

A SYSTEMS APPROACH TO RULE-BASED DATA CLEANING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Amr Ebaid

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Prof. Walid G. Aref, Co-Chair

Department of Computer Science

Prof. Ahmed K. Elmagarmid, Co-Chair

Department of Computer Science, Qatar Computing Research Institute

Dr. Mourad Ouzzani

Qatar Computing Research Institute

Prof. Sunil Prabhakar

Department of Computer Science

Prof. Christopher W. Clifton

Department of Computer Science

Prof. Jennifer Neville

Department of Computer Science

Approved by:

Prof. Voicu Popescu

Head of the Computer Science Graduate Program

To the late Dr. Ahmed Khaled Towfik, may Allah pour his mercy upon you.

To graduate students everywhere, hang in there.

ACKNOWLEDGMENTS

First and foremost, all thanks go to Allah for granting me the power to reach my destination and for his countless blessings throughout this tough challenging phase and my whole life.

I would like to express my deep gratitude to all my advisors. I do believe a PhD journey is all about the advisor, and I was really blessed to be accompanied and advised by great leaders and mentors during my long journey.

Dr. Mourad Ouzzani was there from the very beginning, taught me a lot about research and technical writing, had always been very helpful and cooperative, offered his much appreciated efforts and feedback whenever needed, and pushed me so that I can do my best and achieve my goals.

I am also very grateful to Prof. Walid Aref for his continuous support and guidance. I have learned a lot from him both academically and personally. Although he joined my advisory committee midway, I have always sought his advice and experience from day one. He has been kindly the first person we go to whenever we face obstacles in school and in life. He stands as my role model and I wish one day I can be to someone the same way he is to me.

I really cannot thank Prof. Ahmed Elmagarmid enough for believing in me, supporting me and seeing what I myself could not see sometimes. He has been like a caring father as well as an extraordinary visionary advisor. Despite his extra busy schedule, he has always been available and extremely encouraging, that I never hesitated to ask for his advice and words of wisdom. I truly cannot find the words to express my endless sincerest gratitude, and I am truly grateful for the opportunity to know him personally and extremely honored and proud that I get to call myself his student.

I would like to thank the brilliant scientists and researchers I have worked with either in Purdue or in Qatar Computing Research Institute. I could not ask for better collaboration, and I have really learned a lot from each and every one of them.

I am also thankful to the professors and the staff of the Department of Computer Science in Purdue University. I have really learned a lot from you all, and I am grateful for the various chances I got to deal with such nice and pleasant people, being it the excellent teachers I took courses with, the outstanding professors I assisted in teaching multiple classes, or the amazing team with whom I worked on developing myCS portal.

My deepest gratitude of course goes to my wife, Sohila, and my son, Bilal, the jewel in the crown. You literally are the light in my path, and I would have been completely lost without you. Thank you for your limitless support, enduring love and bearing with me through all these difficulties and hard times.

Furthermore, I express my utmost appreciation to my dear family. I thank my siblings, Alaa, Ashraf and Areej, and dedicate this dissertation particularly to my heroes, my loving father, Hamdy, and devoted mother, Amal. I know you have waited for this long enough, and I hope this journey's end brings some happiness to your hearts. I hope you are proud of me, because no language nor words can describe how much I am proud of and thankful to you.

An old Arabic proverb states that “the neighbour before the house and the companion before the road”. That is why I am really grateful to all the best friends I had here In Purdue. I don't want to forget someone, but you all know you have a special place in my heart. If there is just one reason that makes me absolutely delighted and relieved I chose Purdue, it is the memories we had together that I will always remember and cherish forever.

Last but not least, I would like to thank the members of my PhD advisory and examining committees. I extremely value your insightful comments and feedback. I will always be looking up and forward to your astounding contributions that help shape the field and our future.

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	ix
ABSTRACT	xi
1 INTRODUCTION	1
1.1 The Vital Need for Data Quality	1
1.2 Data Cleaning Approaches	2
1.3 Motivating Example	3
1.4 Related Work	5
1.5 Limitations of Existing Solutions	6
1.6 Challenges and Objectives in Rule-Based Data Cleaning Systems	6
1.6.1 Heterogeneity and Interdependency	7
1.6.2 Extensibility	7
1.6.3 Continuity	7
1.6.4 Explainability	7
1.6.5 Other Challenges	7
1.7 Research Contributions	8
1.8 Dissertation Outline	10
2 NADEEF: GENERALIZED AND EXTENSIBLE DATA CLEANING	11
2.1 An End-to-End Rule-Based Data Cleaning System	11
2.1.1 Challenges	11
2.1.2 Contributions	13
2.2 Related Work	14
2.3 Fundamentals	15
2.4 System Architecture	19
2.5 NADEEF's Programming Interface	21
2.5.1 Programming Interface Operators	22
2.5.2 Sample Rules	24
2.5.3 Programming Interface Optimizations	24
2.6 Inside NADEEF: System Core	28
2.6.1 Violations Detection	28
2.6.2 Data Repairing	33
2.7 Experimental Study	39
2.7.1 Experimental Settings	39

	Page
2.7.2 Generality	41
2.7.3 Extensibility	43
2.7.4 Effectiveness	43
2.7.5 Efficiency	49
2.7.6 Summary	50
2.8 Concluding Remarks	51
3 CONTINUOUS DATA CLEANING	52
3.1 Dynamic Data and the Need for Continuous Data Cleaning	52
3.1.1 Motivating Scenarios	53
3.1.2 Challenges	56
3.1.3 Contributions	57
3.2 Related Work	58
3.3 Limitations of Existing Data Cleaning Approaches	59
3.3.1 Domino Effect: Violations Propagation	60
3.3.2 Memoryless Repairs	61
3.4 Support for Continuous Data Cleaning in NADEEF+	63
3.4.1 Continuous Violations Detection	63
3.4.2 Continuous Data Repairing	67
3.5 Experimental Study	72
3.5.1 Experimental Settings	73
3.5.2 Continuity	74
3.6 Concluding Remarks	77
4 DATA CLEANING EXPLANATIONS	78
4.1 Human-in-the-Loop and the Need for Explainability	78
4.1.1 Challenges	79
4.2 Related Work	79
4.3 Properties of Explanations	81
4.4 Explainability in Data Integration Systems	82
4.5 Data Cleaning Explanations in NADEEF	84
4.5.1 Violations Explanations	84
4.5.2 Repairs Explanations	87
4.6 Explanations beyond Rule-based Data Cleaning	88
4.6.1 Use Case: Entity Resolution Explanations	88
4.6.2 Contributions	89
4.6.3 Overview of EXPLAINER	90
4.6.4 Case Studies	92
4.7 Concluding Remarks	100
5 CONCLUSIONS	101
REFERENCES	104
VITA	110

LIST OF TABLES

Table	Page
2.1 Sample Violations and Candidate Fixes	29
2.2 Benefits of Partitioning and Compression	33
2.3 MAX-SAT Solver Weighted Variables	36
2.4 Inclusive and Exclusive Assignments for MAX-SAT Solver	36
2.5 Violations Avoidance for MAX-SAT Solver	36
2.6 Possible Repairs of MAX-SAT Solver	37
2.7 Running Times for WSAT	50
3.1 Graphs Connected Components for HOSP Dataset	74
3.2 Response Times for Minor Data Changes	75
4.1 Violations (Explanations) in NADEEF	84
4.2 Repairs (Explanations) in NADEEF	87

LIST OF FIGURES

Figure	Page
1.1 Motivating Example	4
1.2 Overview of NADEEF	8
2.1 Database with violations of various heterogeneous rules	16
2.2 Architecture of NADEEF	19
2.3 NADEEF's Programming Interface	21
2.4 Sample Rules in NADEEF	23
2.5 NADEEF's Optimized Programming Interface	25
2.6 NADEEF's Violations Detection Flow for Optimized Rules	26
2.7 Violations Detection Flow for φ_1 on <i>transactions</i>	27
2.8 Partitioning and Compression for φ_1	31
2.9 Violations Expansion	32
2.10 Equivalence Classes Repair Module	38
2.11 Data Quality Rules for Datasets	42
2.12 F-Measure with noise from active domain	44
2.13 F-Measure with variable data size	46
2.14 F-Measure with variable noise rate	46
2.15 Interleaving various types of rules	48
2.16 Running time with variable data size	49
3.1 Motivating Example for Continuous Data Cleaning	53
3.2 Dynamic Data and Domino Effect	60
3.3 Dynamic Data and Memoryless Repairs	62
3.4 Changes-Aware Programming Interface in NADEEF+	64
3.5 Violations Detection Flow in NADEEF+	65
3.6 Sample Changes-Aware Rule	66

Figure	Page
3.7 Dynamic Graphs Repair Module	69
3.8 Dynamic Data and Memoryful Repairs	71
3.9 Running Times for Major Data Changes	76
4.1 Explainability in Data Integration Systems	82
4.2 NADEEF's Data Quality Dashboard	86
4.3 Repairs Auditing in NADEEF	87
4.4 Entity Resolution Pipeline and EXPLAINER Architecture	92
4.5 Global Explanations in EXPLAINER - Feature Importance	93
4.6 Global Explanations in EXPLAINER - Bayesian Rule List (BRL)	94
4.7 Global Explanations in EXPLAINER - Feature Weights for Explanations	95
4.8 Model Analysis in EXPLAINER - Local Explanation by LIME	96
4.9 Model Analysis in EXPLAINER - Incorrect Predictions	97
4.10 Model Analysis in EXPLAINER - Features Frequent Itemsets	97
4.11 Differential Analysis in EXPLAINER - User Interface	99
4.12 Differential Analysis in EXPLAINER - Predictions Disagreement	99

ABSTRACT

Ebaid, Amr Ph.D., Purdue University, May 2019. A Systems Approach to Rule-Based Data Cleaning. Major Professors: Walid G. Aref, Ahmed K. Elmagarmid and Mourad Ouzzani.

High quality data is a vital asset for several businesses and applications. With flawed data costing billions of dollars every year, the need for data cleaning is unprecedented. Many data-cleaning approaches have been proposed in both academia and industry. However, there are no end-to-end frameworks for detecting and repairing errors with respect to a set of *heterogeneous* data-quality rules.

Several important challenges exist when envisioning an end-to-end data-cleaning system: (1) It should deal with heterogeneous types of data-quality rules and interleave their corresponding repairs. (2) It can be extended by various data-repair algorithms to meet users' needs for effectiveness and efficiency. (3) It must support continuous data cleaning and adapt to inevitable data changes. (4) It has to provide user-friendly interpretable explanations for the detected errors and the chosen repairs.

This dissertation presents a systems approach to rule-based data cleaning that is **generalized**, **extensible**, **continuous** and **explaining**. This proposed system distinguishes between a *programming interface* and a *core* to address the above challenges. The programming interface allows the user to specify various types of data-quality rules that uniformly define and explain what is wrong with the data, and how to fix it. Handling all the rules as black-boxes, the core encapsulates various algorithms to holistically and continuously detect errors and repair data. The proposed system offers a simple interface to define data-quality rules, summarizes the data, highlights violations and fixes, and provides relevant auditing information to explain the errors and the repairs.

1 INTRODUCTION

1.1 The Vital Need for Data Quality

High quality data is by far the most valuable asset of any corporation or organization, especially in the era of (big) data intensive economy. In a typical use case, a business would manage and process large amounts of data, in order to extract valuable information; a key component vital for providing services or making any decisions. The Data Warehousing Institute estimated that data quality problems cost U.S. businesses more than \$600 billion a year [1], and a recent study estimated that dirty and duplicate data cost the U.S. economy in excess of \$3 Trillion every year [2]. Moreover, the fact that more than 25% of critical data in Fortune 1000 companies is flawed [3] shows how important, yet challenging, realizing high quality data is. Hence, the evident urge for data cleaning systems; Gartner reported that the market for data cleaning systems is growing at 17% annually, substantially outpacing the 7% average of other IT segments [4].

When dealing with data quality, users and businesses face several challenging problems. Of which the most popular is *data duplicates*, when there are two or more data records referring to the same real world entity. This is a very familiar problem for data warehousing applications or when integrating data from multiple data sources.

Another critical problem is *data inconsistency*, when records do not obey data quality rules, such as integrity constraints or business rules. Then, comes *data inaccuracy*, when the data may be consistent with the rules, but contradicts with the truth. *Data incompleteness* is another problem encountered when there are missing records or values.

Even when the data might be consistent, accurate and complete, it still can become obsolete not reflecting the true recent values, hence *data outdatedness*. Experts

estimate that 2% of records in a customer file become obsolete in one month [1], *i.e.*, in a database of 500K customer records, 10K records may go stale per month, 120K records per year, and within two years about 50% of all the records may be obsolete [5].

These problems are caused by several reasons. Human errors come on top of list, with incorrect data entry and missing values. Then come technical factors such as inaccurate readings from sensors or devices, erroneous applications populating databases, faulty database designs that do not enforce the right quality constraints or integrating data from multiple heterogeneous sources. Moreover, because of data such as addresses, telephones, appointments, etc. changing all the time, databases can become obsolete giving outdated untruthful representation of the dynamic real world.

1.2 Data Cleaning Approaches

A recent study [6] categorized existing data cleaning solutions into these four different categories:

Quantitative Error Detection to expose outliers and data glitches. This comes very handy when dealing with sensory data or data streams to find values that deviate from the distribution of data in a column or table.

