NADEEF: A Commodity Data Cleaning System

Michele Dallachiesa[†][*] Amr Ebaid[‡][*] Ahmed Eldawy[§][*]
Ahmed Elmagarmid[♯] Ihab F. Ilyas[♯] Mourad Ouzzani[♯] Nan Tang[♯]

[†]University of Trento [‡]Purdue University [§]University of Minnesota [♯]QCRI
dallachiesa@disi.unitn.it, aebaid@cs.purdue.edu, eldawy@cs.umn.edu
{aelmagarmid, ikaldas, mouzzani, ntang}@qf.org.qa

ABSTRACT

Despite the increasing importance of data quality and the rich theoretical and practical contributions in all aspects of data cleaning, there is no single 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. In short, there is no commodity platform similar to general purpose DBMSs that can be easily customized and deployed to solve application-specific data quality problems. In this paper, we present NADEEF, 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. We show that the programming interface can be used to express many types of data quality rules beyond the well known CFDs (FDs), MDs and ETL rules. Treating user implemented interfaces as black-boxes, the *core* provides algorithms to detect errors and to clean data. The *core* is designed in a way to allow cleaning algorithms to cope with multiple rules *holistically*, *i.e.,* detecting and repairing data errors without differentiating between various types of rules. We showcase two implementations for core repairing algorithms. These two implementations demonstrate the *extensibility* of our core, which can also be replaced by other user-provided algorithms. Using real-life data, we experimentally verify the generality, extensibility, and effectiveness of our system.

Categories and Subject Descriptors

H.2 [**Database Management**]: General—*integrity*

General Terms

System, Algorithms, Experimentation

[*]Work done while interning at QCRI.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

Keywords

ETL, conditional functional dependency, matching dependency, data cleaning

1. INTRODUCTION

Data has become an important asset in today's economy. Extracting values from large amounts of data to provide services and to guide decision making processes has become a central task in all data management stacks. The quality of data becomes one of the differentiating factors among businesses and the first line of defense in producing value from raw input data. Ensuring the quality of the data with respect to business and integrity constraints has become more important than ever.

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 heterogeneous 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. [14] 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).

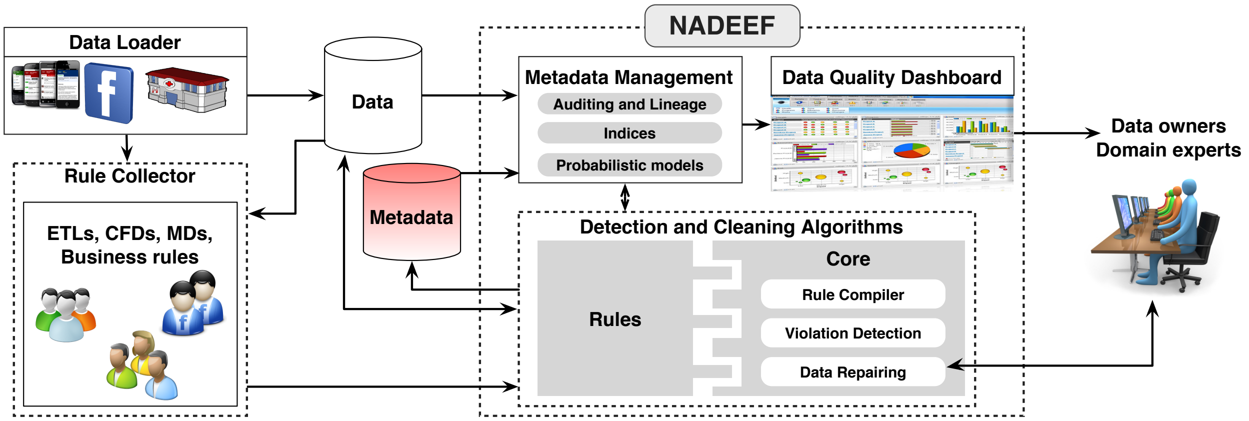


Figure 1: Architecture of NADEEF

Deployment and extensibility: Although many algorithms and techniques have been proposed for data cleaning [5, 14, 29], 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.,* [15, 26, 29]). 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.

We introduce NADEEF, 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.

We show NADEEF's architecture in Fig. 1. In a nutshell, NADEEF first collects data and rules defined by the users. The rule compiler then compiles these *heterogeneous* rules into homogeneous constructs. Next, the violation detection module finds what data is erroneous and possible ways to repair them, based on user provided rules. After identifying errors, the data repairing module handles the *interdependency* of these rules by treating them holistically. Since data cleaning is usually an iterative process and new violations might be introduced when updating data values during repairing, NADEEF adopts a simple strategy to ensure that the process terminates. NADEEF also manages metadata related to its different modules. These metadata can be used to allow domain experts and users to actively interact with the system. We are working on augmenting NADEEF with a data quality dashboard that would exploit these metadata and provide information such as error summarization and error distribution; details on this component are left for future work.

Contributions. We make several notable contributions.

(1) We describe the first end-to-end commodity data cleaning system (Section 2).

(2) We propose a novel programming interface (Section 3) that supports the *generality* of NADEEF by providing users with ways to specify the semantics, both static and dynamic, of multiple types of data quality rules.

(3) We discuss basic error detection methods (Section 4). We also describe *partitioning* and *compression* as two techniques to improve violation detection. The former is to reduce the number of pairwise comparisons when computing violations, while the latter is to reduce the number of comparisons and the size of violations.

(4) To demonstrate the *extensibility* of our system, we present two core implementations for data cleaning algorithms (Section 5). The first one, which is designed to achieve higher accuracy, converts all violations and possible data changes to a variable-weighted conjunctive normal form (CNF). This CNF is then fed to a weighted MAX-SAT solver that computes repairs while minimizing their overall cost. The second one, which is designed to be more efficient, leverages the idea of equivalence classes [5]. It merges violating data into multiple equivalence classes and assigns a unique value to each equivalence class.

(5) We conduct extensive experiments to verify the generality, extensibility, and effectiveness of our system using real-life datasets (Section 6).

2. A MOTIVATING SCENARIO AND SYSTEM ARCHITECTURE

We present a detailed scenario to show the many features of NADEEF, followed by a discussion about its architecture.

2.1 A Motivating Scenario

Consider two databases D_1 and D_2 from a European bank: D_1 maintains customer information collected when credit cards are issued and D_2 records credit card transactions. The databases are specified by the following schemas:

bank (FN, LN, St, city, CC, country, tel, gd),
tran (FN, LN, str, city, CC, country, phn, when, where).

Here, a bank record specifies a credit card holder identified by first name (FN), last name (LN), street (St), city, country code (CC), country, phone number (tel) and gender (gd). A tran tuple is a record of a purchase paid by a credit card at time when and place where, by a customer identified by first name (FN), last name (LN), street (str), city, country code (CC), country and phone (phn). Example instances of bank and tran tables are shown in Figs. 2(a) and 2(b), respectively.

	FN	LN	St	city	CC	country	tel	gd
t_1:	David	Jordan	12 Holywell Street	Oxford	44	UK	66700543	Male
t_2:	Paul	Simon	5 Ratcliffe Terrace	Oxford	44	UK	44944631	Male

(a) D_1: An instance of schema bank

	FN	LN	str	city	CC	country	phn	when	where
r_1:	David	Jordan	12 Holywell Street	Oxford	44	UK	66700543	1pm 6/05/2012	Netherlands
r_2:	Paul	Simon	5 Ratcliffe Terrace	Oxford	44	UK	44944631	11am 2/12/2011	Netherlands
r_3:	David	Jordan	12 Holywell Street	Oxford	44	Netherlands	66700541	6am 6/05/2012	US
r_4:	Peter	Austin	7 Market Street	Amsterdam	31	UK	55384922	9am 6/02/2012	Netherlands

(b) Database D_2: An instance of schema tran

Figure 2: Example databases

Based on the application business logic, users may impose the following rules:

φ_1: *(on table* tran*) if a customer's* CC *is 31, but his/her* country *is neither Netherlands nor Holland, update the country to Netherlands;*

φ_2: *(on tables* bank *and* tran*) if the same person from different tables has different phones, the phone number from table* bank *is more reliable;*

φ_3: *(on table* tran*) a country code (*CC*) uniquely determines a* country*;*

φ_4: *(on table* tran*) if two purchases of the same person happened in the Netherlands and the US (East Coast) within 1 hour (assuming 6 hours' time difference between these two countries), these two purchases are either a fraud or were erroneously recorded.*

Figure 3 shows how these rules are represented in our system via a unified programming interface. In a nutshell, the users specify a rule's static semantics (*what* is wrong) over either one tuple or two tuples via a vio() function, where an error is represented by a set of attribute values. Moreover, the users can also, if possible, specify its dynamic semantics (*i.e., how* to repair) by providing different alternatives to repair the data via a fix() function.

Rule1 (for φ_1) is a transformation rule on a single tuple on table tran. It states via a vio() function that if a tuple's CC is 31, but its country is neither Netherlands nor Holland, then it is problematic. It also states via a fix() function that the violating tuple's country should be updated to Netherlands.

Rule2 (for φ_2) is defined on two tables bank and tran. When it identifies that the same person from different tables has different phones (*i.e.,* the violation is not empty via a vio() function), it updates the phn value in tran to the tel value from bank (in fix()). Actually, the user indicates that the values from bank are more reliable. Here, the $\approx$ could be any (domain-specific) similarity function that a user defines.

