized data cleaning system. *PVLDB*, 2013.
16. A. Elmagarmid, I. F. Ilyas, M. Ouzzani, J. Quiane-Ruiz, N. Tang, and S. Yin. NADEEF/ER: Generic and interactive entity resolution. In *SIGMOD*, 2014.
17. W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.
18. W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 2008.
19. W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011.
20. W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.
21. W. Fan, F. Geerts, and J. Wijsen. Determining the currency of data. In *PODS*, 2011.
22. 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.
23. W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.
24. W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 2012.
25. W. Fan, J. Li, N. Tang, and W. Yu. Incremental detection of inconsistencies in distributed data. In *ICDE*, pages 318–329, 2012.
26. I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 1976.
27. S. Kolahi and L. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.
28. C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.
29. V. Raman and J. M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *VLDB*, 2001.
30. J. Wang and N. Tang. Towards dependable data with fixing rules. In *SIGMOD*, 2014.
31. M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.