Record Linkage (*a.k.a.* Data Deduplication) with the goal of identifying distinct data tuples that refer to the same real world entities, and applying *data fusion* to consolidate those duplicate records into one.

Pattern Enforcement and Transformation to discover wither syntactic or semantic patterns in the data and detect the records that do not follow these patterns. This set of tools can resolve issues with formatting, misspellings, unmatching data types, etc.

Rule-Based Data Cleaning when a set of rules is enforced on the data. Data cleaning tools try to detect violations of these rules and repair data records to reach a consistent database instance satisfying the rules. Rules can vary from simple not-null constraints, to integrity constraints or multi-attribute dependencies, up to user-defined functions and custom business rules.

In this dissertation, we focus on rule-based data cleaning. Data quality rules, such as dependencies [7], denial constraints [8,9] or business rules, are used to detect violations and repair the data accordingly. A *violation* is a group of cells that do not conform to a given data quality rule. Intuitively, in order to resolve a violation, one or more of the data attributes involved in the violation must be changed, *a.k.a.* a *data repair*. This is the data cleaning approach that US national statistical agencies, among others, have been practicing for decades [10]. Thus, a typical data cleaning system would involve two phases in an iterative process, *violations detection*, to find errors in the data, and *data repairing*, to actually fix these errors. Such a process would target *consistency*, making the data satisfy the quality rules, and *accuracy*, being as close as possible to the ground truth, according to a specific *cost model*, *e.g.*, the least possible number of data changes or the minimal cost to repair the data.

1.3 Motivating Example

As a motivating example, consider the data in Figure 1.1. First, if we have an FD (Functional Dependency) [11] $\varphi_1 : \text{CC} \rightarrow \text{country}$, this will signal a violation in the tuples t_1 , t_2 and t_3 , since all of them have the same country code, but different countries. So, one possible repair would be to change $t_3[\text{country}]$ to UK.

Now, let's have another rule, an MD (Matching Dependency) [12] φ_2 , stating that if two records from **transactions** and **customers** both match on first name, last name, street, city and country, then the transaction country code and phone in **transactions** should match those of the master data in **customers**. With this rule, and the latest

	FN	LN	street	city	country	CC	tel
c_1 :	David	Jordan	12 Holywell Street	Oxford	UK	44	66700543
c_2 :	Paul	Simon	5 Ratcliffe Terrace	Oxford	UK	44	44944631

(a) C : An instance of schema **customers**

	FN	LN	street	city	country	CC	phone	when	where
t_1 :	David	Jordan	12 Holywell Street	Oxford	UK	44	66700543	12:00 6/5/2012	Netherlands
t_2 :	Paul	Simon	5 Ratcliffe Terrace	Oxford	UK	44	44944631	11:00 2/12/2011	Netherlands
t_3 :	David	Jordan	12 Holywell Street	Oxford	Netherlands	44	66700541	6:00 6/5/2012	NY, USA
t_4 :	Peter	Austin	7 Market Street	Amsterdam	Netherlands	31	55384922	9:00 6/2/2012	Netherlands

(b) T : An instance of schema **transactions**

Figure 1.1. Motivating Example

country change of t_3 , t_3 would match c_1 , and hence $t_3[\text{phone}]$ should be updated accordingly to 66700543.

Finally, let's assume we have a third business rule, φ_3 , that would mark two transactions as either a fraud or erroneously recorded, if both were by the same person (identified by the first name, last name, country code, and phone) and in two different locations within 1 hour (considering different time zones). Again, with the recent **phone** change of t_3 , a violation would be encountered between t_1 and t_3 ; two different transactions by David Jordan, in both NY, USA and Netherlands at the same exact time.

Clearly, we can see that when taken together, different data quality rules help each other, and interleaving their repairs gives better results than dealing with each type individually. In [13], it was shown that repairing can effectively help identify matches, and vice versa. It was one of the first few steps in that direction, but dealt only with CFDs (Conditional Functional Dependencies) [10, 14] and MDs. We propose the first data cleaning system capable of handling heterogeneous data quality rules, including user-defined ones.

1.4 Related Work

There has been an increasing amount of research about dependency theory (*a.k.a.* integrity constraints) [7] to specify data consistency, *e.g.*, FDs (Functional Dependencies) [11], its extension CFDs (Conditional Functional Dependencies) [10, 14], INDs (Inclusion Dependencies) [15], its extension CINDs (Conditional Inclusion Dependencies) [16], MDs (Matching Dependencies) [12] and Denial Constraints [8, 9]. However, limited expressiveness often does not allow to state problems commonly found in real life data as violations of these dependencies [7].

Different approaches for data repairing were proposed in literature, adopting different methods and techniques: Heuristic methods for FDs and INDs [15] or CFDs [10], graph-based data repairing [17, 18], user-provided confidence values [10, 15], user-guided repairs [19, 20] and statistical inference [19].

A number of studies have also been proposed to tackle different data quality rules in one framework. Fan et al. [13] studied the interaction between record matching (MDs) and data repairing (CFDs). Chu et al. [18] discussed holistically cleaning the data *w.r.t.* to a set of denial constraints. We propose the only system that deals with heterogeneous types of data quality rules including dependencies and custom business rules.

Moreover, because of data dynamism, several techniques have been proposed targeting continuous data cleaning. The underlying concept when encountering data changes is to avoid reprocessing data that has previously been processed. However, these techniques are also far from heterogeneity and only target specific types of data quality rules. For example, [10] discussed an incremental repairing algorithm, but only for CFDs. Recently, [21] proposed a continuous data cleaning framework for environments where the data and constraints are changing, but they also considered FDs only. SmartClean was introduced in [22] as an incremental data cleaning tool, but can only deal with some restricted data quality problems at the attribute level, *e.g.*, missing values and domain violations.

Furthermore, there has been an increasing interest in data cleaning explanations in the recent years. A recent study of explainability in data integration systems [23] shows that there have been some approaches towards explaining and explainable systems for tasks like schema matching, schema mapping, record linkage and data fusion. However, the state-of-the-art data cleaning solutions often target goals like consistency, accuracy and minimality, but do not provide useful explanations to the end-user about the repairing model or their decision process.

1.5 Limitations of Existing Solutions

Taking all of these approaches into consideration, we can clearly notice some limitations of the existing data cleaning solutions. We do not know about any other end-to-end data cleaning system that can handle a set of heterogeneous data quality rules. There is no unified language for rule definition, hence the different overwhelming syntax and semantics of miscellaneous rules, and most importantly the lack of support for ad-hoc or user-defined rules. Consequently, unaffordability comes next, as involving a data cleaning expert is not always a cheap option. Even if affordable, it is infeasible to build a separate algorithm for each different rule type from scratch. Moreover, existing solution offer no interaction among the various types of data quality rules, which greatly improves the data cleaning process, as we have shown in our motivating scenario.

1.6 Challenges and Objectives in Rule-Based Data Cleaning Systems

Comparing to state-of-the-art approaches and considering these limitations of existing solutions, when envisioning a modern data cleaning system, we need to tackle these challenges:

1.6.1 Heterogeneity and Interdependency

It needs to support heterogeneous types of data quality rules, *e.g.*, ETL (Extract, Transform and Load) rules, dependencies, and user-defined rules. It also has to interleave the repairs of these various types of rules for better cleaning results.

1.6.2 Extensibility

It can be extended by plugging different repairing algorithms to meet the user needs for effectiveness and efficiency.

1.6.3 Continuity

It should be able to efficiently handle data changes – insertions, deletions and updates – without re-processing the data from scratch.

1.6.4 Explainability

It has to provide user-friendly explanations for detected violations and chosen fixes, to explain the model as a whole and the individual data repairs.

1.6.5 Other Challenges

Other important challenges arise too, but are out of the scope of this dissertation.

Interactivity where the system provides a data quality dashboard to summarize the data, violations and repairs. This intersects with different research areas such as data profiling, violations visualization, and sampling/summarizing techniques.

Scalability is also an important aspect, where the system can scale to real world big data and clean the data in a distributed or parallel environment.

1.7 Research Contributions

We propose NADEEF¹, an end-to-end data cleaning system that is

Generalized: Covers a more general spectrum of data quality rules, including standard dependencies and user-defined rules.

Extensible: Users can plug-in their own data repairing algorithms.

Continuous: Can easily adapt to continuous data changes.

Explaining: Keeps the end-user in the loop through the whole process from start (defining rules) to finish (explaining violations and repairs).

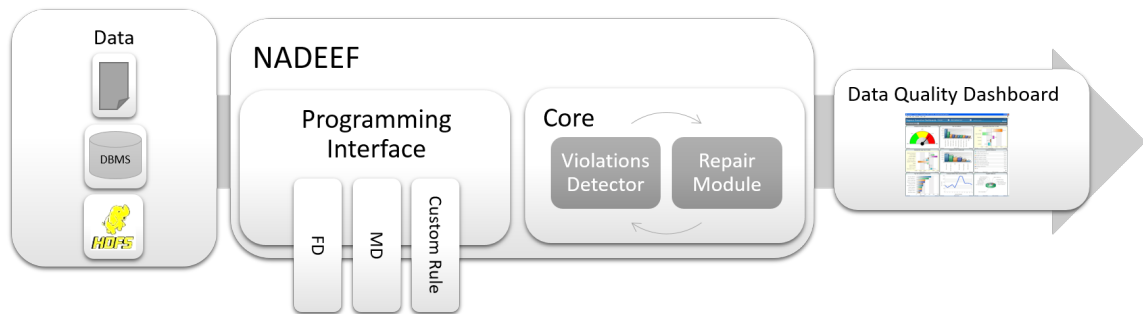


Figure 1.2. Overview of NADEEF

Figure 1.2 gives an overview of the system. Data comes from different sources, *e.g.*, a data file, a DBMS or a distributed file system. NADEEF provides a unified programming interface that allows the user to specify various types of data quality rules. Inside its core, the cleaning process is an iterative process of two phases: *violations detection* and *data repairing*. Finally, the output can be fed into a data quality dashboard, to show cleaned data, summarize the violations and repairs and bring the user into the loop.

¹<http://www.nadeef.info/>

Specifically, we can outline our contributions as follows:

Generalized and Extensible Data Cleaning [24, 25]: Despite 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 this dissertation, we present NADEEF, a generalized and extensible data cleaning platform that distinguishes between a *programming interface* and a *core*. 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. Treating user implemented interfaces as black-boxes, the *core* provides algorithms to detect the errors and clean the data holistically without differentiating between various types of rules. We showcase different implementations for repairing algorithms to 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.

Continuous Data Cleaning: Flawed data costing billions of losses every year urged the need for data cleaning systems and several approaches have been proposed since. However, with today’s data dynamism and velocity, new use cases appeared against which the current state-of-the-art approaches stand helpless, such as interactive data cleaning and cleaning of data streams, hence arises the necessity for continuous data cleaning. Existing frameworks are still limited to supporting only certain types of rules, suffer from errors propagation with data changes and offer no way to undo misinformed repairs. In this dissertation, we extend NADEEF to support these various use cases which traditional frameworks cannot deal with. We present the modified architecture of the system, explain how it can continuously detect violations and repair data and experiment the system against synthetic and real-world dynamic data.

Data Cleaning Explanations: With human involvement, data cleaning systems should provide explanations for their models. In this dissertation, we discuss properties of explanations, and highlight a recent classification of explainability in data integration systems. We discuss how NADEEF explains its process via providing detailed auditing information about detected violations and data repairs. We also discuss explanations beyond rule-based data cleaning and target entity resolution as a use case. We also propose a tool to understand and explain entity resolution classifiers with different granularity levels, and we demonstrate how it can handle different scenarios for a variety of classifiers on several benchmark datasets.

1.8 Dissertation Outline

Chapter 2 presents the structure of our end-to-end rule-based data cleaning system, and describes the system architecture. We define NADEEF’s programming interface and dive inside its core to cover the two phases of the data cleaning process, violations detection and data repairing.

Chapter 3 highlights the need for continuous data cleaning due to dynamic data and constant change. We explain how existing approaches cannot deal properly with data updates, and propose an extension to NADEEF to support continuity in data cleaning.

Chapter 4 focuses on the human involvement and on the need for explainability. It discusses explanations in NADEEF and explanations beyond rule-based data cleaning with entity resolution as a use case.

Finally, Chapter 5 concludes the dissertation by presenting our vision for how this line of research can be extended further to achieve the end-goal of higher data quality.

2 NADEEF: GENERALIZED AND EXTENSIBLE DATA CLEANING

2.1 An End-to-End Rule-Based Data Cleaning System

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* is wrong in the data. For instance, an FD will tell us that two tuples are inconsistent because of identical LHS but different RHS, but not how to fix this inconsistency.

2.1.1 Challenges

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. Fan et al. [13] show 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 [13, 15, 20], 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: 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 [20, 26, 27]. 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.

2.1.2 Contributions

We introduce NADEEF, a generalized and extensible data cleaning system that leverages 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.

In this work, we make several notable contributions:

- We describe the first end-to-end commodity data cleaning system that allows the users to specify multiple types of data quality rules defining *what* is wrong with the data and (possibly) *how* to repair it, and provides algorithms to detect the errors and clean the data holistically without differentiating between these various types of rules.
- We propose a novel programming interface 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.
- We discuss how violations detection works inside NADEEF. We also describe *partitioning* and *compression* to improve violations 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.
- To demonstrate the *extensibility* of our system, we present two core implementations for data cleaning algorithms. The first one, designed to achieve higher accuracy, employs a weighted MAX-SAT solver to compute repairs while minimizing their overall cost. The second one, designed to be more efficient, leverages the idea of equivalence classes [15]. It merges violating data into multiple equivalence classes and assigns a unique value to each equivalence class.
- We conduct extensive experiments to verify the generality, extensibility and effectiveness of our system using real-life datasets.