Rule3 (for φ_3) specifies how to identify the violations via a vio() function on two tuples from table tran. It also defines via a fix() function two options to resolve a violation V: either update $V.s_2$[country] to $V.s_1$[country], or update $V.s_1$[country] to $V.s_2$[country].

Rule4 (for φ_4) only specifies how to detect a violation via a vio() function. No fix() function is given since the users do not know how to resolve this violation.

By allowing the expression of multiple types of rules (*e.g.,* eCFD [6] for the vio() of φ_1 and ETL for the fix() of φ_1, an MD φ_2, an FD φ_3 augmented with a fix() that changes values from the right hand side of the rule, and a custom rule φ_4) through our unified programming interface, NADEEF provides a simple way to determine what is wrong via vio() functions, and more importantly, it also captures how to repair it via a fix() function.

```
Class Rule1 {                                                      /* for $\varphi_1$ */
  set⟨cell⟩ vio(Tuple $s_1$) {                                  /*$s_1$ in table tran */
    if ($s_1$[CC]=31 $\wedge$ ($s_1$[country] $\neq$Netherlands $\vee$ $s_1$[country] $\neq$Holland))
      return { $s_1$[CC, country]; }
    return $\emptyset$;
  }
  set⟨Expression⟩ fix (set⟨cell⟩ V) {
    return { $V.s$[country] $\leftarrow$ Netherlands; }
  } } /* end of class definition */
```

```
Class Rule2 {                                                      /* for $\varphi_2$ */
  set⟨cell⟩ vio (Tuple $s_1$, Tuple $s_2$) {             /*$s_1$ in bank, $s_2$ in tran */
    if ($s_1$[LN, St, city]=$s_2$[LN, str, city]$\wedge s_1$[FN] $\approx$ $s_2$[FN]$\wedge s_1$[tel]$\neq s_2$[phn])
      return { $s_1$[FN, LN, St, city, tel], $s_2$[FN, LN, str, city, phn]; }
    return $\emptyset$;
  }
  set⟨Expression⟩ fix (set⟨cell⟩ V) {
    return { $V.s_2$[phn] $\leftarrow$ $V.s_1$[tel]; }
  } } /* end of class definition */
```

```
Class Rule3 {                                                      /* for $\varphi_3$ */
  set⟨cell⟩ vio (Tuple $s_1$, Tuple $t_2$) {             /* $s_1$, $s_2$ in table tran */
    if ($s_1$[CC] = $s_2$[CC] $\wedge$ $s_1$[country] $\neq$ $s_2$[country])
      return { $s_1$[CC, country], $s_2$[CC, country]; }
    return $\emptyset$;
  }
  set⟨Expression⟩ fix (set⟨cell⟩ V) {
    set⟨Expression⟩ fixes;
    fixes.insert($V.s_1$[country] $\leftarrow V.s_2$[country]);
    fixes.insert($V.s_2$[country] $\leftarrow V.s_1$[country]);
    return fixes;
  } } /* end of class definition */
```

```
Class Rule4 {                                                      /* for $\varphi_4$ */
  set⟨cell⟩ vio (Tuple $s_1$, Tuple $s_2$) {             /*$s_1$, $s_2$ in table tran */
    if ($s_1$[LN, city, CC, tel] = $s_2$[LN, city, CC, tel]
      $\wedge s_1$[where] = $Netherlands$ $\wedge$ $s_2$[where] = $US$ $\wedge$ $s_1$[FN] $\approx$ $s_2$[FN]
      $\wedge$($s_1$[when] $-$ $s_2$[when] $\geq 5$) $\wedge$ ($s_1$[when] $-$ $s_2$[when] $\leq 7$)
      return { $s_1$[FN, LN, city, CC, tel, when, where],
        $s_2$[FN, LN, city, CC, tel, when, where]; }
    return $\emptyset$;
  } } /* end of class definition */
```

Figure 3: Sample rules

Next, we show how the data in Fig. 2 can be repaired by the rules in Fig. 3 in a sequence of interdependent operations, as illustrated below:

(a) *Rule1* updates r_4[country] from UK to Netherlands.

(b) *Rule2* identifies that t_1 in D_1 and r_3 in D_2 are the same person, but with different phones. It updates r_3[phn] by taking 66700543 from t_1[tel], since bank is more reliable.

(c) *Rule3* detects that r_3 violates r_1 and r_2, *i.e.,* they have the same CC value but carry different country values. To resolve these two violations, one can either change r_3[country] from Netherlands to UK, or change both r_1[country] and r_2[country] from UK to Netherlands. A typical cleaning algorithm will choose the one with the minimum number of changes, *i.e.,* changing r_3[country] from Netherlands to UK.

(d) *Rule4* detects that the two purchases from the same person, r_1 and r_3, are problematic. Note that this rule only reports the violation, without repairing it.

Observe that only after changing r_3[phn] to 66700543 (the step (b) above), the violation in step (d) can be detected by a customized rule. We can see that (1) when taken together, different data quality rules help each other, and (2) to make practical use of their interaction, repairing operations for various types of data quality rules should be interleaved.

From this scenario, we can see that a commodity data cleaning system has to provide (a) a unified format for the users to specify *both* the static semantics *and* dynamic semantics of various types of rules, and (b) common data structures to represent data errors, which provide a natural way to design algorithms that holistically repair data.

2.2 Architecture Overview

Figure 1 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.

Inside the core of NADEEF, we also implement an extensible Updater that decides which computed data changes will be finally committed. The Updater is needed since data cleaning is an expensive, (mostly) nondeterministic, and iterative process. When applied to the database, updates computed by repairing algorithms may trigger new violations and the cleaning process may not terminate. The Updater provides a termination test, ensuring that the entire data cleaning process terminates. We adopt a simple termination test in each iteration of the data repair process that is similar to data cleaning techniques proposed in [8,20]; the Updater applies the computed updates only if an attribute value is not changed more than x times (x is a parameter set to 2 by default). Otherwise, the Updater changes the attribute value to a special null value that eliminates any potential violations on this attribute value in the future.

Metadata management and data quality dashboard. Building a commodity data cleaning system requires collecting and handling several types of metadata to help in understanding and improving the cleaning process as well as supporting many features of NADEEF. The role of a metadata management module is to keep full lineage information about data changes, the order of changes, as well as maintaining indices to support efficient metadata operations. The data quality dashboard helps the users to understand the health of the database through progress indicators, data quality health information, as well as summarized, sampled or ranked data errors. It also facilitates the solicitation of users' feedback for data repairs. Details about this module are left for future work.

3. FUNDAMENTALS

In this section, we present the proposed programming interface for rule specification as well as some of the notations and concepts needed in NADEEF.

We consider a database $\mathcal{D} = \{D_1, \cdots, D_m\}$, where each D_j ($j \in [1, m]$) is an *instance* whose relation schema is R_j as $R_j = \{A_{j_1}, \cdots, A_{j_n}\}$. We use the term *cell* to denote a combination of a tuple and an attribute of a table, *i.e.,* $D.s[A]$. For simplicity, we write a cell as $s[A]$, when D is clear from the context.

For example, in Fig. 2, D_1 is an instance of relation bank, and t_1[FN] is a cell in instance D_1 whose value is David.

Data quality rules. The *programming interface* class *Rule* for defining the semantics of data errors and possible ways to fix them is as follows.

```
class Rule {
    set⟨cell⟩ vio (Tuple s1) { return ∅ };
    set⟨cell⟩ vio (Tuple s1, Tuple s2) { return ∅ };
    set⟨Expression⟩ fix (set⟨cell⟩) { return ∅ };
}    /* end of class definition */
```

In the above class we define three functions:

(1) vio(s) takes a single tuple s as input, and returns a set of problematic cells, *e.g., Rule1* in Fig. 3. By default, it returns an empty set.

(2) vio(s_1, s_2) takes two tuples s_1, s_2 as input, and returns a set of problematic cells, *e.g., Rule2*, *Rule3* and *Rule4* in Fig. 3. By default, it returns an empty set. Note that s_1, s_2 can come from the same relation or two different relations.

(3) fix (*set*⟨*cell*⟩) takes a nonempty set of problematic cells as input, and returns a set of suggested expressions to repair these data errors.

We refer to a class that inherits from *Rule* and implements at least one of the error detection functions vio() as a *data quality rule*, in order to express its static semantics (*what* is wrong). The function fix() is to reflect its dynamic semantics (*how* to repair errors). The presence of function fix() is optional, and its absence indicates that the users are not clear about its dynamic semantics, *e.g., Rule4* in Fig. 3.

We can get φ_i ($i \in [1, 4]$) in Section 2 by instantiating *Rulei* in Fig. 3. In the rest of the paper, we shall use φ_i and *Rulei* interchangeably.

In contrast to traditional ways of defining data quality rules by strictly following some declarative logical formalism, the class *Rule* provides a *unified* and *generic* object-oriented programming interface. Such interface is expressive enough to allow users to easily capture both static and dynamic semantics of a large spectrum of data quality rules, as well as complex (*e.g.,* probabilistic or knowledge-based) processes.

Violations and candidate fixes. Violations specify what is wrong, while candidate fixes capture how to repair errors.