2.2 Related Work

The first serious discussions and analysis of data repairing have emerged in [28]. After that, a large and growing body of literature has investigated ETL tools (see [29] for a survey), which support data transformations, and can be employed to merge and repair data [30]. Recently, there has been an increasing amount of literature on dependency theory (*a.k.a.* integrity constraints) [7] to specify data consistency for data repairing, *e.g.*, FDs [11], CFDs [10, 14], INDs [15], CINDs [16], MDs [12] and Denial Constraints [8, 9]. 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 [7]. Through its programming interface, our framework is expressive enough to specify these standard constraints as well as custom business rules.

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 or duplicate detection. See [31] for a survey). In contrast to matching rules [12, 32, 33], our approach is more general since it also considers data repairing.

Several repairing algorithms were proposed over the past few decades [10, 15, 17–20, 27, 28]. Heuristic methods have been developed based on FDs and INDs [15], CFDs [10] and editing rules [27, 28]. Several graph-based heuristics have been also proposed for data repairing, *e.g.*, vertex-cover [17] and hyper-graphs [18]. The methods of [10, 15] employ confidence values provided by the users to guide a repairing process. Other approaches [19, 20] require consulting the users to ensure the accuracy of the generated repairs. Statistical inference is studied in [19] to derive missing values.

A few approaches have been proposed to tackle different data quality rules in one framework. The interaction between record matching (MDs) and data repairing

(CFDs) was discussed in [13]. AJAX [34] and TAILOR [35] are toolboxes for record linkage. Most recently, [18] proposed a hyper-graph-based solution to clean the data holistically *w.r.t.* a set of denial constraints.

In contrast, we propose a system that: (a) Covers a wider spectrum of data quality rules; and (b) Is extensible such that users can plug-in their own cores for error detection and data repairing.

2.3 Fundamentals

In this section, we present some of the notations and concepts needed before we introduce NADEEF’s architecture in more details.

Recall back our example from Section 1.3, shown again in Figure 2.1, and its different data quality rules:

1. An FD $\varphi_1 : \text{CC} \rightarrow \text{country}$.
2. An MD φ_2 stating that if two records from **transactions** and **customers** both match on first name, last name, street, city and country, then the transaction country code and phone in **transactions** should match those of the master data in **customers**.
3. A business rule φ_3 that would mark two transactions as either a fraud or erroneously recorded, if both were by the same person (identified by the first name, last name, country code, and phone) and in two different locations within 1 hour (considering different time zones).

We consider a database $\mathcal{D} = \{D_1, \dots, D_m\}$, where each D_j ($j \in [1, m]$) is an *instance* whose relation schema is R_j as $R_j = \{A_{j_1}, \dots, A_{j_n}\}$. 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 Figure 2.1, T is an instance of relation **transactions**, t_1 is a tuple from that instance and $t_1[\text{FN}]$ is a cell from that tuple whose value is David.

	FN	LN	street	city	country	CC	tel
c_1 :	David	Jordan	12 Holywell Street	Oxford	UK	44	66700543
c_2 :	Paul	Simon	5 Ratcliffe Terrace	Oxford	UK	44	44944631

(a) C : An instance of schema **customers**

	FN	LN	street	city	country	CC	phone	when	where
t_1 :	David	Jordan	12 Holywell Street	Oxford	UK	44	66700543	12:00 6/5/2012	Netherlands
t_2 :	Paul	Simon	5 Ratcliffe Terrace	Oxford	UK	44	44944631	11:00 2/12/2011	Netherlands
t_3 :	David	Jordan	12 Holywell Street	Oxford	Netherlands	44	66700541	6:00 6/5/2012	NY, USA
t_4 :	Peter	Austin	7 Market Street	Amsterdam	Netherlands	31	55384922	9:00 6/2/2012	Netherlands

(b) T : An instance of schema **transactions**

Figure 2.1. Database with violations of various heterogeneous rules

Data Quality Rules

In contrast to traditional ways of defining data quality rules by strictly following some declarative logical formalism, our programming interface (covered in details shortly) 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 it.

Violation: A *violation* V is a nonempty set of *cells* that is detected by 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 t_1 and t_3 shown in Figure 2.1 violate φ_1 , since they have the same CC value, but carry different **country** values. The corresponding violation consists of four cells $\{t_1[\text{CC}], t_1[\text{country}], t_3[\text{CC}], t_3[\text{country}]\}$.

For a database \mathcal{D} and a data quality rule φ , we denote by $\text{vio}(\mathcal{D}, \varphi)$ the set of all violations returned by φ . For a database \mathcal{D} and a set Σ of rules, we denote by $\text{vio}(\mathcal{D}, \Sigma) = \bigcup_{\varphi \in \Sigma} \text{vio}(\mathcal{D}, \varphi)$ the set of all violations for data \mathcal{D} and rules Σ .

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. For instance, the violation from previous example would have two candidate fixes, but each contains only one single expression in that case, either assigning $V.t_1[\text{country}]$ to $V.t_3[\text{country}]$ or $V.t_3[\text{country}]$ to $V.t_1[\text{country}]$; either way can resolve the violation V .

Cost Functions

As a database can be repaired in multiple ways, an immediate question is which 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 $\text{cost}(c, v_1, v_2)$ be the cost of changing the cell c from value v_1 to v_2 , and $\text{cost}(\mathcal{D}, \mathcal{D}_r)$ is defined as the sum of $\text{cost}(c, v_1, v_2)$ for each cell whose value is modified from v_1 in \mathcal{D} to v_2 in \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 no violations are detected by φ for all tuples 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 , $\text{fix}(V)$ returns an empty set of candidate fixes. We say \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 Figure 2.1. After all changes, we obtain a modified database \mathcal{D}' . While one violation still remains, *i.e.*, tuples (t_1, t_3) w.r.t. rule φ_3 , 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_3 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.*, $\text{detect}(\mathcal{D}'_r, \varphi)$ 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 [13] that given CFDs only, or 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 that case and for common rules such as CFDs and MDs, several algorithms have been proposed [12, 14] to check their consistency. 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 those algorithms. In this work, we assume collections of Σ and \mathcal{D} that are consistent.

2.4 System Architecture

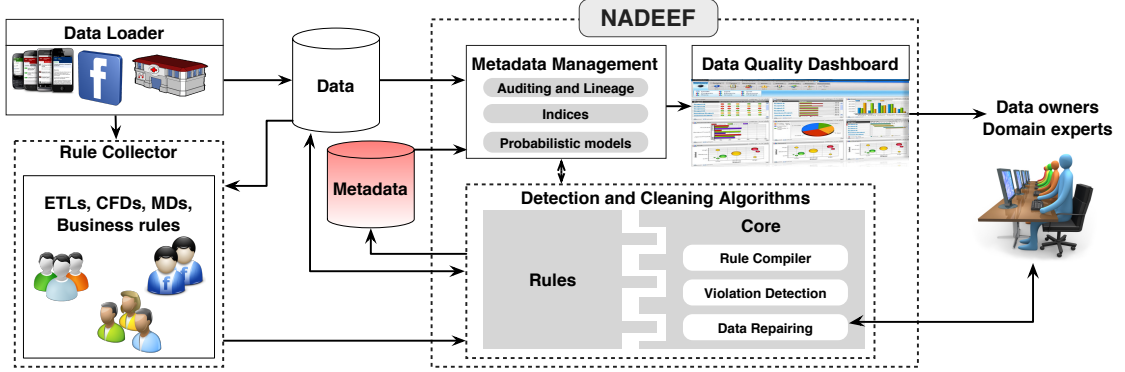


Figure 2.2. Architecture of NADEEF

We show NADEEF’s architecture in Figure 2.2. 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 violations 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. This metadata can be used to allow domain experts and users to actively interact with the system. It employs a data quality dashboard that would exploit this metadata and provide information such as error summarization and error distribution.

NADEEF contains these components: (1) the *Rule Collector* to gather user-specified quality rules; (2) the *Core* which encapsulates violations detection and data repairing for heterogeneous rules and allows for holistic data cleaning; and (3) the *Metadata Management* and *Data Quality Dashboard* modules that are concerned with maintaining and querying various metadata for data errors and their possible fixes. They also allow 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.

NADEEF Core

It contains components: *rule compiler*, *violations detection* and *data repairing*.

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

Violations Detection: This module takes the data and the compiled rules as input, and computes a set of data errors.

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) non-deterministic 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 [10,17]; the **Updater** applies the computed updates only if an attribute value is not changed more than x times. 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 understand and improve the cleaning process. 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 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.

2.5 NADEEF's Programming Interface

As a generalized data cleaning system, NADEEF introduces a simple programming interface through which the user can define heterogeneous types of data quality rules. Such a unified interface allows to treat all these rules as black-boxes when detecting violations and repairing the data accordingly. This also gives a great flexibility to define rules that represent business logic or custom rules. Moreover, NADEEF provides optimized implementations of some rules, *e.g.*, FDs and MDs, to cover the standard dependency-based widely used rules. Figure 2.3 shows the programming interface for defining the semantics of data errors and possible ways to fix them.

```

Interface Rule {
    Violations detect(Tuples tuples);
    Fixes fix(Violation violation);
}

```

Figure 2.3. NADEEF's Programming Interface

2.5.1 Programming Interface Operators

We define two main functions:

- **detect**: Takes a single tuple or a pair of tuples as input, and returns a set of problematic cells (violations), if any, in order to express the static semantics (*what* is wrong) of the rule.
- **fix**: Takes a nonempty set of problematic cells as input, and returns a set of suggested expressions to repair these data errors. It reflect the dynamic semantics (*how* to repair errors) of the rule. It suggests a set of candidate fixes for a violation, from which the repair module would later choose the “best” repairs to apply on data.

We refer to a class that inherits from *Rule* and implements the error detection function `detect()` as a *data quality rule*. We show different examples for various rule types in the following section.

The presence of a function `fix()` is optional, and its absence indicates that the users are not clear about its dynamic semantics, *e.g.*, φ_3 from our example.

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.*, FDs, variable CFDs, MDs and deduplication rules) or a set of tuples (*e.g.*, aggregation constraints [36]). 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 work. This is *especially* true for the efficiency of the violations detection process. 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.

```

Class Rule1 {
  Violations detect (Tuple  $t_1$ , Tuple  $t_2$ ) {
    if ( $t_1[CC] = t_2[CC] \wedge t_1[country] \neq t_2[country]$ )
      return { $t_1[CC, country], t_2[CC, country]$ };
    return  $\emptyset$ ;
  }
  Fixes fix (Violation  $v$ ) {
    Fixes fixes;
    fixes.insert( $v.t_1[country] \leftarrow v.t_2[country]$ );
    fixes.insert( $v.t_2[country] \leftarrow v.t_1[country]$ );
    return fixes;
  }
}

```

```

Class Rule2 {
  Violations detect (Tuple  $t$ , Tuple  $c$ ) {
    if ( $(t[LN, street, city, country] = c[LN, street, city, country] \wedge t[FN] \approx c[FN] \wedge t[CC, phone] \neq c[CC, tel])$ )
      return {  $t[FN, LN, street, city, country, CC, phone], c[FN, LN, street, city, country, CC, tel];$  }
    return  $\emptyset$ ;
  }
  Fixes fix (Violation  $V$ ) {
    return {  $V.t[CC] \leftarrow V.c[CC] \wedge V.t[phone] \leftarrow V.c[tel]$  }
  }
}

```

```

Class Rule3 {
  Violations detect (Tuple  $t_1$ , Tuple  $t_2$ ) {
    if ( $t_1[LN, CC, phone] = t_2[LN, CC, phone] \wedge t_1[FN] \approx t_2[FN]$ 
         $\wedge t_1[where] \neq t_2[where]$ 
         $\wedge time\_diff(t_1[when, where], t_2[when, where]) < 1$ )
      return {  $t_1[FN, LN, CC, phone, when, where],$ 
                $t_2[FN, LN, CC, phone, when, where];$  }
    return  $\emptyset$ ;
  }
}

```

Figure 2.4. Sample Rules in NADEEF

2.5.2 Sample Rules

Figure 2.4 illustrates how the rules in our example are represented via the unified programming interface. In a nutshell, the users specify a rule’s static semantics over either one tuple or two tuples via a **detect()** function, where an error is represented by a set of attribute values. Moreover, the users can also, if possible, specify its dynamic semantics by providing different alternatives to repair the data via a **fix()** function.

1. *Rule1* (for φ_1) specifies how to identify the violations via a **detect()** function on two tuples from table **transactions**. It also defines via a **fix()** function two options to resolve a violation V : Either update $V.t_2[\text{country}]$ to $V.t_1[\text{country}]$, or update $V.t_1[\text{country}]$ to $V.t_2[\text{country}]$.
2. *Rule2* (for φ_2) is defined on two tables **customers** and **transactions**. When it identifies that the same person from different tables has different phones (*i.e.*, the violation is not empty via a **detect()** function), it updates the **phone** value in **transactions** to the **tel** value from **customers** (in **fix()**). Actually, the user indicates that the values from **customers** are more reliable. Here, the \approx could be any (domain-specific) similarity function that a user defines.
3. *Rule3* (for φ_3) only specifies how to detect a violation via a **detect()** function. No **fix()** function is given since the users do not know how to resolve this violation.

By allowing multiple types of rules NADEEF provides a simple way to determine what is wrong via the **detect()** function and how to repair it via the **fix()** function.

2.5.3 Programming Interface Optimizations

Going back to our programming interface, we introduce a few other (optional) operators to help “efficiently” detect data quality errors, as shown in Figure 2.5. Should the user implement these methods, it would help speed up the detection phase. Note that these methods are already provided for pre-defined rules implemented in NADEEF, *e.g.*, FDs and MDs.

```

Interface Rule {
    Table scope(Table table);
    Tables block(Table table);
    Tuples iterator(Table block);

    Violations detect(Tuples tuples);
    Fixes fix(Violation violation);
}

```

Figure 2.5. NADEEF's Optimized Programming Interface

We define these functions:

- *scope*: Takes a database table as input, and returns a smaller table with only the relevant rows and columns needed to check for violations.
- *block*: Takes a table as input, and partitions it into smaller independent tables (blocks), such that tuples from different blocks will not encounter any violations. Tuples within the same partition only need to be compared together checking for possible violations.
- *iterator*: Takes a block as input, and defines the ordering in which its tuples should be passed to the **detect** method for a faster and more efficient traversal, rather than comparing all pairs of tuples within the block.

The optimized flow is given in Figure 2.6. For example (Refer to Figure 2.7), for the FD φ_1 from our example, the violations detector uses the rule methods, among other rules, as follows:

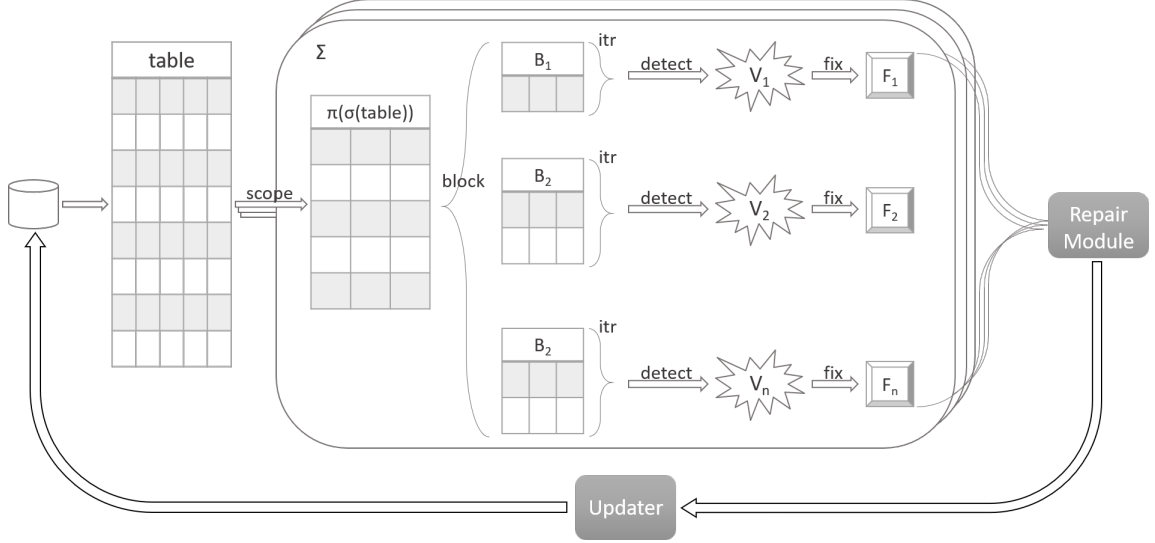


Figure 2.6. NADEEF's Violations Detection Flow for Optimized Rules

1. *transactions* data table is loaded from database
2. For φ_1 , among other rules, *scope* method is used to scope the table horizontally and vertically to only keep the relevant rows and columns to this rule, *i.e.*, CC and **country**
3. The table is then partitioned into smaller *blocks*, to avoid comparing all data tuples, but rather compare possibly violating tuples only. In this case, all tuples with the same LHS will be in the same block.
4. Inside every block, instead of comparing all pairs of tuples, an efficient *iterator* would sort the tuples on RHS and linearly scan the tuples searching for tuples with different **country** values.
5. As explained earlier, for every pair of tuples, *detect* method would produce a violation marking those tuples that share the same CC, but have different **country**, and *fix* method would suggest the corresponding candidate fixes.

	FN	LN	street	city	country	CC	phone	when	where
t_1 :	David	Jordan	12 Holywell Street	Oxford	UK	44	66700543	12:00 6/5/2012	Netherlands
t_2 :	Paul	Simon	5 Ratcliffe Terrace	Oxford	UK	44	44944631	11:00 2/12/2011	Netherlands
t_3 :	David	Jordan	12 Holywell Street	Oxford	Netherlands	44	66700541	6:00 6/5/2012	NY, USA
t_4 :	Peter	Austin	7 Market Street	Amsterdam	Netherlands	31	55384922	9:00 6/2/2012	Netherlands

(a) T : An instance of schema **transactions**

	country	CC
t_1 :	UK	44
t_2 :	UK	44
t_3 :	Netherlands	44
t_4 :	Netherlands	31

(b) *scope*

	country	CC		country	CC
t_1 :	UK	44	t_3 :	Netherlands	44
t_2 :	UK	44	t_1 :	UK	44
t_3 :	Netherlands	44	t_2 :	UK	44

	country	CC		country	CC
t_4 :	Netherlands	31	t_4 :	Netherlands	31

(c) *block*(d) *iterator*

Violations
$V_1 : \{t_1[\text{CC}, \text{country}], t_3[\text{CC}, \text{country}]\}$
$V_2 : \{t_2[\text{CC}, \text{country}], t_3[\text{CC}, \text{country}]\}$

(e) *detect*

Candidate Fixes
$F_1 : \{t_1[\text{country}] \leftarrow t_3[\text{country}]\}$
$F_2 : \{t_3[\text{country}] \leftarrow t_1[\text{country}]\}$
$F_3 : \{t_2[\text{country}] \leftarrow t_3[\text{country}]\}$
$F_4 : \{t_3[\text{country}] \leftarrow t_2[\text{country}]\}$

(f) *fix*Figure 2.7. Violations Detection Flow for φ_1 on *transactions*

6. Eventually, for all rules, all candidate fixes are fed into the repair module, in order for the repairing algorithm to choose the appropriate data repairs holistically, considering the data quality rules altogether at the same time.
7. Those repairs might introduce other violations to other rules, so the process is repeated until the data is consistent or the **Updater** decides to terminate it.

2.6 Inside NADEEF: System Core

2.6.1 Violations Detection

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

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 **GetViolations**, 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}_φ (resp. \mathcal{F}_φ) for each φ in Σ . A straightforward way to compute \mathcal{V} and \mathcal{F} for each φ , is to invoke the functions **detect**(t) and **detect**(t_1, t_2) for each single tuple and each pair of tuples on which φ is defined, respectively. The **fix**() function will be invoked for each violation returned by **detect**(t) or **detect**(t_1, t_2).

Considering our ongoing example from Figure 2.1, The detected violations and suggested candidate fixes are shown in Table 2.1. By applying φ_1 , attribute values from tuples t_1 and t_3 (resp. t_2 and t_3) violate φ_1 , *i.e.*, violation V_1 (resp. V_2), leading to two candidate fixes F_1 and F_2 (resp. F_3 and F_4). Next, φ_2 detects a violation V_3 between t_3 and c_1 and suggests F_5 as a candidate fix to copy the data from the more reliable master data. Finally, φ_3 marks the two transactions t_1 and t_3 as fraud or erroneous, but with no suggested fixes.

Table 2.1.
Sample Violations and Candidate Fixes

Rule	Violation	Candidate Fixes
φ_1	$V_1: \{t_1[\text{CC}, \text{country}], t_3[\text{CC}, \text{country}]\}$	$F_1: t_1[\text{country}] \leftarrow t_3[\text{country}]$ $F_2: t_3[\text{country}] \leftarrow t_1[\text{country}]$
φ_1	$V_2: \{t_2[\text{CC}, \text{country}], t_3[\text{CC}, \text{country}]\}$	$F_3: t_2[\text{country}] \leftarrow t_3[\text{country}]$ $F_4: t_3[\text{country}] \leftarrow t_2[\text{country}]$
φ_2	$V_3: \{t_3[\text{FN}, \text{LN}, \text{street}, \text{city}, \text{country}, \text{CC}, \text{phone}],$ $c_1[\text{FN}, \text{LN}, \text{street}, \text{city}, \text{country}, \text{CC}, \text{tel}]\}$	$F_5: t_3[\text{CC}] \leftarrow c_1[\text{CC}]$ $\wedge t_3[\text{phone}] \leftarrow c_1[\text{tel}]$
φ_3	$V_4: \{t_1[\text{FN}, \text{LN}, \text{CC}, \text{phone}, \text{when}, \text{where}],$ $t_3[\text{FN}, \text{LN}, \text{CC}, \text{phone}, \text{when}, \text{where}]\}$	\emptyset

Note that resolving the violations of φ_1 introduced the violation of φ_2 , whose fix in turn led to another violation of φ_3 . That is why NADEEF performs detection and repairing in an iterative manner until the **Updater** terminates the process.

Optimizations for Violations Detection

Usually, systems that are general and extensible have to trade in performance for generality and extensibility. This is also the case of NADEEF. We hereby discuss some optimization techniques that are possible in more restricted settings.

For instance, violation detection upon all pairs of tuples using the **detect**(t_1, t_2) 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* (Recall the *block* operator in our optimized programming interface), 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.*, φ_1 in our example), we may be able to divide them using the CC attribute.

Thus, violations of tuples with the same CC but different **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 . We refer to this as violations *compression*.

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. Such knowledge can be passed through the different operators of our programming interface (*scope*, *block*, *iterator*, *detect* and *fix*) and is already provided for several pre-defined rules that are implemented and employed in NADEEF, *e.g.*, FDs, CFDs and MDs.

We highlight two functions $\text{LHS}(\varphi)$ and $\text{RHS}(\varphi)$ to return the set of attributes for the above (a) and (b), respectively. For example, consider *Rule1* (φ_1) in our ongoing example, we have $\text{LHS}(\varphi_1) = \{\text{CC}\}$ and $\text{RHS}(\varphi_1) = \{\text{country}\}$.

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 [37, 38] and can be leveraged within NADEEF.

Single data quality rule

For a single data quality rule φ , *partitioning* groups all tuples whose $\text{LHS}(\varphi)$ 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 $t_1[\text{LHS}(\varphi) \cup \text{RHS}(\varphi)] = t_2[\text{LHS}(\varphi) \cup \text{RHS}(\varphi)]$. When they are merged, their cell values related to attributes $\text{LHS}(\varphi) \cup \text{RHS}(\varphi)$ are the same and will be compressed as $t_{12}[\text{LHS}(\varphi) \cup$

$\text{RHS}(\varphi)] = t_1[\text{LHS}(\varphi) \cup \text{RHS}(\varphi)]$. We use the term *super cell* for a compressed cell. A super cell c is a cell with a set of identifiers of original cells, referred to as the extension of cell c , denoted by $\text{ext}(c)$. For the attributes that are irrelevant to φ , their cell values in t_{12} are set to null, and their extensions are empty. It is worth noting that when similarity comparisons are considered, two cells whose values are similar cannot be compressed, if the similarity function is not transitive.

Reconsider the database in Figure 2.1 and the rule *Rule1* (φ_1) in Figure 2.4. Figure 2.8 illustrates how the partitioning and compression techniques work. Here, $\text{LHS}(\varphi_1) = \{\text{CC}\}$ and $\text{RHS}(\varphi_1) = \{\text{country}\}$.

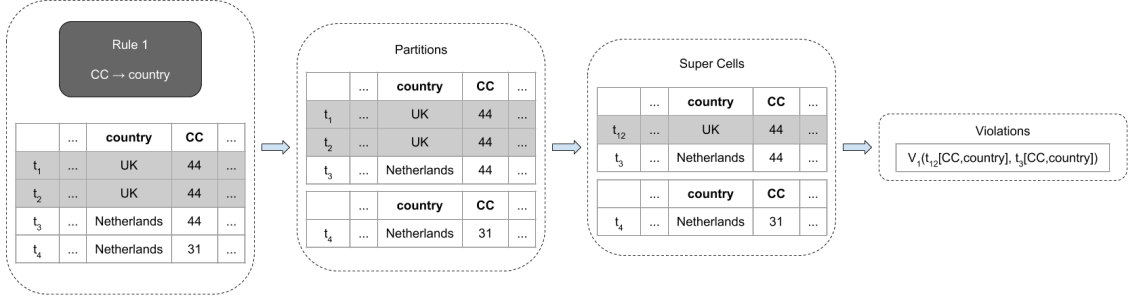


Figure 2.8. Partitioning and Compression for φ_1

Partitioning table transactions gives two partitions: *Partition1* with three tuples t_1 – t_3 since their CC values are the same (44), and *Partition2* with one tuple t_4 .