Violation. A *violation* is a nonempty set V of *cells* that are returned by the function vio(s) or vio(s_1, s_2) of a data quality rule φ, referred to as a violation of φ.

Intuitively, in a violation, at least one of the cells is erroneous and should be modified. For example, the two tuples r_1 and r_3 shown in Fig. 2 violate *Rule3*, since they have the same CC value but carry different $\mathsf{country}$ values. The corresponding violation consists of four cells $\{r_1[\mathsf{CC}], r_1[\mathsf{country}], r_3[\mathsf{CC}], r_3[\mathsf{country}]\}$.

For a database $\mathcal{D}$ and a data quality rule φ, we denote by $\mathsf{vio}(\mathcal{D}, \varphi)$ the set of all nonempty results returned by $\varphi.\mathsf{vio}(s)$ (*i.e.,* a single tuple) and $\varphi.\mathsf{vio}(s_1, s_2)$ (*i.e.,* a pair of tuples), where s is a single tuple in $\mathcal{D}$ and (s_1, s_2) is a pair of distinct tuples in $\mathcal{D}$, respectively. For a database $\mathcal{D}$ and a set Σ of rules, we denote by $\mathsf{vio}(\mathcal{D}, \Sigma) = \bigcup_{\varphi\in\Sigma} \mathsf{vio}(\mathcal{D}, \varphi)$ the set of all violations for data $\mathcal{D}$ and rules Σ.

In practice, many types of violations are usually defined on either a single tuple (*e.g.,* constant CFDs and many ETL rules), two tuples (*e.g.,* variable CFDs, MDs and deduplication rules), or a set of tuples (*e.g.,* aggregation constraints [21]). Supporting aggregation functions and other data quality rules defined on a subset of the data triggers a different class of challenges which are beyond the scope of this paper. This is *especially* true for the efficiency of the violation detection process. In this paper, we focus on the first two classes of violations, *i.e.,* violations defined on one tuple or two tuples. These two classes already cover a very large spectrum of data quality rules found in practice, thus not diminishing the expressiveness of NADEEF.

Candidate fix. A *candidate fix* F is a conjunction of expressions of the form "$c \leftarrow x$", where c is a cell, and x is either a constant value or another cell.

Intuitively, a candidate fix F is a set of expressions on a violation V, such that to resolve violation V, the modifications suggested by the expressions of F must be taken together. That is, to resolve a violation, more than one cell may have to be changed. In our example, each possible fix contains a single expression. For instance, consider *Rule3* in Fig. 3, each violation V has two candidate fixes, either assigning $V.s_2[\mathsf{country}]$ to $V.s_1[\mathsf{country}]$, or $V.s_1[\mathsf{country}]$ to $V.s_2[\mathsf{country}]$. Either way can resolve violation V.

Cost functions. As a database can be repaired in multiple ways, an immediate question is which repair to choose? Similarly to what most data cleaning methods use to make their decision, we adopt minimality, *i.e.,* compute an instance that repairs a database while incurring the least cost in terms of fixing operations. Let $\mathsf{cost}(c, v_1, v_2)$ be the cost of changing the cell c from value v_1 to v_2, and $\mathsf{cost}(\mathcal{D}, \mathcal{D}_r)$ is defined as the sum of $\mathsf{cost}(c, v_1, v_2)$ for each cell whose value is modified from $\mathcal{D}$ to a modified database $\mathcal{D}_r$. Since the users can plugin their own repairing algorithms, they can also replace the cost functions by their own.

Consistent database. Consider an instance $\mathcal{D}$ of $\mathcal{R}$, and a data quality rule φ, we say that $\mathcal{D}$ *satisfies* φ, denoted by $\mathcal{D} \models \varphi$, if (i) $\mathsf{vio}(s)$ returns an empty set for each tuple t in $\mathcal{D}$; and (ii) $\mathsf{vio}(s_1, s_2)$ returns an empty set for all pairs of distinct tuples s_1, s_2 in $\mathcal{D}$. We say that $\mathcal{D}$ *satisfies* a set Σ of data quality rules, denoted by $\mathcal{D} \models \Sigma$, if $\mathcal{D}$ *satisfies* each φ in Σ. We say that $\mathcal{D}$ is *consistent w.r.t.* Σ if $\mathcal{D} \models \Sigma$.

Fixed database. For an instance $\mathcal{D}$ of $\mathcal{R}$ and a data quality rule φ, we say that $\mathcal{D}$ is *fixed w.r.t.* φ, if for each violation V of $\mathcal{D}$ *w.r.t.* φ, $\mathsf{fix}(V)$ returns an empty set. We say that $\mathcal{D}$ is *fixed w.r.t.* a set Σ of data quality rules, if $\mathcal{D}$ is *fixed w.r.t.* each φ in Σ.

Consider our example in Section 2. After steps (a)-(d), we obtain a modified database $\mathcal{D}'$. While one violation still remains, *i.e.,* tuples (r_1, r_3) *w.r.t.* rule φ_4, it has no candidate fixes. Hence, the database $\mathcal{D}'$ is said to be *fixed*, although it is inconsistent, *e.g.,* it does not *satisfy* the rule φ_4 in Σ.

Unresolved violations. Note that, in our system, $\mathcal{D}_r$ must be *fixed* but may contain unresolved violations, *i.e.,* $\mathcal{D}_r$ may be inconsistent *w.r.t.* Σ. As opposed to traditional approaches that compute a fix $\mathcal{D}_r'$ that must be consistent, *i.e.,* $\mathsf{vio}(\mathcal{D}_r', \varphi)$ is also empty for each φ in Σ, the problem we study only repairs dirty data for which candidate fixes are known. This is based on the fact that in practice and for some rules, there may not exist sufficient knowledge on how to resolve the corresponding violations; neither the users know *a priori* nor the data cleaning system could guess. Heuristically resolving such violations may introduce more errors, triggering a disastrous domino effect.

Further data quality aspects. There are several fundamental problems associated with quality rules. The *consistency problem* is to determine, given Σ and schemas $\mathcal{R}$, whether there exists a nonempty instance $\mathcal{D}$ (each table in $\mathcal{D}$ is nonempty) of $\mathcal{R}$ such that $\mathcal{D} \models \Sigma$. The *implication problem* is to decide, given Σ and another data quality rule ψ, whether Σ implies ψ. In simpler terms, the consistency problem is to decide whether the data quality rules are dirty themselves, and the implication problem is to decide whether a data quality rule is redundant. When treating the rules as black-boxes, it is difficult, if not impossible, to check whether they are internally consistent.

It has been verified in [14] that given CFDs only, or for CFDs and MDs taken together, the consistency (resp. implication) problem is NP-complete (resp. coNP-complete). However, when either the database schema is predefined or no attributes involved in the CFDs have a finite domain, the consistency check for CFDs is PTIME [14], which actually covers many practical applications. For the previous case or when the number of rules (*e.g.,* CFDs) is small, several algorithms have been proposed to check their consistency [12]. It is worth mentioning that we have implemented default classes for CFDs (FDs) and MDs; the consistency of such particular rules can be checked by these algorithms. In the following of this work, we assume collections of Σ and $\mathcal{D}$ that are consistent.

4. VIOLATION DETECTION

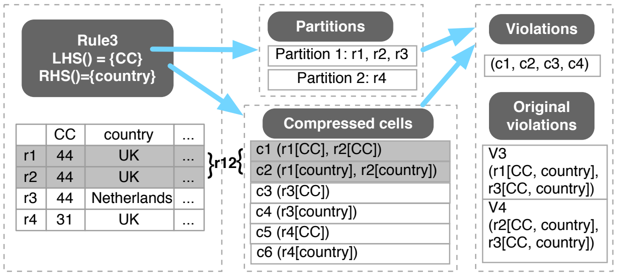
We describe a basic approach to compute violations as well as optimization techniques, namely partitioning and compression, that are possible under some restricted settings.

4.1 Finding Violations and Candidate Fixes

Given a database $\mathcal{D}$ and a set Σ of data quality rules, the method for violation detection, referred to as GetVio, returns a set $\mathcal{V}$ of violations and a set $\mathcal{F}$ of candidate fixes, where $\mathcal{V}$ (resp. $\mathcal{F}$) is the union of the violations (resp. candidate fixes) of $\mathcal{V}_\varphi$ (resp. $\mathcal{F}_\varphi$) for each φ in Σ. A straightforward way to compute $\mathcal{V}$ and $\mathcal{F}$ for each φ, is to invoke the functions $\mathsf{vio}(s)$ and $\mathsf{vio}(s_1, s_2)$ for each single tuple and each pair of tuples on which φ is defined, respectively. The $\mathsf{fix}()$ function will be invoked for each violation returned by $\mathsf{vio}(s)$ or $\mathsf{vio}(s_1, s_2)$. Since the quadratic time pairwise comparison for any two tuples is inherently expensive, we will discuss in Section 4.2 *partitioning* and *compression* as two techniques to improve its performance.

Rules	Violations	Candidate fixes
φ_1	V_1: $\{r_4[\mathsf{CC, country}]\}$	F_1: $r_4[\mathsf{country}]$←Netherlands
φ_2	V_2:$\{t_1[\mathsf{FN, LN, St, city, tel}]$, $r_3[\mathsf{FN, LN, str, city, phn}]\}$	F_2: $r_3[\mathsf{phn}] \leftarrow t_1[\mathsf{tel}]$
φ_3	V_3: $\{r_1[\mathsf{CC, country}]$, $r_3[\mathsf{CC, country}]\}$	F_3: $r_1[\mathsf{country}] \leftarrow r_3[\mathsf{country}]$ F_4: $r_3[\mathsf{country}] \leftarrow r_1[\mathsf{country}]$
φ_3	V_4: $\{r_2[\mathsf{CC, country}]$, $r_3[\mathsf{CC, country}]\}$	F_5: $r_2[\mathsf{country}] \leftarrow r_3[\mathsf{country}]$ F_6: $r_3[\mathsf{country}] \leftarrow r_2[\mathsf{country}]$

Figure 4: Sample violations and candidate fixes



	CC	country	...
r1	44	UK	...
r2	44	UK	...
r3	44	Netherlands	...
r4	31	UK	...

Figure 5: Partitioning and compression for a rule

Example 1: Consider steps (a-c) of the example in Section 2. There are 4 violations and 6 candidate fixes, as shown in Fig. 4. By applying φ_1 (resp. φ_2), we find a violation V_1 (resp. V_2) with two cells (resp. ten cells) involved from tuple r_4 (resp. tuples t_1 and r_3). The violation V_1 (resp. V_2) has a candidate fix F_1 (resp. F_2) that assigns Netherlands to $r_4[\mathsf{country}]$ (resp. $t_1[\mathsf{tel}]$ to $r_3[\mathsf{phn}]$). Moreover, attribute values from tuples r_1 and r_3 (resp. r_2 and r_3) violate φ_3, *i.e.,* violation V_3 (resp. V_4), leading to two candidate fixes F_3 and F_4 (resp. F_5 and F_6). Note that no violations of φ_4 exist before resolving the violations of φ_2. □

4.2 Optimizations for Violation Detection

In the general case, NADEEF has to trade in performance for generality and extensibility. For instance, violation detection upon all pairs of tuples using the $\mathsf{vio}()$ function is inherently expensive. However, it should be possible to do better in more restricted settings. One typical optimization is to divide large sets of input tuples into groups, referred to as *partitions*, such that violations are detected on each partition, provided that some information is known about the rules. For example, if we are resolving country code (CC) violations (*e.g., Rule3* in Fig. 3), we may be able to divide them using the CC attribute. Thus, violations of tuples with the same CC but different $\mathsf{country}$ values need to be only detected inside each partition.

Another way to reduce the number of pairwise comparisons is to merge tuples. Specifically, two tuples t_1 and t_2 can be merged for a rule φ, if (1) (t_1, t_2) do not violate φ; and (2) for any t_3, $(t_1, t_3) \not\models \varphi$ iff $(t_2, t_3) \not\models \varphi$. Intuitively, the second condition requires that t_1 and t_2 can be merged if for any tuple t_3, they either both violate φ with t_3, or neither violates φ with t_3. This method is referred to as *compression* for violations.

In order to apply the above two techniques, we need users to provide some knowledge about their rules and how they use the attributes. Concretely, the system needs to know (a) the set of attributes that are used to indicate why two tuples should be compared; and (b) the set of attributes that need to be modified when there is a violation. We provide two functions LHS(φ) and RHS(φ) that the users need to implement to return the set of attributes for the above (a) and (b), respectively. For example, consider *Rule3* (φ_3) in Fig. 4, we have LHS(φ_3) = $\{\mathsf{CC}\}$ and RHS(φ_3) = $\{\mathsf{country}\}$.

Next, we first discuss the case for one data quality rule, and then extend to multiple quality rules. For simplicity, in the following, we focus our discussion on equality comparison. For various similarity comparisons, several blocking-based techniques are already in place and can be leveraged within NADEEF (see *e.g.,* [7, 25]).

Single data quality rule. For a single data quality rule φ, *partitioning* groups all tuples whose LHS(φ) attribute values are the same. Using a hash table, the partitioning can be performed in linear time, assuming that a hash table requires a constant cost per operation.

For *compression*, two tuples t_1 and t_2 are merged into t_{12} *w.r.t.* φ if $t_1[\mathrm{LHS}(\varphi) \cup \mathrm{RHS}(\varphi)] = t_2[\mathrm{LHS}(\varphi) \cup \mathrm{RHS}(\varphi)]$. When they are merged, their cell values related to attributes LHS(φ) ∪ RHS(φ) are the same and will be compressed as $t_{12}[\mathrm{LHS}(\varphi) \cup \mathrm{RHS}(\varphi)] = t_1[\mathrm{LHS}(\varphi) \cup \mathrm{RHS}(\varphi)]$. We use the term *super cell* for a compressed cell. A super cell c is a cell with a set of identifers of original cells, referred to as the extension of cell c, denoted by $\mathsf{ext}(c)$. For the attributes that are irrelevant to φ, their cell values in t_{12} are set to null, and their extensions are empty.

It deserves to note that when similarity comparisons are considered, two cells whose values are similar cannot be compressed, if the similarity function is not transitive.

Example 2: Consider the database in Fig. 2 and the rule *Rule3* (φ_3) in Fig. 3. Figure 5 illustrates how the partitioning and compression techniques work. Here, LHS(φ_3) = $\{\mathsf{CC}\}$ and RHS(φ_3) = $\{\mathsf{country}\}$.

Partitioning table tran leads to two partitions: *Partition*1 with three tuples r_1–r_3 since their CC values are the same *i.e.,* 44, and *Partition*2 with one tuple r_4.

For compression, since $r_1[\mathsf{CC, country}] = r_2[\mathsf{CC, country}]$, the two tuples r_1 and r_2 (shaded tuples) will be merged into r_{12}. Their cells $r_1[\mathsf{CC}]$ and $r_2[\mathsf{CC}]$ (resp. $r_1[\mathsf{country}]$ and $r_2[\mathsf{country}]$) are merged into a super cell c_1 (resp. c_2) in r_{12} whose extension is $\mathsf{ext}(c_1) = \{r_1[\mathsf{CC}], r_2[\mathsf{CC}]\}$ (resp. $\mathsf{ext}(c_2) = \{r_1[\mathsf{country}], r_2[\mathsf{country}]\}$). The other cells of r_{12}, which are irrelevant to φ_3, have empty extensions and are omitted here. The other tuples (r_3 and r_4) are not compressed, and their cells c_3–c_6 are shown in Fig. 5.

From *Partition*1 and after compression (tuples r_{12} and r_3), one violation with four cells, *i.e.,* $\{c_1, c_2, c_3, c_4\}$, is detected. Note that this violation was originally represented by two violations as given in Fig. 5. There are no violations from *Partition*2. □

```
Algorithm ParComVio
Input:  a database $\mathcal{D}$, a set $\Sigma$ of data quality rules.
Output: the set $\mathcal{V}$ of all violations.
$\mathcal{V} = \emptyset$;
for each data quality rule $\varphi$ in $\Sigma$ do
    $\mathcal{P} = \mathsf{Partition}(\mathcal{D}, \varphi)$;
    $\mathcal{C} = \mathsf{Compress}(\mathcal{P}, \varphi)$;
    $\mathcal{V} = \mathcal{V} \cup \mathsf{DetectViolation}(\mathcal{P}, \mathcal{C}, \varphi)$;
while $\exists c_a, c_b \in \mathcal{V}$ such that $\mathsf{ext}(c_a) \cap \mathsf{ext}(c_b) \neq \emptyset$
        and $\mathsf{ext}(c_a) \neq \mathsf{ext}(c_b)$ do
    $\mathcal{V} = \mathsf{ExpandViolation}\ (c_a, \varphi_a, c_b, \varphi_b)$;
return $\mathcal{V}$;
```

Figure 6: Algorithm for partitioning and compression for multiple quality rules

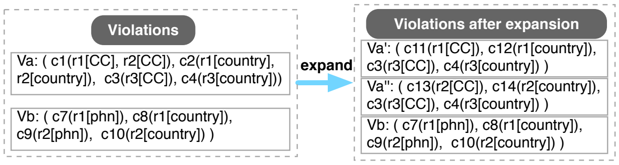

Violations
Va: (c1(r1[CC], r2[CC]), c2(r1[country], r2[country]), c3(r3[CC]), c4(r3[country]))
Vb: (c7(r1[phn]), c8(r1[country]), c9(r2[phn]), c10(r2[country]))
expand
Violations after expansion
Va': (c11(r1[CC]), c12(r1[country]), c3(r3[CC]), c4(r3[country]))
Va'': (c13(r2[CC]), c14(r2[country]), c3(r3[CC]), c4(r3[country]))
Vb: (c7(r1[phn]), c8(r1[country]), c9(r2[phn]), c10(r2[country]))

Figure 7: Expanding violations

Multiple data quality rules. The algorithm for detecting violations for multiple rules, referred to as ParComVio, is given in Fig. 6. Given a database $\mathcal{D}$ and a set Σ of data quality rules as input, the algorithm computes and returns a set $\mathcal{V}$ of all violations. It first creates partitions for each data quality rule (lines 2-3). It then compresses tuples for each partition (line 4). The violations for each rule will be computed within its partitions (line 5; see Example 2). After computing all violations, the algorithm needs to expand violations that have intersections across different rules (see Example 3 below) (lines 6-7). The violations are then returned (line 8).