For compression, since $t_1[\text{CC}, \text{country}] = t_2[\text{CC}, \text{country}]$, the two tuples t_1 and t_2 (shaded tuples) will be merged into t_{12} . Their cells $t_1[\text{CC}]$ and $t_2[\text{CC}]$ (resp. $t_1[\text{country}]$ and $t_2[\text{country}]$) are merged into a super cell cc_{12} (resp. $country_{12}$) in t_{12} whose extension is $\text{ext}(cc_{12}) = \{t_1[\text{CC}], t_2[\text{CC}]\}$ (resp. $\text{ext}(country_{12}) = \{t_1[\text{country}], t_2[\text{country}]\}$). The other cells of t_{12} , which are irrelevant to φ_1 , have empty extensions and are omitted here. The other tuples (t_3 and t_4) are not compressed as shown in Figure 2.8.

From *Partition1* and after compression (tuples t_{12} and t_3), one violation with four cells, *i.e.*, $\{cc_{12}, country_{12}, cc_3, country_3\}$, is detected. Note that this violation was originally represented by two violations had we not applied compression, and that there are no violations from *Partition2*.

Multiple data quality rules

Given a database \mathcal{D} and a set Σ of data quality rules as input, NADEEF computes and returns a set \mathcal{V} of all violations.

It first creates partitions for each data quality rule. Then, it compresses tuples for each partition. The violations for each rule will be computed within its partitions. After computing all violations, the system needs to expand violations that have intersections across different rules before the violations are then returned.

Notably, a super cell is originally related to one rule, 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. We illustrate how the expansion works with the example below.

For table **transactions** (Figure 2.1), assume that there another rule φ_4 that causes another violation V_b as shown in Figure 2.9. The violation V_a in Figure 2.9 is the one derived in Figure 2.8. Note that the two cells, $country_{12}$ in V_a and $country_{11}$ in V_b are not the same, but overlap on $t_1[\text{country}]$. Hence, violation V_a needs to be expanded to V_{a1} and V_{a2} , as depicted in Figure 2.9.

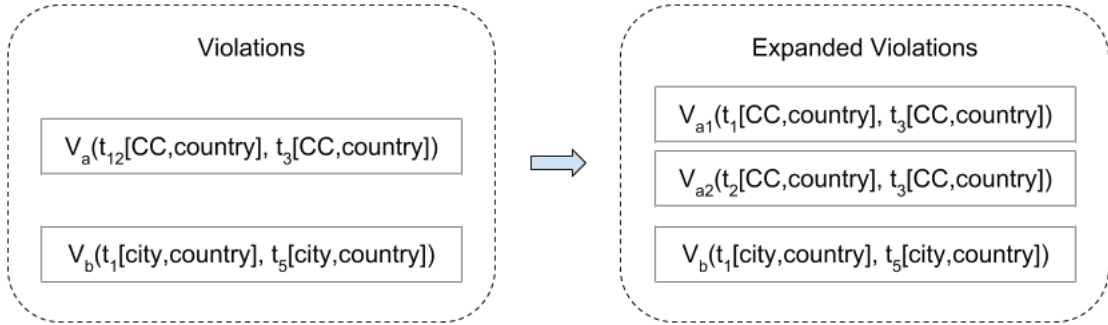


Figure 2.9. Violations Expansion

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

Performance Gain

Considering the worst case complexity of brute force method, Table 2.2 shows the benefits of partitioning and compression using a table of 10K tuples of HOSP dataset (More about datasets in Section 2.7). Without any optimization, it requires more than 1 billion pairwise comparisons and produces 130K violations. With partitioning and compression, it requires only 4.4K pairwise comparisons and produces only 4.4K violations.

Table 2.2.
Benefits of Partitioning and Compression

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

2.6.2 Data Repairing

In this section, we describe two algorithms for the data repairing module (implemented in NADEEF), referred to as **GetFixes**. 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 solutions. 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.

Variable-Weighted MAX-SAT Solver Repair Module

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 $\text{cost}(\mathcal{D}, \mathcal{D}')$ 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 [39] 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 $x_{t[A]}^a$. We denote by $\text{wt}(x_{t[A]}^a)$ the weight of $x_{t[A]}^a$. Intuitively, a variable $x_{t[A]}^a$ set to *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 $\text{wt}(x_{t[A]}^a)$. Naturally, if $t[A] = a$, we have $\text{wt}(x_{t[A]}^a) = 0$. Note that the compression technique discussed previously 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.*, $|\text{ext}(c)|$.

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 $\text{val}(\mathcal{F}, t[A])$ 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_2 in Table 2.1, we have $\text{val}(\{F_2\}, t_3[\text{country}]) = \{\text{Netherlands}, \text{UK}\}$.

(a) Inclusive Assignments: For each cell $t[A]$ such that $n > 1$ where $n = |\text{val}(\mathcal{F}, t[A])|$, we generate a clause with the form $(x_{t[A]}^{a_1} \vee \dots \vee x_{t[A]}^{a_n})$. 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 $n > 1$ where $n = |\text{val}(\mathcal{F}, t[A])|$, we generate $n(n-1)/2$ clauses, where each clause is in the form of $(\neg x_{t[A]}^{a_i} \vee \neg x_{t[A]}^{a_j})$, for each $a_i, a_j \in \text{val}(\mathcal{F}, t[A])$, and $a_i \neq a_j$. Intuitively, these clauses assure that at most one of the values can be assigned to $t[A]$.

(c) Violations 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 value of $t_i[A_i]$, we generate a clause $(\neg x_{t_1[A_1]}^{a_1} \vee \dots \vee \neg x_{t_n[A_n]}^{a_n})$. Intuitively, this clause assures these values cannot all be true simultaneously to avoid the violation when put together.

Recall that the cost of each variable is determined by a function $\text{cost}(c, v_1, v_2)$ 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.*, update unit) when $v_1 \neq v_2$ and 0 otherwise.

To better understand how the variable-weighted MAX-SAT solver works, we consider slightly modified versions of our data quality rules, so that the violations and their candidate fixes are encountered together in the same iteration:

1. An FD $\varphi_1 : \text{CC} \rightarrow \text{country}$.
2. An MD φ'_2 : If two records match on first name and last name, then the transaction country code and phone in **transactions** should match those of **customers**.
3. A business rule φ'_3 that would mark two transactions as fraud or erroneous if both were by the same person (identified by the first name and last name) in two different locations within 1 hour.

Variables with weight 1 are shown in Table 2.3, *e.g.*, the variable $x_{t_3[\text{country}]}^{UK}$ indicates that $t_3[\text{country}]$ can be changed to UK. Variables with weight 0 are omitted.

Table 2.3.
MAX-SAT Solver Weighted Variables

$x_{t_1[\text{country}]}^{Netherlands}$	$x_{t_2[\text{country}]}^{Netherlands}$	$x_{t_3[\text{country}]}^{UK}$	$x_{t_3[\text{phone}]}^{66700543}$
---	---	--------------------------------	------------------------------------

We generate four *inclusive* (resp. *exclusive*) clauses, corresponding to the above four non-zero weight variables, as shown in Table 2.4.

Table 2.4.
Inclusive and Exclusive Assignments for MAX-SAT Solver

Inclusive Assignments	Exclusive Assignments
$(x_{t_1[\text{country}]}^{UK} \vee x_{t_1[\text{country}]}^{Netherlands})$	$(\neg x_{t_1[\text{country}]}^{UK} \vee \neg x_{t_1[\text{country}]}^{Netherlands})$
$(x_{t_2[\text{country}]}^{UK} \vee x_{t_2[\text{country}]}^{Netherlands})$	$(\neg x_{t_2[\text{country}]}^{UK} \vee \neg x_{t_2[\text{country}]}^{Netherlands})$
$(x_{t_3[\text{country}]}^{UK} \vee x_{t_3[\text{country}]}^{Netherlands})$	$(\neg x_{t_3[\text{country}]}^{UK} \vee \neg x_{t_3[\text{country}]}^{Netherlands})$
$(x_{t_3[\text{phone}]}^{66700541} \vee x_{t_3[\text{phone}]}^{66700543})$	$(\neg x_{t_3[\text{phone}]}^{66700541} \vee \neg x_{t_3[\text{phone}]}^{66700543})$

Moreover, four clauses are generated to guarantee avoidance of the existing four violations, as shown in Table 2.5.

Table 2.5.
Violations Avoidance for MAX-SAT Solver

Violations Avoidance
$(\neg x_{t_1[\text{CC}]}^{44} \vee \neg x_{t_1[\text{country}]}^{UK} \vee \neg x_{t_3[\text{CC}]}^{44} \vee \neg x_{t_3[\text{country}]}^{Netherlands})$
$(\neg x_{t_2[\text{CC}]}^{44} \vee \neg x_{t_2[\text{country}]}^{UK} \vee \neg x_{t_3[\text{CC}]}^{44} \vee \neg x_{t_3[\text{country}]}^{Netherlands})$
$(\neg x_{t_3[\text{FN}]}^{David} \vee \neg x_{t_3[\text{LN}]}^{Jordan} \vee \neg x_{t_3[\text{CC}]}^{44} \vee \neg x_{t_3[\text{phone}]}^{66700541} \vee \neg x_{c_1[\text{FN}]}^{David} \vee \neg x_{c_1[\text{LN}]}^{Jordan} \vee \neg x_{c_1[\text{CC}]}^{44} \vee \neg x_{c_1[\text{tel}]}^{66700543})$
$(\neg x_{t_1[\text{FN}]}^{David} \vee \neg x_{t_1[\text{LN}]}^{Jordan} \vee \neg x_{t_1[\text{when}]}^{12:00-6/5/2012} \vee \neg x_{t_1[\text{where}]}^{Netherlands} \vee \neg x_{t_3[\text{FN}]}^{David} \vee \neg x_{t_3[\text{LN}]}^{Jordan} \vee \neg x_{t_3[\text{when}]}^{6:00-6/5/2012} \vee \neg x_{t_3[\text{where}]}^{NY,USA})$

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 3 and 2, respectively, as in Table 2.6.

Table 2.6.
Possible Repairs of MAX-SAT Solver

Cost	Truth Assignments of non-zero Weighted Variables
3	$x_{t_1[\text{country}]}^{Netherlands}$, $x_{t_2[\text{country}]}^{Netherlands}$, $x_{t_3[\text{phone}]}^{66700543}$
2	$x_{t_3[\text{country}]}^{UK}$, $x_{t_3[\text{phone}]}^{66700543}$

The variable-weighted MAX-SAT solver will choose the second option above, which will update two cells, *i.e.*, changing $t_3[\text{country}]$ to UK and $t_3[\text{phone}]$ to 66700543, since it has the lowest total cost of 2.

Equivalence Classes Repair Module

An alternative implementation of our core algorithm for data repairing is based on equivalence classes. It was originally introduced for FD and CFD repairs [10], but we extend it for our generalized system to support other rules. We hereby revise the concept and 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 $\text{eq}(c)$. An equivalence class E is associated with a set of candidate values, denoted by $\text{cand}(E)$, and a unique *target* value, denoted by $\text{targ}(E)$.

Reconsider the violations and candidate fixes of φ_1 in Table 2.1. 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 (See Figure 2.10):

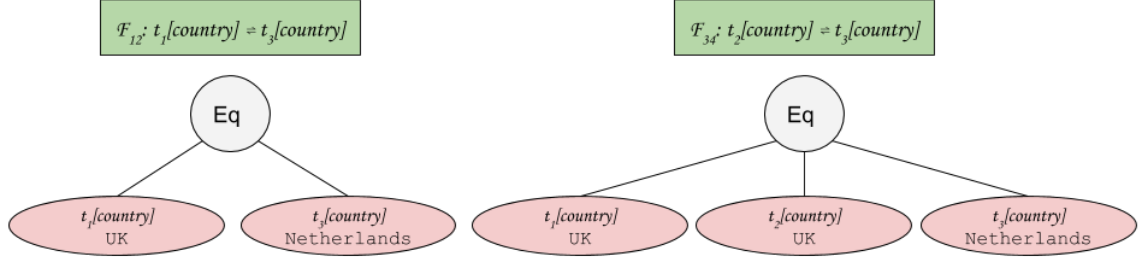


Figure 2.10. Equivalence Classes Repair Module

1. **Initialization:** Each cell c involved in \mathcal{V} is an equivalence class $\text{eq}(c)$, and its candidate value is its current cell value.
2. **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 $\text{eq}(c_1)$ and $\text{eq}(c_2)$ will be merged into one, and the new set of candidate values is the union of the two sets of candidate values for $\text{eq}(c_1)$ and $\text{eq}(c_2)$, *i.e.*, $\text{cand}(\text{eq}(c_1)) \cup \text{cand}(\text{eq}(c_2))$.
 - (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 $\text{eq}(c_1)$ will become $\text{cand}(\text{eq}(c_1)) \cup \text{cand}(\text{eq}(c_2))$.
3. **Assign a target value:** For each equivalence class E , select one target value $\text{targ}(E)$ from its candidate values $\text{cand}(E)$, such that the total cost of changing all cell values in E to $\text{targ}(E)$ is minimum.

Using equivalence classes, we separate the decision of which cell values should be the same from that of what target value should be assigned to an equivalence class. We defer the assignment of $\text{targ}(E)$ as late as possible to reduce poor local decisions.

Note again that the compression technique discussed previously can be readily applied here. The cost of making a super cell value change in an equivalence class is multiplied by $|\text{ext}(c)|$, the cardinality of the super cell.