Notably, a super cell is originally related to one rule (line 4), and all the cells in one super cell have the same value. Hence, extensions of any two super cells should be either the same, or disjoint, such that the value assignment to different super cells, for any cleaning algorithm, should be irrelevant. Otherwise, we need to expand these super cells (lines 6-7). We illustrate how the expansion works with the example below.

Example 3: For table tran (Fig. 2), assume that there is another rule φ_5 that states country uniquely determines phn. For simplicity, we only show one violation of φ_5, which is V_b as shown in Fig. 7. The violation V_a in Fig. 7 is the one derived in Fig. 5. Note that the two cells, c_2 in V_a and c_8 in V_b are not the same but overlap on r_1[country]. Hence, violation V_a needs to be expanded to V'_a and V''_a, as depicted in Fig. 7. □

One can derive original tuples from a compressed tuple, by uncompressing their super cells. Similarly, one can derive original violations from the violations given by super cells.

Complexity. The first loop (lines 2-5) runs in $O(|\mathcal{D}|^2|\Sigma|)$ time, by employing a hash table for partitioning, which is the worst case to detect violations pairwise without any compression. Similarly, the second loop runs in $O(|\mathcal{D}|^2|\Sigma|)$ time. In total, algorithm ParComVio runs in $O(|\mathcal{D}|^2|\Sigma|)$ time, where $|\Sigma|$ is typically small in practice.

Although the worst case complexity is the same as a brute force method, Fig. 8 shows the benefits of partitioning and compression using a 10K tuples HOSP data. Without any optimization, it requires 1 billion pairwise comparisons and produces 100K violations. With partitioning and compression, it requires only 4.4K pairwise comparisons and produces 4.4K violations.

5. DATA REPAIRING

In this section, we describe two algorithms for the data repairing module (implemented in NADEEF), referred to as GetFix. The first algorithm, which is designed to achieve higher accuracy, encodes a data cleaning problem to a variable-weighted conjunctive normal form (CNF) such that existing MAX-SAT solvers can be invoked to compute repairs with minimum cost. The second algorithm, which is designed to be more efficient, heuristically computes repairs by extending the idea of equivalence classes employed by many existing data cleaning algorithms. These two implementations illustrate the extensibility of our system using different repairing approaches. Better still, users can override these core classes with their own implementations.

Method	Comparisons	Violations
Brute force	1,199,880,000	130,038
Partitioning	6,797,429	130,038
Compression	52,512,009	4,434
Part. + Comp.	4,434	4,434

Figure 8: Benefits of partitioning and compression

5.1 A Variable-Weighted MAX-SAT Solver based Algorithm

Given a set $\mathcal{V}$ of violations and a set $\mathcal{F}$ of candidate fixes, the algorithm computes a set $\mathcal{F}'$ of fixes with the target that (1) when $\mathcal{F}'$ is applied to a database $\mathcal{D}$, the updated database $\mathcal{D}'$ is *fixed*, and (2) the overhead $\mathsf{cost}(\mathcal{D}, \mathcal{D}')$ of changing $\mathcal{D}$ to $\mathcal{D}'$ is minimum.

We propose to achieve the above target by converting our problem to a variable-weighted MAX-SAT problem, a well studied NP-hard problem. Given a CNF where each variable has an associated weight, this problem is to decide a set of Boolean assignments of variables such that (1) the maximum number of clauses can be satisfied (the whole CNF being satisfied translates to $\mathcal{D}'$ being fixed); and (2) the total weight of variable assignments to *true* is minimum, which translates to the cost of changing $\mathcal{D}$ to $\mathcal{D}'$ being minimum. Several high-performance tools for SAT (SAT-solvers) are in place [4] and have proved to be effective in areas such as software verification, AI, and operations research.

Weighted variables. Each assignment $t[A] \leftarrow a$ in a candidate fix F is represented by a Boolean variable $x^a_{t[A]}$. We denote by $\mathsf{wt}(x^a_{t[A]})$ the weight of $x^a_{t[A]}$. Intuitively, a variable $x^a_{t[A]}$ is *true* means that the attribute value of $t[A]$ should be changed to value a and if this update is applied to the database the cost is $\mathsf{wt}(x^a_{t[A]})$. Naturally, if $t[A] = a$, we have $\mathsf{wt}(x^a_{t[A]}) = 0$. Note that the compression technique discussed in Section 4 can be readily applied here. For each variable in a CNF corresponding to a super cell c, its weight is the cost of changing a normal cell multiplied by the cardinality of the super cell, *i.e.,* $|\mathsf{ext}(c)|$.

Clauses. The clauses are designed to represent three different semantics: (a) *inclusive* assignments: each cell that causes a violation should be assigned a (possibly) new value; (b) *exclusive* assignments: each cell can be assigned only one value; and (c) violation *avoidance*: cells that cause a violation cannot coexist with current values.

Given a set $\mathcal{F}$ of candidate fixes, let $\mathsf{val}(\mathcal{F}, t[A])$ denote $\{t[A]\} \cup \{a_i \mid (t[A] \leftarrow a_i) \in \mathcal{F}\}$, *i.e.,* the set of all values that $t[A]$ can be assigned to, including its current value. For example, for the candidate fix F_1 in Fig. 4, we have $\mathsf{val}(\{F_1\}, r_4[\text{country}]) = \{\text{UK, Netherlands}\}$.

(a) Inclusive assignments. For each cell $t[A]$ such that $\mathsf{val}(\mathcal{F}, t[A])$ satisfies $n > 1$ where $n = |\mathsf{val}(\mathcal{F}, t[A])|$, we generate a clause with the form $(x^{a_1}_{t[A]} \vee \cdots \vee x^{a_n}_{t[A]})$. Intuitively, this clause is to ensure that at least one of the values should be assigned to $t[A]$.

(b) Exclusive assignments. For each cell $t[A]$ such that $\mathsf{val}(\mathcal{F}, t[A])$ satisfies $n > 1$ where $n = |\mathsf{val}(\mathcal{F}, t[A])|$, we generate $n(n-1)/2$ clauses, where each clause is in the form of $(\neg x^{a_i}_{t[A]} \vee \neg x^{a_j}_{t[A]})$, for each $a_i, a_j \in \mathsf{val}(\mathcal{F}, t[A])$, and $a_i \neq a_j$.

Intuitively, these clauses assure that at most one of the values can be assigned to $t[A]$.

(c) Violation avoidance. For each violation that consists of a set of n cells in the form of $t_i[A_i] = a_i$ for $i \in [1, n]$, A_i an attribute in t_i and a_i the current cell value of $t_i[A_i]$, we generate a clause $(\neg x^{a_1}_{t_1[A_1]} \vee \cdots \vee \neg x^{a_n}_{t_n[A_n]})$. Intuitively, this clause assures that these values cannot all be true simultaneously since they cause a violation when putting together.

Recall that the cost of each variable is determined by a function $\textsf{cost}(c, v_1, v_2)$ that returns a value representing the cost of changing the value of cell c from v_1 to v_2. By default, it returns 1 (*i.e.,* unit update) when $v_1 \neq v_2$ and 0 otherwise.

Example 4: To better understand how the variable-weighted MAX-SAT solver works, we consider the six candidate fixes shown in Fig. 4. The variables with weight 1 are shown in the table below, *e.g.,* the variable $x^{Netherlands}_{r_4[\textsf{country}]}$ indicates that $r_4[\textsf{country}]$ can be changed to Netherlands. The variables with weight 0 are omitted.

$x^{Netherlands}_{r_4[\textsf{country}]}$	$x^{66700543}_{r_3[\textsf{phn}]}$	$x^{Netherlands}_{r_1[\textsf{country}]}$	$x^{Netherlands}_{r_2[\textsf{country}]}$	$x^{UK}_{r_3[\textsf{country}]}$

We generate five *inclusive (resp. exclusive)* clauses, corresponding to the above five non-zero weight variables, as shown in the table below.

Inclusive assignments	Exclusive assignments
$(x^{UK}_{r_4[\textsf{country}]} \vee x^{Netherlands}_{r_4[\textsf{country}]})$	$(\neg x^{UK}_{r_4[\textsf{country}]} \vee \neg x^{Netherlands}_{r_4[\textsf{country}]})$
$(x^{66700541}_{r_3[\textsf{phn}]} \vee x^{66700543}_{r_3[\textsf{phn}]})$	$(\neg x^{66700541}_{r_3[\textsf{phn}]} \vee \neg x^{66700543}_{r_3[\textsf{phn}]})$
$(x^{UK}_{r_1[\textsf{country}]} \vee x^{Netherlands}_{r_1[\textsf{country}]})$	$(\neg x^{UK}_{r_1[\textsf{country}]} \vee \neg x^{Netherlands}_{r_1[\textsf{country}]})$
$(x^{UK}_{r_2[\textsf{country}]} \vee x^{Netherlands}_{r_2[\textsf{country}]})$	$(\neg x^{UK}_{r_2[\textsf{country}]} \vee \neg x^{Netherlands}_{r_2[\textsf{country}]})$
$(x^{UK}_{r_3[\textsf{country}]} \vee x^{Netherlands}_{r_3[\textsf{country}]})$	$(\neg x^{UK}_{r_3[\textsf{country}]} \vee \neg x^{Netherlands}_{r_3[\textsf{country}]})$

Moreover, four clauses are generated to avoid the four violations from Fig. 4, as shown in the table below, where "12 . . ." is the abbreviation for "12 Holywell Street".

Avoid violations
$(\neg x^{31}_{r_4[\textsf{CC}]} \vee \neg x^{UK}_{r_4[\textsf{country}]})$
$(\neg x^{David}_{t_1[\textsf{FN}]} \vee \neg x^{Jordan}_{t_1[\textsf{LN}]} \vee \neg x^{12\ldots}_{t_1[\textsf{St}]} \vee \neg x^{Oxford}_{t_1[\textsf{city}]} \vee \neg x^{66700543}_{t_1[\textsf{tel}]}$ $\vee \neg x^{David}_{r_3[\textsf{FN}]} \vee \neg x^{Jordan}_{r_3[\textsf{LN}]} \vee \neg x^{12\ldots}_{r_3[\textsf{str}]} \vee \neg x^{Oxford}_{r_3[\textsf{city}]} \vee \neg x^{66700541}_{r_3[\textsf{phn}]})$
$(\neg x^{44}_{r_1[\textsf{CC}]} \vee \neg x^{UK}_{r_1[\textsf{country}]} \vee \neg x^{44}_{r_3[\textsf{CC}]} \vee \neg x^{Netherlands}_{r_3[\textsf{country}]})$
$(\neg x^{44}_{r_2[\textsf{CC}]} \vee \neg x^{UK}_{r_2[\textsf{country}]} \vee \neg x^{44}_{r_3[\textsf{CC}]} \vee \neg x^{Netherlands}_{r_3[\textsf{country}]})$

A variable-weighted MAX-SAT solver will take the conjunction of all the above clauses as input. There are two possible ways to fix the database, *i.e.,* the whole CNF is satisfiable, with a cost of 4 and 3 resp. as shown below.

Cost	Truth assignments of non-zero weight variables
4	$x^{UK}_{r_4[\textsf{country}]}, x^{66700543}_{r_3[\textsf{phn}]}, x^{Netherlands}_{r_1[\textsf{country}]}, x^{Neterlands}_{r_2[\textsf{country}]}$
3	$x^{UK}_{r_4[\textsf{country}]}, x^{66700543}_{r_3[\textsf{phn}]}, x^{UK}_{r_3[\textsf{country}]}$

The variable-weighted SAT solver will choose the second option above, which will update three cells, *i.e.,* changing $r_4[\textsf{country}]$ to UK, $r_3[\textsf{phn}]$ to 66700543 and $r_3[\textsf{country}]$ to UK, since it has the lowest total cost 3. □

5.2 An Equivalence Class based Algorithm

An alternative implementation of our core algorithm for data repairing is based on equivalence classes [8]. These were originally designed for FD and CFD violations. We first revise this notion and then present our strategy.

Equivalence classes. An *equivalence class* consists of a set E of cells. In a database $\mathcal{D}$, each cell c has an associated equivalence class, denoted by $\textsf{eq}(c)$. An equivalence class E is associated with a set of candidate values, denoted by $\textsf{cand}(E)$, and a unique *target* value, denoted by $\textsf{targ}(E)$.

Given a set $\mathcal{V}$ of violations and a set $\mathcal{F}$ of candidate fixes as input, we build equivalence classes and find fixes as follows:

(i) Initialization. Each cell c involved in $\mathcal{V}$ is an equivalence class $\textsf{eq}(c)$, and its candidate value is its current cell value.

(ii) Merge equivalence classes. (a) If there are two candidate fixes $c_1 \leftarrow c_2$ and $c_2 \leftarrow c_1$ in $\mathcal{F}$, the two equivalence classes for $\textsf{eq}(c_1)$ and $\textsf{eq}(c_2)$ will be merged into one, and the new set of candidate values is the union of the two sets of candidate values from $\textsf{eq}(c_1)$ and $\textsf{eq}(c_2)$, *i.e.,* $\textsf{cand}(\textsf{eq}(c_1)) \cup \textsf{cand}(\textsf{eq}(c_2))$. (b) If there is only one candidate fix $c_1 \leftarrow c_2$ (*i.e.,* $c_2 \leftarrow c_1$ is not a candidate fix), the candidate values of $\textsf{eq}(c_1)$ will become $\textsf{cand}(\textsf{eq}(c_1)) \cup \textsf{cand}(\textsf{eq}(c_2))$.

(iii) Assign a target value. For each equivalence class E, select one target value $\textsf{targ}(E)$ from its candidate values $\textsf{cand}(E)$, such that the total cost of changing all cell values in E to $\textsf{targ}(E)$ is minimum.

Using equivalence classes, we separate the decision of which cell values should be the same from the decision of what target value should be assigned to an equivalence class. We defer the assignment of $\textsf{targ}(E)$ as late as possible to reduce poor local decisions. Note again that the compression technique discussed in Section 4 can be readily applied here. The cost of making a super cell value change in an equivalence class is multiplied by $|\textsf{ext}(c)|$, the cardinality of the super cell (see Section 4.2).

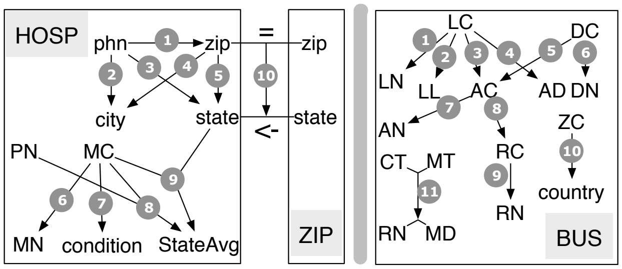
6. EXPERIMENTAL STUDY

Using two real-life datasets, we evaluated our data cleaning system NADEEF along with four dimensions: (1) *Generality*: the programming interface of NADEEF can be used to specify multiple types of rules. (2) *Extensibility*: different core algorithms can be used to detect errors and clean data. (3) *Effectiveness*: while aiming to be generic, our method can find a *fixed* database with high accuracy compared with existing techniques. (4) *Efficiency*: our system can work on data in reasonable sizes.

Experimental Setting. We used two real-life datasets.

(1) HOSP *data* was taken from US Department of Health & Human Services (`http://www.hospitalcompare.hhs.gov`). It has 100K records with 9 attributes used in data quality rules: Provider Number (PN), zip, city, state, phone, Measure Code (MC), Measure Name (MN), condition, and stateAvg. We also downloaded another table of US ZIP codes from (`http://databases.about.com/od/access/a/zipcodedatabase.htm`). The table has 43K tuples with two attributes: zip and state.

(2) BUS *data* is a one table dataset obtained by joining 8 tables using primary-foreign key relationships. These tables were from the UK government public datasets (`http://data.gov.uk/data`). The table has 160K tuples with 16 attributes relevant to data quality rules: Locality Code (LC), Locality Name (LN), Locality Name Language (LL), Administrative Area Code (AC), Area District Code (AD), Cre-



(a) HOSP dataset (b) BUS dataset

Figure 9: Data quality rules

ation DateTime (CT), Modification DateTime (MT), Revision Number (RN), Modification (MD), Area Name (AN), Region Code (RC), Region Name (RN), District Code (DC), District Name (DN), Bus Zone Code (ZC) and country.

Dirty datasets. Dirty data was generated as follows. For any dataset $\mathcal{D}$, we first cleaned $\mathcal{D}$ to get $\mathcal{D}'$ by using some cleaning algorithms followed by careful manual check, ensuring that $\mathcal{D}'$ is consistent *w.r.t.* the defined rules. We treated $\mathcal{D}'$ as the ground truth. We then added noise to $\mathcal{D}'$, which is controlled by noise rate noi%. Note that, we only added noise to the attributes that are used in data quality rules.

Data quality rules. All data quality rules have been designed manually (see Exp-1 below for details).

Algorithms. The system NADEEF was implemented in C++, including the following: (*i*) The algorithm for violation detection, using partitioning and compression techniques (Section 4.2) by default, referred to as GetVio. (*ii*) The algorithm for computing fixes using the variable-weighted SAT-solver (see Section 5.1), referred to as WSAT. We used a variable-weighted MAX-SAT solver from [22]. (*iii*) The algorithm for computing fixes using equivalence classes (Section 5.2), referred to as EQU. For comparison, we obtained the implementation of two algorithms for FD repairing, a cost-based heuristic method [5], referred to as HEU, and a vertex cover based approach [20], referred to as VER. Both approaches were implemented in Java.

We conducted all experiments on a Windows machine with an Intel 3.4GHz Intel CPU and 8GB of Memory.

Measuring Quality. To assess the accuracy of data cleaning algorithms, we used precision, recall and F-measure, which are commonly used in measuring the result of repairs, where

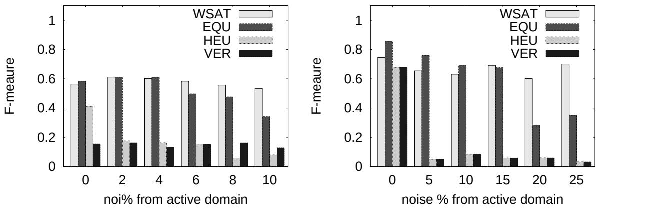
$$\text{F-measure} = 2 \cdot (\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall}).$$

Here, precision is the ratio of attributes correctly updated to the number of all the attributes that have been updated, and recall is the ratio of correctly updated attributes to the number of all erroneous attributes.