2.7 Experimental Study

We evaluated our data cleaning system NADEEF along four dimensions:

Generality: The programming interface can specify multiple types of rules.

Extensibility: Different core algorithms can be used to detect errors and clean data.

Effectiveness: Our system can find a *fixed* database with high accuracy.

Efficiency: Our system can work on data in reasonable sizes.

2.7.1 Experimental Settings

Datasets

We used two real-life datasets:

1. HOSP *data* was taken from the US Department of Health & Human Services ¹. 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 ² that 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 from the UK government public datasets ³. 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), Creation 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.

¹<http://www.hospitalcompare.hhs.gov>

²<http://databases.about.com/od/access/a/zipcodedatabase.htm>

³<http://data.gov.uk/data>

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 `noise%`. 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 (as detailed shortly).

Algorithms

NADEEF was implemented initially in C++, including the following algorithms:

1. The algorithm for violation detection, using partitioning and compression techniques by default, referred to as `GetViolations`.
2. The algorithm for computing fixes using the variable-weighted SAT-solver, referred to as `WSAT`. We used a variable-weighted MAX-SAT solver from [40].
3. The algorithm for computing fixes using equivalence classes, referred to as `EQU`.

For comparison, we obtained the implementation of two algorithms for FD repairing, a cost-based heuristic method [15], referred to as `HEU`, and a vertex cover based approach [17], referred to as `VER`. Both approaches were implemented in Java.

We conducted all experiments on a Microsoft Windows machine with a 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.

2.7.2 Generality

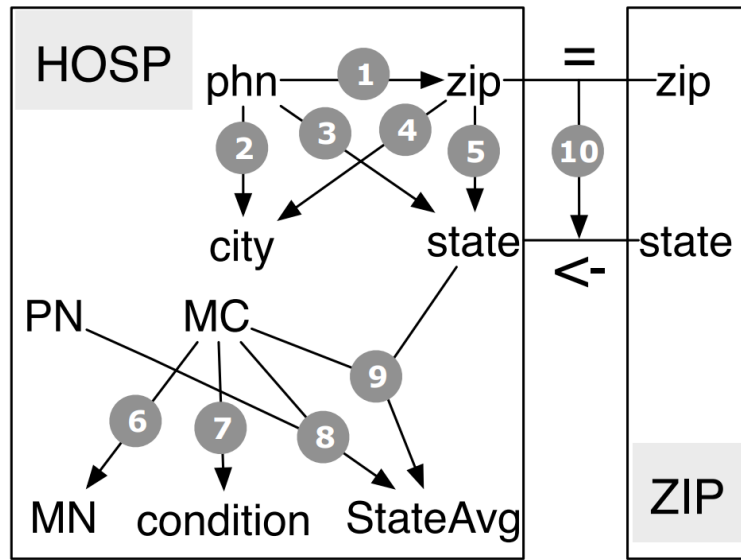
We show the *generality* of our framework by demonstrating that it can be used to specify multiple types of data quality rules as shown in Figure 2.11.

We define 10 rules over the HOSP dataset, where rules 1-9 are all FDs and rule10 is an MD. For example, rule9 states that in table HOSP, **MC** and **state** determines **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.

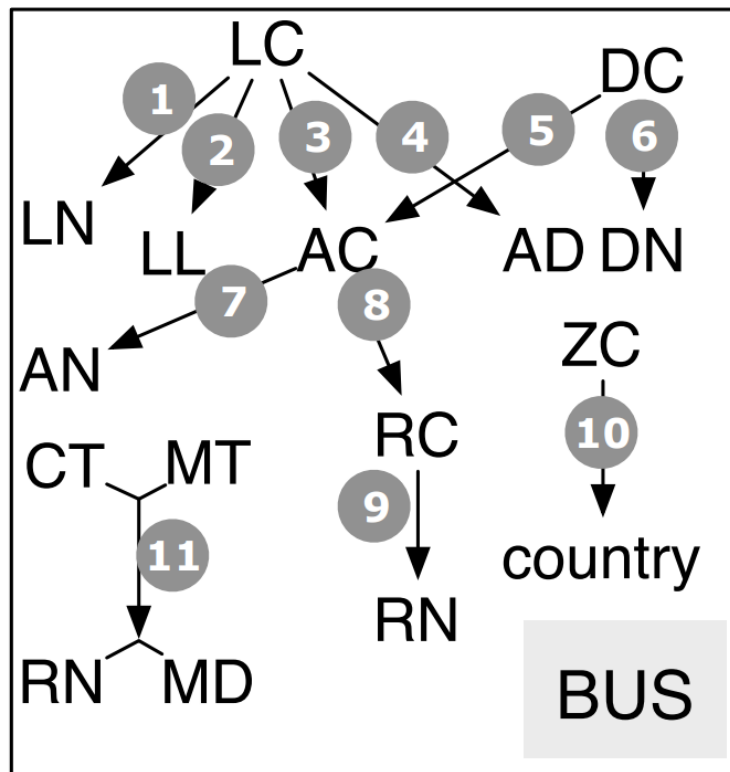
There are 11 rules defined over the BUS dataset, where rules 1-10 are FDs, and rule11 is a customized rule. Rule11, a single tuple rule, states that for each tuple, if its Creation DateTime (CT) and Modification DateTime (MT) are the same, then its Revision Number (RN) must be 0, and its status Modification (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.

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.

Remark: To ease the use of NADEEF, we have implemented some common classes that follow our optimized programming interface in Figure 2.5 and can be easily reused by users, *e.g.*, FDs, CFDs and MDs. For example, to specify various FDs, users only need to specify the LHS and RHS attributes of each FD. We have also implemented some common similarity functions, with string edit distance employed by default.



(a) Rules for HOSP Dataset



(b) Rules for BUS Dataset

Figure 2.11. Data Quality Rules for Datasets

2.7.3 Extensibility

We have already mentioned in Section 2.6 that we provide two different algorithms for fixing errors (*i.e.*, `GetFixes`), namely `WSAT` (Weighted MAX-SAT Solver Repair Module) and `EQU` (Equivalence Classes Repair Module). A user can specify through a configuration file which repair module to use. Users can also specify a totally different core algorithm `GetFixes` as long as that algorithm can take as input sets of violations and candidate fixes, and returns a set of fixes to be applied to the database. Moreover, not only `GetFixes`, but also the algorithm for `GetViolations` can be overridden.

2.7.4 Effectiveness

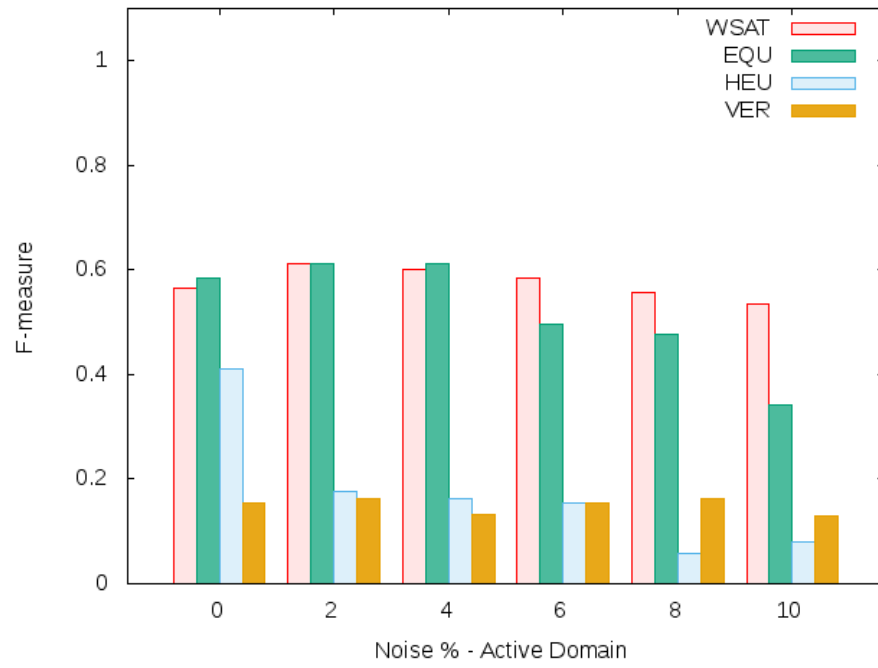
We evaluate the accuracy of different cleaning algorithms by varying noise rate (`noise%`) and the size of data. Noise is added by either introducing typos to an attribute value or changing an attribute value by another one from the active domain of that specific attribute.

Noise from the active domain:

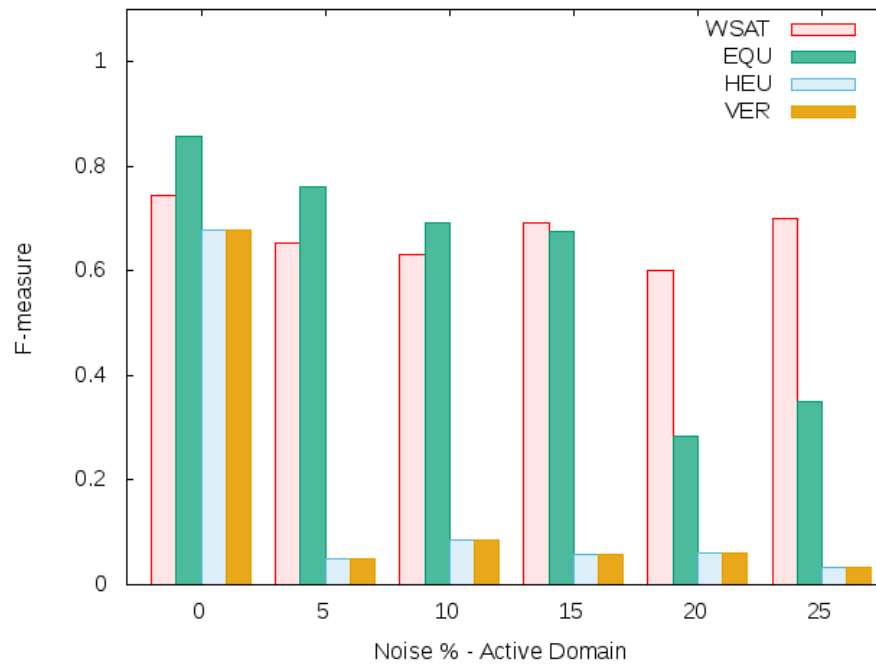
In this series of experiments, we fix the overall noise rate to 1 percent, and then vary the noise rate from the active domain (x -axis in both charts in Figure 2.12). The results of comparing `WSAT`, `EQU`, `HEU` and `VER` are given in the figure (the y -axis represent **F-measure** values), where Figure 2.12(a) is for HOSP data and Figure 2.12(b) is for BUS data. We use 10K tuples for both datasets with FDs only in order to compare against `HEU` and `VER` that only support FDs.

Figure 2.12 illustrates 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 in Figures 2.12(a) and 2.12(b).

However, when there is noise from active domain, the **F-measure** values of both `EQU` and `HEU` drop quickly. Thus, while `EQU` and `HEU` are sensitive to noise from the active domain, `WSAT` is not.



(a) HOSP Dataset



(b) BUS Dataset

Figure 2.12. F-Measure with noise from active domain

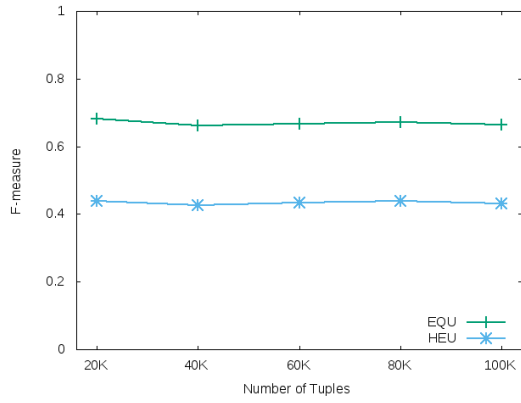
We explain the above results as follows:

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 a target value for an equivalence class, 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 [17]) 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.

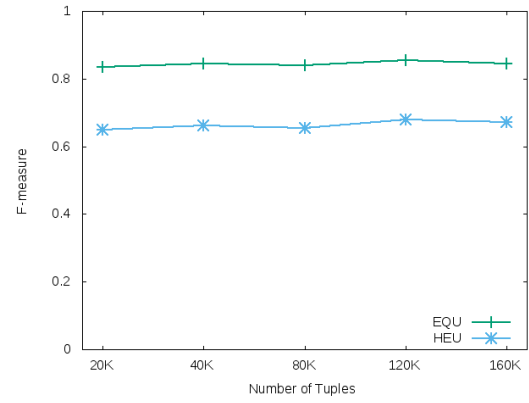
From this group of experiments, we can observe the following:

1. The new problem studied in this work 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 are efficient enough to cover a large part of existing data cleaning algorithms, are sensitive to the noise from the active domain.

In the following, to favor the other approaches, we only consider noise from typos. Since the algorithms **WSAT** and **VER** cannot scale well, we focus our comparisons mainly on algorithms **EQU** and **HEU**.

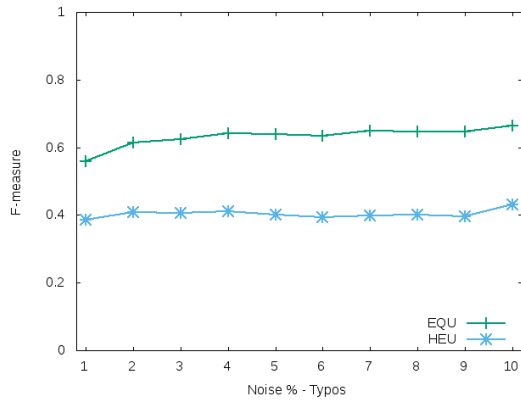


(a) HOSP Dataset

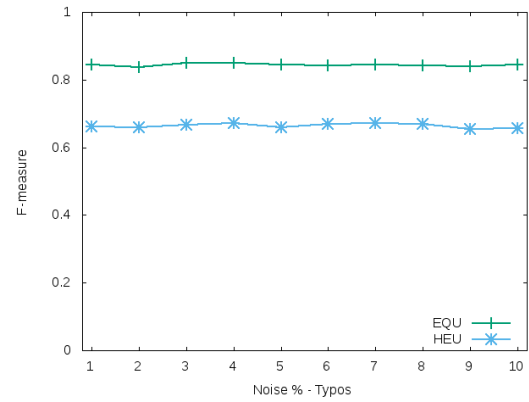


(b) BUS Dataset

Figure 2.13. F-Measure with variable data size



(a) HOSP Dataset



(b) BUS Dataset

Figure 2.14. F-Measure with variable noise rate

Noise from typos only:

We evaluate the effectiveness of different algorithms when there is only noise from typos. Figure 2.13(a) (resp. Figure 2.13(b)) shows the case for HOSP (resp. BUS) dataset when fixing `noise%` at 1% while varying the size of the data from 20K to 100K tuples (resp. 20K to 160K tuples). Moreover, Figure 2.14(a) (resp. Figure 2.14(b)) shows the results for HOSP (resp. BUS) dataset when varying the noise rate `noise%` 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 2.13(a) and 2.13(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 2.14(a) and 2.14(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 Figure 2.12.

Interleaving various types of data quality rules:

We evaluate the effect of executing various types of data quality rules together versus executing them sequentially. Recall from Figure 2.11 that for HOSP dataset, the MD rule10 overlaps with other FD rules, but 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. For BUS, since there is no overlap of rules, executing them in any order makes no difference.

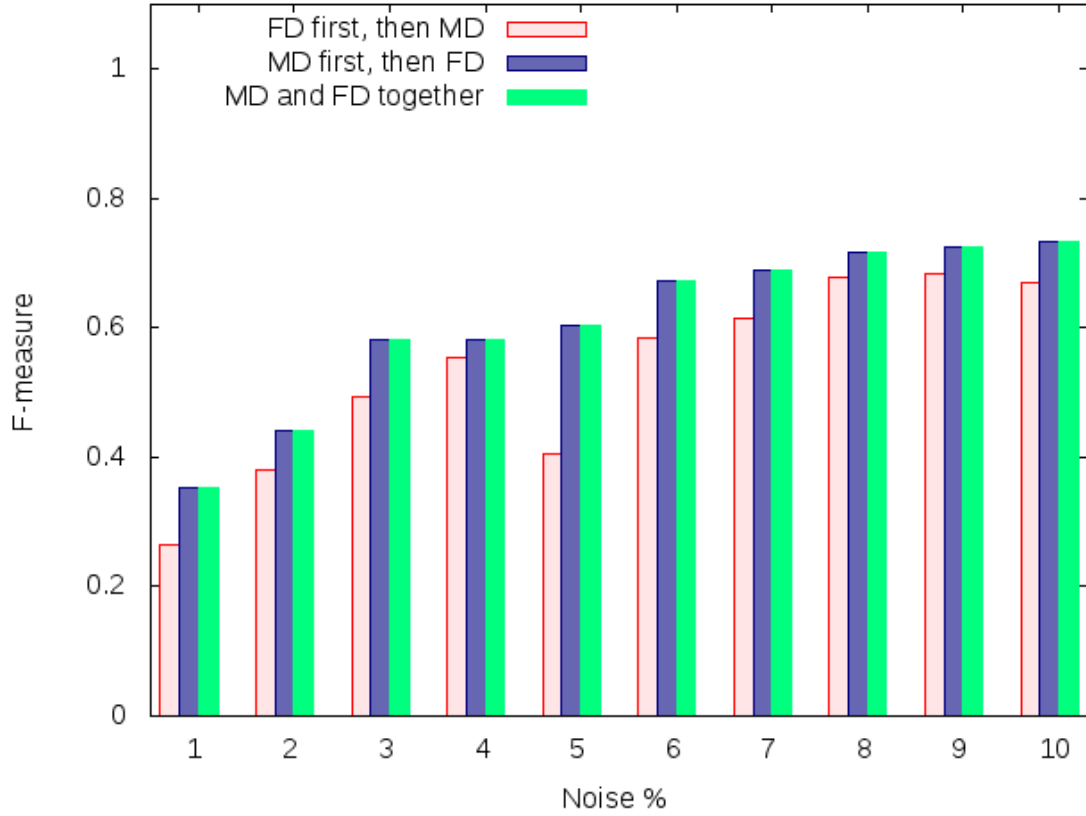


Figure 2.15. Interleaving various types of rules

We run Algorithm HEU on the HOSP data with 100K tuples, varying the noise rate (noise%) from 1% to 10%. To clearly show the result, we add 5% extra noise to the attributes that are related to the intersected rules, *i.e.*, attributes **zip** and **state**. The result is given in Figure 2.15. 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.

From this experiment, observe 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 the experiment in Figure 2.15.

2.7.5 Efficiency

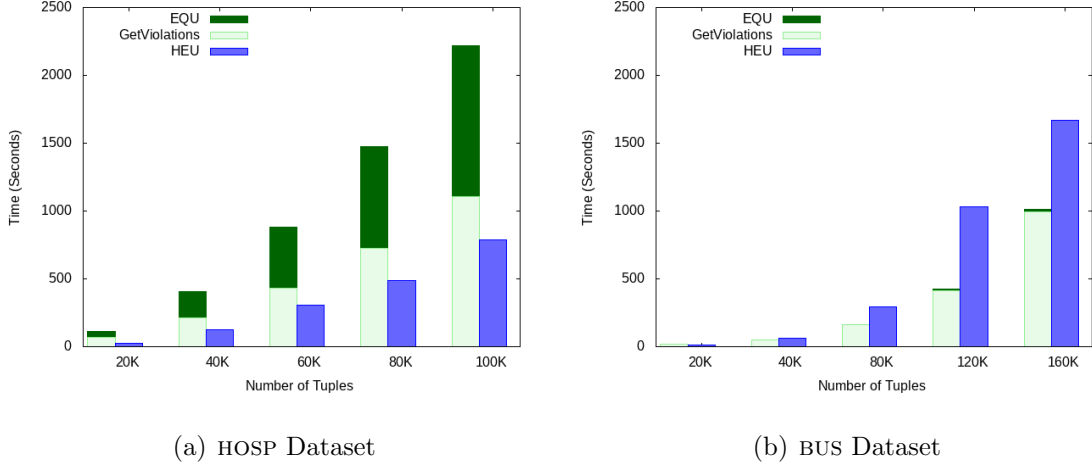


Figure 2.16. Running time with variable data size

In the last group of experiments, we study the efficiency of various algorithms. We start by comparing EQU and HEU running times, and then evaluate **WSAT** efficiency on both datasets.

Figure 2.16 shows running times of EQU and HEU: Figure 2.16(a) relates to Figure 2.13(a), and Figure 2.16(b) relates to Figure 2.13(b). In these two figures, the x -axis represents the number of tuples while the y -axis represents the running time. We show two components of the running time of our method, where the lower part is for detecting violations (*i.e.*, **GetViolations**) 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:

1. EQU and HEU deliver good results but for different applications, *e.g.*, HEU is faster for HOSP (Figure 2.16(a)) while EQU is faster for BUS (Fig. 2.16(b)).
2. While **GetViolations** takes some time, the EQU repair module for computing fixes is quite efficient, which proves the benefits of our partitioning and compression techniques (Recall the results from Table 2.2).

Table 2.7.
Running Times for WSAT

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

Moreover, we study the efficiency of WSAT on Datasets HOSP and BUS, where we fix `noise%` at 1% while varying the size of data from 20K to 100K for HOSP and 20K to 160K for BUS. The running times are given in Table 2.7. The running times above 2 hours are not given because they do 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 repair module, *e.g.*, WSAT or EQU, to invoke for each group.

2.7.6 Summary

From the above experimental study, we conclude that:

1. Providing a *generalized* programming interface, which requires minimum user efforts to specify heterogeneous data quality rules, is practically needed and is possible (see Section 2.7.2).
2. 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 `GetViolations` and `GetFixes` (see Section 2.7.3).

3. Our system, NADEEF, can achieve better accuracy than existing methods since the users provide the system with the dynamic semantics to resolve violations (see Section 2.7.4).
4. Multiple types of data quality rules should be treated *holistically*, since in practice, it is difficult to know *a priori* the best order of sequential execution for algorithms designed for different rules (see Section 2.7.4).
5. NADEEF, can work for data in reasonable sizes (see Section 2.7.5).

2.8 Concluding Remarks

We present NADEEF, a commodity data cleaning system. The main design concept of NADEEF is to separate a *programming interface* that allows users to flexibly define multiple types of data quality rules and provide 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 the system *generality* and that our interface is 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.

The system has been open-sourced ⁴ and is available as a live demo ⁵ with a more user friendly interface (*i.e.*, GUI) to help the users define their rules easier and a data quality dashboard to summarize the data, violations and repairs. It is the first approach towards an end-to-end generalized data cleaning system, and has since been highly cited with a notable impact on the data cleaning and data quality research communities.

⁴<https://github.com/daqcri/NADEEF>

⁵<http://nadeef.da.qcridemos.org/>

3 CONTINUOUS DATA CLEANING

3.1 Dynamic Data and the Need for Continuous Data Cleaning

We have already discussed how vital high quality data can be for any business and noted that low quality data is costing trillions of losses every year. This shows how much important, yet challenging, realizing high quality data is and highlights the urging need for data cleaning systems; a need that easily justifies why the market for data cleaning systems is growing much larger than other IT segments [4].

Data Cleaning has been studied for several years, and various approaches and systems have been proposed. Some embraced ETL tools [30], while some dealt with dependencies [7] and integrity constraints [10–12, 14, 15, 20]. Other approaches targeted significant problems particularly, such as Data Deduplication [31].

With today’s data dynamism and velocity, new use cases appeared against which the current state-of-the-art approaches stand helpless, such as interactive data cleaning and cleaning of dynamic data, hence arises the urgent necessity for continuous data cleaning.

Current techniques assume that the data and rules are *static* and available beforehand. They are inefficient dealing with *dynamic* use cases, neither as complicated as data streams when chunks of data come periodically and the data is not wholly available at once, nor even as simple as some slight changes of a few records. Of course, some incremental frameworks that can handle data changes have been proposed [10, 21, 22], but they actually stand limited to a few specific types of data or quality rules.

3.1.1 Motivating Scenarios

We present some fundamental scenarios to showcase the limitations of traditional data cleaning systems, highlighting the necessity of a continuous generalized data cleaning system and how vital such a system has become for today's big data and applications.

	name	city	zipcode	areacode	phone
c_1 :	Homer Simpson	Springfield	12345	111	1234567
c_2 :	Marge Simpson	Springfield	12345	111	2345678
c_3 :	Brian Griffin	Springfield	67890	111	9876543
c_4 :	Peter Griffin	Quahog	67890	999	8765432
c_5 :	Lois Griffin	Quahog	67890	999	7654321

(a) C : An instance of schema **customers**

	name	card	city	country	time
t_1 :	Homer Simpson	1111111	Springfield	USA	01 Nov 2014, 01:00 PM ET
t_2 :	Homer Simpson	1111111	Springfield	USA	01 Nov 2014, 01:15 PM ET
t_3 :	Peter Griffin	2222222	Quahog	USA	03 Dec 2014, 10:00 PM CT
t_4 :	Marge Simpson	1111111	Springfield	USA	22 Nov 2014, 10:00 AM ET
t_5 :	Lucille Botzowski	1111111	London	UK	22 Nov 2014, 02:47 PM GMT
t_6 :	Peter Griffin	2222222	Quahog	USA	03 Nov 2014, 11:00 AM CT
t_7 :	Steve Bellows	2222222	Vegas	USA	03 Dec 2014, 08:15 PM PT

(b) T : An instance of schema **transactions**

Figure 3.1. Motivating Example for Continuous Data Cleaning

Iterative Data Cleaning

Data cleaning is actually an iterative process. In one iteration, violating cells are identified; Then, the cleaning algorithm chooses the proper data changes for violating data cells to reach a fixed database. New violations might be introduced when applying those changes, which requires even more iterations to obtain a clean consistent data instance.

Refer to Figure 3.1. Consider Table C that contains customers information along with the following two rules: φ_1 (**zip code** uniquely determines customer's **city**) and φ_2 (**city** uniquely determines customer's **area code**). On the one hand, in one iteration, a traditional data cleaning system, say X , finds that c_3 is in violation with c_4 and c_5 according to φ_1 , as they all share the same **zip code** while having different **city** values.

Then, it would fix that error by modifying the wrong **city** field of c_3 to *Quahog*. A change that would cause another violation of the same records according to φ_2 , as they all share the same **city** now, but have different **area code** values. This violation can only be discovered in the second iteration, during which, System X would unnecessarily repeat some irrelevant comparisons such as $c_1 - c_2$ and $c_4 - c_5$ again finding no new corresponding violations.