Experimental Results. We now report our findings for the four dimensions mentioned earlier.

Exp-1: Generality. We show the *generality* of our framework by demonstrating that it can be used to specify multiple types of data quality rules. Fig. 9(a) depicts the rules defined for HOSP dataset and Fig. 9(b) depicts the rules for BUS dataset.

We defined 10 rules over the HOSP dataset, where rules 1-9 are all FDs and rule10 is an MD. For example, rule1 (resp. rule9) states that in table HOSP, phn (resp. MC and state) determines zip (resp. StateAvg). Rule10 states that if two zip code values from table HOSP and table ZIP are the same, but their state values are different, then the state value from ZIP table is more reliable. All of these data quality rules are defined over a pair of tuples.



(a) HOSP (b) BUS

Active domain	WSAT	EQU	HEU	VER
0	132/186	137/187	73/73	39/220
2	141/195	141/195	72/553	40/237
4	137/199	139/199	70/606	42/374
6	127/178	126/250	67/612	39/265
8	132/204	136/301	59/1749	38/260
10	121/199	122/461	72/1543	26/189

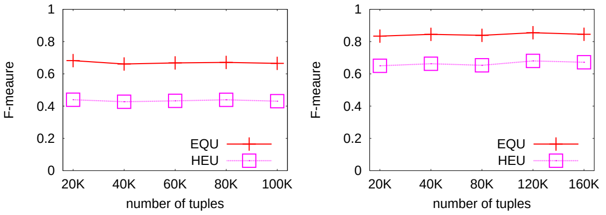
Figure 10: Accuracy when the noise is from the active domain

There are 11 rules defined over the BUS dataset, where rules 1-10 are FDs, and rule11 is a customized rule. Rule11 states that, for each tuple, if its Creation DateTime (CT) and Modification DateTime (MT) are the same, then its Revision Number (RN) must be 0, and its status Modification (MD) must be new. We can use rule11 to fix erroneous cells as well as capturing missing values since the values for many RN and MD cells are missing. Remark that rules 1-10 are defined for a pair of tuples, and rule11 is defined on a single tuple.

For the rule classes that the users have to implement, FDs (resp. MDs) only require 65 and 34 (resp. 45 and 46) lines of code to define the functions for detecting violations and generating candidate fixes, respectively. For the customized rule (*i.e.,* rule11 for BUS data), the users need to write 50 lines of code in total. We see that using the concept of programming interface as defined in NADEEF, users will have to write only a few lines of code for data quality rules that are relevant to their dataset, without having to write an entire data cleaning system specially crafted for these rules. Note that for the FDs mentioned above, we assign them a default dynamic semantics of fixing errors by changing the values from the right hand side of the rules (see *e.g., Rule2* in Fig. 3).

Remark. To ease the use of NADEEF, we have implemented some common classes that can be easily reused by users. Specifically, we have implemented *Rule* classes for CFDs (FDs) and MDs. For example, to specify various FDs, users only need to specify the left- and right-hand side attributes of each FD. We have also implemented some common similarity functions, with string edit distance by default.

Exp-2: Extensibility. We have already mentioned (Section 5) that we provide two different algorithms for fixing errors (*i.e.,* GetFix), namely WSAT and EQU. A user can specify in a configuration file which algorithm to use. Users can also specify a totally different core algorithm GetFix as long as that algorithm can take as input sets of violations



(a) HOSP (b) BUS

Figure 11: Accuracy when varying the size of data

and candidate fixes, and returns a set of fixes to be applied to the database. Moreover, not only GetFix, but also the algorithm for GetVio, can be overridden.

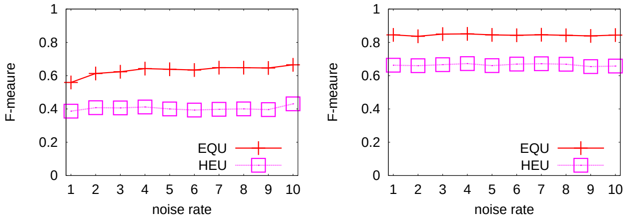
Exp-3: Effectiveness. We evaluated the accuracy of different cleaning algorithms by varying noise rate (noi%) and the size of data. Noise is added by either introducing typos to an attribute value or changing an attribute value by another one from the active domain of that specific attribute.

Varying noise from the active domain. In this series of experiments, we fixed the overall noise rate to 1 percent and then varied the noise rate from the active domain (x-axis in both charts from Fig. 10). The results of comparing WSAT, EQU, HEU and VER are given in Fig. 10 (the y-axis represent F-measure values), where Fig. 10(a) is for HOSP data and Fig. 10(b) is for BUS data. We used 10K tuples for both datasets with FDs only.

Figure 10 shows that when there is no noise from active domain, *i.e.,* x-axis value is 0, both WSAT and EQU have comparable F-measure values with HEU as shown in Fig. 10(a) and 10(b). However, when there is noise from active domain, the F-measure values of both EQU and HEU drop quickly. This tells us that WSAT is not sensitive to noise from the active domain, while EQU and HEU are sensitive.

We explain the above results as follows: (1) WSAT is not sensitive to noise from active domain since it treats variables *independently* with the target of selecting variables with the least cost to satisfy the whole CNF or a maximum number of clauses. (2) EQU is sensitive since when there are erroneous values from the active domain, the algorithm merges (originally) irrelevant equivalence classes. Hence, when making decision to set an equivalence class a target value, many assignments are wrong. The case for HEU is similar. (3) VER is sensitive since when there are errors from the active domain, the algorithm will erroneously connect some tuples using hyper-edges (see [20]) as violations, which might connect two previously irrelevant violations and reduce the accuracy when repairing the data. (4) The reason that our approaches have higher accuracy is that users have to specify possible ways to fix errors, which avoids blindly making the database consistent by only targeting minimality.

We further explain the reason the recall of VER is low using the table in Fig. 10, the one for BUS data is similar and omitted for space constraints. The first column corresponds to the x-axis values in Fig. 10(a), *i.e.,* the percentage of noise from the active domain. Each cell in the table is for the number of correct changes over the number of changes. For example, the cell 132/186 represents that WSAT changed 186 values, and 132 of them are correct changes. We notice that when the noise from the active domain is large, *e.g.,* the last row, VER made only 189 changes to make the database consistent, while HEU made 1543 changes. Hence, VER uses less changes than HEU to compute a consistent database.



(a) HOSP (b) BUS

Figure 12: Accuracy when varying noise rate

This group of experiments tells us that: (1) The new problem studied in this paper is meaningful, *i.e.,* instead of trying to compute a consistent database, we should try to resolve errors when candidate fixes are known. It highlights the need for users to provide useful information to guide the process of repairing data. (2) Equivalence class based solutions, which cover a large part of existing data cleaning algorithms, are sensitive to the noise from the active domain.

To favor the other approaches, in the following, we only consider noise from typos. Since WSAT and VER cannot scale well, we focus our comparisons mainly on EQU and HEU. Notice that for WSAT, we need a variable-weighted MAX-SAT solver. However, most popular SAT solvers are non-weighted or clause-weighted. While MiniSAT [18] is very efficient, its effectiveness was too low when we tried it and thus is not reported in this paper. Whenever a more robust variable-weighted MAX-SAT solver becomes available, we will evaluate it within our system.

Noise from typos only. We evaluated the effectiveness of different algorithms when there is only noise from typos. Figure 11(a) (resp. Fig. 11(b)) shows the case for HOSP (resp. BUS) when fixing noi% at 1% while varying the size of the data from 20K to 100K tuples (resp. 20K to 160K tuples). Moreover, Fig. 12(a) (resp. Fig. 12(b)) shows the result of HOSP (resp. BUS) data when varying the noise rate noi% from 1% to 10%, with 100K tuples (resp. 40K tuples). The results for the datasets of different sizes show similar trend, and hence are omitted here.

Figures 11(a) and 11(b) show that for different sizes of data, the F-measure values of different algorithms stay almost the same if the noise is only from typos, which verifies the previous group of experiments. The reason that EQU is better confirms that it is really helpful to have users specify the dynamic semantics of rules, *i.e.,* telling the system about the different alternatives to modify attributes when there is a violation.

Figures 12(a) and 12(b) show that when there is only noise from typos, existing algorithms are not sensitive to the noise rate. On the one hand, typos will only introduce independent violations. When treating these violations separately, the same algorithm will get a similar F-measure value. On the other hand, when there are errors from the active domain, most algorithms will associate (originally) irrelevant violations. This will negatively affect the results of most algorithms, which has been verified in Fig. 10.

Interleaving various types of data quality rules. We evaluated the effect of executing various types of data quality rules together versus executing them sequentially. Recall Fig. 9(a) for HOSP dataset, the MD rule10 overlaps with other FD rules. However, for BUS dataset, the customized rule11 has no overlap with other FD rules. Hence, we focus on evaluating the cleaning of HOSP in different rule order.

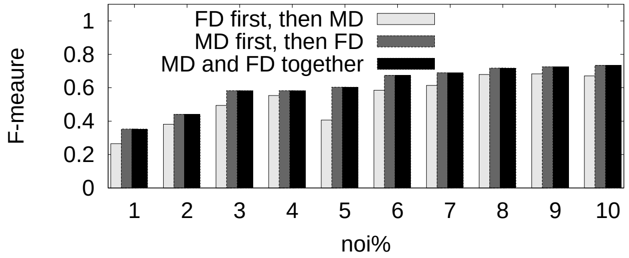


Figure 13: Interleaving various types of rules

For BUS, since there is no overlap of rules, executing them in any order makes no difference.

We ran algorithm HEU on HOSP data with 100K tuples, varying the noise rate noi% from 1% to 10%. To clearly show the result, we added 5% extra noise to the attributes that are related to the intersected rules, *i.e.,* attributes zip and state. The result is shown in Fig. 13. When executing FDs first followed by MDs, the overall performance is worse than executing MDs first. Moreover, executing MDs and FDs together has the same performance as executing MDs first.

This experiment tells us that the order of executing multiple types of rules matters. That is, executing multiple types of rules in different orders will get different results. However, in practice, it is impossible to know the optimal order *a priori.* Hence, when having to deal with multiple types of rules, we should treat them *holistically*, as verified by Fig. 13.

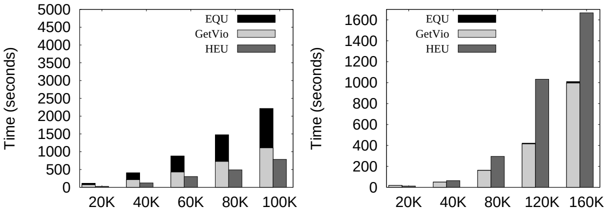
Exp-4: Efficiency. In the last group of experiments, we studied the efficiency of various algorithms. We start by comparing EQU and HEU, followed by evaluating WSAT. Figure 14 shows running time of EQU and HEU: Fig. 14(a) relates to Fig. 11(a), and Fig. 14(b) relates to Fig. 11(b).

In these two figures, the x-axis represents the number of tuples and the y-axis the running time. We show two components of the running time of our method, where the lower part is for detecting violations (*i.e.,* GetVio) and the upper part is for repairing errors using EQU. The time for Updater is in milliseconds and thus not reported here.

The results show the following: (a) EQU and HEU deliver good results but for different applications, *e.g.,* HEU is faster for HOSP (Fig. 14(a)) while EQU is faster for BUS (Fig. 14(b)). (b) GetVio takes some time, but the time of EQU for computing fixes is quite efficient, which proves the benefit of our partitioning and compression techniques (see Fig. 8).

We also studied the efficiency of WSAT on HOSP and BUS data, where we fixed noi% at 1% while varying the size of data from 20K to 100K for HOSP and 20K to 160K for BUS. The running time is given in Fig. 15. The running times above 2 hours are not given since they did not run to completion. The high running times come from the inherent complexity of the problem of variable-weighted MAX-SAT problem, which is NP-hard. Thus, using WSAT for the whole dataset is not practical. However, it opens the opportunities to design an optimizer that first partitions the whole data cleaning problem into many smaller independent groups, then determines which algorithm, *e.g.,* HEU or WSAT, to invoke for each group.

Summary. From the above experimental study, we conclude that: (a) Providing a *generalized* programming interface, which requires minimum user efforts to specify data quality rules, is practically needed and is possible (Exp-1). (b) By making our system *extensible*, we can benefit from algorithms that expert users implement for their own applications to replace the default core algorithms, especially for GetVio and GetFix (Exp-2). (c) Our systems can achieve better accuracy than existing methods since the users provide the system with the dynamic semantics to resolve violations (Exp-3). (d) Multiple types of data quality rules should be treated *holistically* (Exp-3), since in practice, it is difficult to know *a priori* the best order of sequential execution for algorithms designed for different rules. (e) Our system can work for data in reasonable sizes (Exp-4).



(a) HOSP (b) BUS

Figure 14: Efficiency study

HOSP	time (sec)	BUS	time (sec)
20K	617.4	20K	131.8
40K	4759.3	40K	482.1
60K	> 2 hours	80K	2473.2
80K	> 2 hours	120K	> 2 hours
100K	> 2 hours	160K	> 2 hours

Figure 15: Running time for WSAT

7. RELATED WORK

The first serious discussions and analyses of data repairing have emerged in [16]. After that, a large and growing body of literature has investigated ETL tools (see [2] for a survey), which support data transformations, and can be employed to merge and repair data [24]. Recently, there has been an increasing amount of literature on dependency theory (*a.k.a.* integrity constraints) to specify data consistency for data repairing, *e.g.,* FDs [28], its extension CFDs [8, 12], MDs [13], FDs and INDs [5], and denial constraints [1]. However, they are in the class of universally quantified first-order sentences. Their limited expressiveness often does not allow to state problems commonly found in real life data as violations of these dependencies [11]. Through its programming interface, our framework is expressive enough to specify these problems as well as any of the existing constraints.

Data quality techniques often rely on domain-specific similarity and matching operators, beyond pure first-order logic. While these domain-specific operations may not be themselves expressible in any reasonable declarative formalism, it is still possible to integrate them into the framework of dependencies, especially for record matching (*a.k.a.* record linkage, entity resolution, and duplicate detection, see [10] for a survey). In contrast to matching rules (*e.g.,* [13,19,27]), our approach is more general since it also considers data repairing, among other things.

A general approach to tackle the problem of entity resolution is to define two functions, namely match and merge, over two tuples as proposed in [3], where match identifies duplicates and merge combines the two duplicated records into one. One can easily verify that the match can be naturally defined using function vio(s_1, s_2) to find duplicates, and fix()

can be used to merge duplicates. However, merge cannot be used to represent updates of finer granularity (only some cell needs to be changed) as what fix() does in our system.

Several repairing algorithms have been proposed [5, 8, 15, 16, 23, 29]. Heuristic methods have been developed based on FDs and INDs [5], CFDs [12], and editing rules [16]. The methods of [5, 8] employ confidence values provided by the users to guide a repairing process. Statistical inference is studied in [23] to derive missing values. To ensure the accuracy of the generate repairs, [23, 29] require consulting the users. We do not assume the availability of confidence values in NADEEF. Moreover, while we do not support user interactions in the current implementation of NADEEF, the metadata we are handling can feed to a data quality dashboard that can then visualize the necessary information to understand the current health of the data and eventually allow the users to intervene to guide the cleaning process.

A number of studies have been proposed to tackle different data quality rules in one framework. The work [14] studies the interaction between record matching (MDs) and data repairing (CFDs). AJAX [17] and TAILOR [9] are toolboxes for record linkage. In contrast, the proposed system (a) covers a more general spectrum of data quality rules; (b) is extensible such that users can plug-in their own cores for error detection and data repairing, among other things.

8. CONCLUSION

We presented NADEEF, a commodity data cleaning system. The main design of NADEEF is to separate a *programming interface* that allows users to flexibly define multiple types of data quality rules about both their static semantics and dynamic semantics, and a *core* that implements algorithms to detect and repair dirty data by treating multiple types of quality rules holistically. We have demonstrated that our interface is general and expressive enough to define data quality rules beyond the well known ETL rules, CFDs and MDs. We have also shown the *extensibility* of NADEEF by showing that users can plug in other core algorithms, allowing experts to further customize our system.

Several extensions are underway. (1) To handle large volume of data, we plan to move our system from being memory-based to being disk-based, using an open source database *e.g.,* PostgresSQL. (2) We are designing more user friendly interface (*i.e.,* GUI) to help users define their rules easier. (3) We plan to incorporate various indices and blocking techniques in our framework to efficiently support similarity comparisons. (4) We plan to investigate techniques for partitioning a big cleaning problem into multiple small ones such that each one of them can be executed separately. This would provide opportunities for designing an optimizer to select the appropriate core implementation (*i.e.,* detection and repairing) for each partition as well running NADEEF in a parallel and distributed environment. (5) We will design and implement a live data quality dashboard on top of NADEEF and explore summarization techniques to more effectively involve users in the cleaning process.

9. REFERENCES

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. *TPLP*, 3(4-5), 2003.

[2] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[3] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1), 2009.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[5] 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.

[6] L. Bravo, W. Fan, F. Geerts, and S. Ma. Increasing the expressivity of conditional functional dependencies without extra complexity. In *ICDE*, 2008.

[7] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *TKDE*, 19(1), 2011.

[8] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.

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

[10] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1), 2007.

[11] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.

[12] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.

[13] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2(1), 2009.

[14] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD Conference*, 2011.

[15] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2), 2012.

[16] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 71(353), 1976.

[17] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.

[18] E. Giunchiglia and A. Tacchella, editors. *Theory and Applications of Satisfiability Testing*, SAT, 2004.

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

[20] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.

[21] G. M. Kuper. Aggregation in constraint databases. In *PPCP*, 1993.

[22] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In *SAT (Selected Papers)*, 2004.

[23] C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[24] F. Naumann, A. Bilke, J. Bleiholder, and M. Weis. Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Eng. Bull.*, 29(2), 2006.

[25] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In *WSDM*, 2012.

[26] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[27] S. E. Whang, O. Benjelloun, and H. Garcia-Molina. Generic entity resolution with negative rules. *VLDB J.*, 18(6), 2009.

[28] J. Wijsen. Database repairing using updates. *TODS*, 30(3), 2005.

[29] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5), 2011.