On the other hand, an incremental data cleaning system should optimize this iterative cleaning process. After each iteration, it should incrementally handle data changes, reflecting on the data and the violations, and proceed to the next iteration without repeating the same irrelevant tuples comparisons or data changes again.

Interactive Data Cleaning

Another typical scenario would be a data cleaning system that allows the user to define the data source(s) along with the corresponding quality rules, provides the user with an interactive dashboard that visualizes the data, the rules, the detected violations and possible fixes, and allows for user interaction with those elements. This interaction with the data mainly helps the user analyze it, understand the connections among the data records themselves and with the rules, and investigate where the violations come from and possibly how to fix them.

More importantly, by allowing the user to change some values and immediately notice how this reflects on the data and violations, the system helps the user develop profound data quality rules that can indeed represent the semantics of the user's business or application.

This interactive dashboard would be impractical, rather impossible, to implement within a traditional data cleaning system that lacks any support for continuity. Even a small change of a single data cell in the database would require re-performing the whole cleaning process from scratch, which cannot be afforded in today's evolving big-data world.

Cleaning of Dynamic Data

In most of the applications, data is not available entirely at once, but rather comes in periodic chunks or batches, as soon as they are available. Consider the T table holding credit cards transactions in Figure 3.1, where the data comes into 3 different blocks as shown, and the rule φ_3 (If two transactions of the same card occur within one hour difference in two locations, mark those transactions as prospective fraud).

Thus, the flow of a continuous data cleaning system proceeds as follows. When the first block arrives, it only needs to examine the pair $t_1 - t_2$ for rule violations. When the second block arrives, the new records t_4 and t_5 need to be compared along with the old ones, i.e., t_1 and t_2 , and the possible fraud transactions $t_4 - t_5$ are noted (Note the time difference between different time zones). Notice that the system should intelligently avoid redoing any steps that have been conducted with the arrival of the first block (*e.g.*, $t_1 - t_2$ comparison). Then, when the third block arrives, the newest records t_6 and t_7 need to be compared with the old t_3 , and the pair $t_3 - t_7$ is marked as possible fraud. Again, a smart system would avoid repeating any vain operations that have been already performed with the arrival of the first two blocks of data (*e.g.*, $t_1 - t_4$ comparison).

Observe that fraud detection, as most of data streams cleaning scenarios, needs to be performed in real time, and cannot afford reprocessing all the old data again. With today's data volume and velocity, a continuous data cleaning system that would avoid reprocessing billions of records with each new block of data arriving, is no longer a luxury, but rather an essential approach for data quality.

3.1.2 Challenges

Looking back at our data cleaning system, NADEEF, like other state-of-the-art systems, its process assumes static data and rules, standing helpless against any dynamism. A user should first provide all the data tables, and the various data quality rules; Then, violations are detected, and, if specified by rules, repaired accordingly. For even a single change of just one cell, the current NADEEF can only handle it as a fresh new data instance, and would redo the whole process of violations detection and data repairing, repeating all the tuples comparisons and cells changes. With such inefficiency, it is incapable of coping with today's evolving data and applications.

Inspired by those aforementioned motivating scenarios, we propose several extensions to NADEEF's architecture to support *continuous* data cleaning. Recall that NADEEF distinguishes between a *programming interface* and a *core* to achieve *generality* and *extensibility*. The programming interface allows users to specify multiple types of data quality rules, while the core provides algorithms to detect violations and clean the data holistically without differentiating between these different types of rules.

Support for continuous data cleaning, that can efficiently handle small and big data changes, should target the following challenges:

Iterativity: Better performance for the traditional iterative data cleaning process;

Each iteration should build upon the output of the previous one, instead of redoing insignificant tasks.

Interactivity: Interactive data cleaning, where the user can interact with the data and inspect the effect of data changes on the go.

Continuity: Cleaning of dynamic data, when the data is not available at once, but rather comes into periodic increments.

Generality: Holistic data cleaning, with the capability of simultaneously handling different types of data quality rules.

3.1.3 Contributions

In this work, we build upon NADEEF support for generality, and propose several changes to its architecture, both the programming interface and the core, to support the other challenges incurred by data minor and major changes.

In this chapter, we make the following contributions:

- We optimize NADEEF’s iterative data cleaning to incrementally handle data changes throughout the cleaning process iterations. Recall that NADEEF is already generalized (supports multiple types of data quality rules) and extensible (allows defining new types of rules, or extending its core detection and cleaning algorithms).
- We formalize a **changes-aware programming interface** to facilitate optimized performance of user-defined rules, and we provide its compatible implementations for some predefined rules.
- We propose a **dynamic graphs repair module** as an extension in NADEEF’s core that can efficiently handle minor and major data changes incurred by continuous data cleaning, and avoid the problems other traditional techniques suffer from.
- We experiment our proposed extension against NADEEF, with multiple types of rules on various data sets, to evaluate their performance in different scenarios and use cases.

In the following, we refer to our extended framework for continuous data cleaning as NADEEF+, encompassing the changes-aware programming interface and the graph-based repair module, in contrast to NADEEF’s MAX-SAT solver and equivalence classes repair modules.

3.2 Related Work

Data Quality has gained much attention recently with several data cleaning and repairing approaches proposed in the literature. However, many of these approaches and systems target only certain types of data quality rules. Among these, [30] uses ETL (Extract, Transform and Load) tools and data transformations to merge and repair data. Other approaches focus on data repairing with regard to dependencies [7] and integrity constraints, such as Functional Dependencies (FDs) [11], Conditional Functional Dependencies (CFDs) [10,14,20], Inclusion Dependencies (INDs) [15], Conditional Inclusion Dependencies (CINDs) [16], Matching Dependencies (MDs) [12] and Denial Constraints [8,18].

Several techniques have been proposed targeting incremental data cleaning to deal with data changes without reprocessing the data from scratch. However, they also target only specific types of quality rules. A continuous data cleaning framework for FDs where the data and constraints are changing has been proposed in [21]. An incremental repairing algorithm for CFDs is introduced in [10]. SmartClean [22] is another incremental data cleaning tool, but can only deal with problems like missing values and domain violations.

Data Deduplication, being one of the most significant data quality problems (See [31] for a survey), has also been studied extensively in incremental context, in which data updates are encountered, making old linkage results obsolete. For Example, [41] introduces the idea of representative records of the whole original database; Still, the challenge is how these representatives get chosen. [42] proposed progressive techniques with the goal of maximizing the deduplication quality within a fixed time slot. Also, [43] proposes an end-to-end incremental record linkage framework using incremental graph clustering over similarity graphs. However, again, all these approaches target the problem of data deduplication specifically, and consequently could be leveraged to address MDs, but are incapable of tackling other types of data quality rules.

Some systems address data cleaning through an interactive process, in which the user’s feedback is employed to improve the accuracy of the model selection of candidate fixes. For example, AJAX [34] proposes modeling the data cleaning process as a directed graph of data transformations, allowing for user involvement to handle exceptional cases and inspect intermediate results. Potter’s Wheel [26] allows users to gradually build transformations to clean the data by adding or undoing transforms on a spreadsheet-like interface. Recently, [44] has also introduced a data cleaning framework that leverages user feedback through an interactive process, but it could only handle CFDs, CINDs and MDs. Observe also that these approaches handle only a limited set of rules types. Moreover, they deal only with small data changes and cannot support batch continuous data cleaning scenarios, when the data is not available all at once, but rather comes as periodic (big) incremental updates.

Other approaches have been proposed that target the cleaning of data streams. For example, [45] introduces the ESP framework for online cleaning of receptor data streams, to account for missing readings or outliers. It only targets specific types of data, and is inapplicable in other data cleaning scenarios.

NADEEF [24,25] is the first quest for a generalized data cleaning system that could handle different types of data quality rules, including user-defined rules that allow for custom data quality semantics. Yet, the current system stands ineffective against even the slightest data changes incurred within any of the aforementioned typical continuous data cleaning scenarios. We hereby propose NADEEF+ to target these scenarios and to address their corresponding challenges.

3.3 Limitations of Existing Data Cleaning Approaches

Other than lacking generality and supporting limited types of data quality rules, existing data cleaning solutions suffer from two main problems when it comes to continuous data cleaning: (1) Errors propagation from old data to new data; and (2) Committing to old “wrong” decisions taken due to incomplete information.

3.3.1 Domino Effect: Violations Propagation

	country	CC		country	CC		country	CC
t_1 :	Netherlands	44	t_1 :	Netherlands	44	t_1 :	UK	44
t_2 :	Netherlands	44	t_2 :	Netherlands	44	t_2 :	UK	44
t_3 :	UK	44	t_3 :	Netherlands	44	t_3 :	UK	44
t_4 :	UK	44	t_4 :	Netherlands	44	t_4 :	UK	44
t_5 :	UK	44	t_5 :	Netherlands	44	t_5 :	UK	44
(a) Dynamic Data			(b) Violations Propagation			(c) Correct Data Repair		

Figure 3.2. Dynamic Data and Domino Effect

Recall that with dynamic data as in our motivating scenarios, data might not be available as a whole from the beginning, but rather comes into separate batches. In such case, data cleaning systems suffer from the domino effect of violations propagating from old data to new data.

Consider the example in Figure 3.2, and assume that data arrives into 3 consecutive chunks (Figure 3.2(a)). Let X be a data cleaning solution trying to clean the data *w.r.t.* a single FD $\varphi_1 : \text{CC} \rightarrow \text{country}$ and targeting minimality.

When the first block of data arrives, X would detect a violation among tuples $t_1 : t_3$ since they all share the same CC, but have different **country**. A typical repair would be to change the cell $t_3[\text{country}]$ to Netherlands since it is the minimal possible change to reach a consistent database. When the second block of data arrives, t_4 now violates with the other tuples in the database, and again X would change its **country** to Netherlands to achieve consistency. The same would happen again for t_5 when the third block arrives, and this could keep going indefinitely (Figure 3.2(b)).

Had we had the full data from the beginning, the error in the old data would not have propagated to new data, and the end result would have been the correct consistent data repair in Figure 3.2(c).

3.3.2 Memoryless Repairs

Another tricky problem traditional solutions suffer from is what we can call memoryless repairs; data cleaning algorithms would make some decisions regarding data repairs at some point of time, then later when data changes are encountered, they focus on detecting new violations and how to repair them, but do not reconsider those old decisions to *rollback* any “wrong” ones taken due to incomplete information.

Consider another example in Figure 3.3. Let X be our equivalence-classes-based data cleaning solution (see Section 2.6), and assume we have two FDs: $\varphi_1 : \mathbf{CC} \rightarrow \mathbf{country}$ and $\varphi_2 : \mathbf{city, state, zipcode} \rightarrow \mathbf{country}$. The process would go as follows:

1. According to the first rule, X would detect a violation between tuples $t_1 : t_3$ since they have the same **zipcode** but different **country**. Hence, their **country** cells are all merged into one equivalence class eq_1 .
2. X would also detect another similar violation between tuples $t_4 : t_7$, so their **country** cells are all merged into one equivalence class eq_2 as well.
3. The second rule would trigger a violation between t_3 and t_4 because of their different **country** values. Consequently, the two corresponding equivalence classes eq_1 and eq_2 are merged into one eq_m .
4. With **country** cells of $t_1 : t_7$ all in one equivalence class, X targeting minimality would “repair” the data as shown in Figure 3.3(b).
5. Now, consider an update to t_4 **state** and **zipcode** as shown in Figure 3.3(b), X would not detect any inconsistencies and the database is considered “fixed”.
6. However, the connection between t_3 and t_4 is no longer applicable, and the two equivalence classes eq_1 and eq_2 should be *un-merged*, an operation that is not supported by traditional disjoint sets data structures.

Had we had the full data from the beginning, the resulting repair (Figure 3.2(c)) would have had no trouble with old “wrong” decisions that need to be reconsidered.

	city	state	zipcode	country	CC
t_1 :	Oxford	-	OX1 2PH	UK	44
t_2 :	Oxford	-	OX1 4AU	UK	44
t_3 :	London	-	WC2E 9DD	Finland	44
t_4 :	London	-	WC2E 9DD	Canada	1
t_5 :	Lafayette	IN	47905	USA	1
t_6 :	West Lafayette	IN	47906	USA	1
t_7 :	West Lafayette	IN	47907	USA	1

(a) Dirty Data

	city	state	zipcode	country	CC
t_1 :	Oxford	-	OX1 2PH	USA	44
t_2 :	Oxford	-	OX1 4AU	USA	44
t_3 :	London	-	WC2E 9DD	USA	44
t_4 :	London	OH	43140	USA	1
t_5 :	Lafayette	IN	47905	USA	1
t_6 :	West Lafayette	IN	47906	USA	1
t_7 :	West Lafayette	IN	47907	USA	1

(b) Memoryless Repairs

	city	state	zipcode	country	CC
t_1 :	Oxford	-	OX1 2PH	UK	44
t_2 :	Oxford	-	OX1 4AU	UK	44
t_3 :	London	-	WC2E 9DD	UK	44
t_4 :	London	OH	43140	USA	1
t_5 :	Lafayette	IN	47905	USA	1
t_6 :	West Lafayette	IN	47906	USA	1
t_7 :	West Lafayette	IN	47907	USA	1

(c) Correct Data Repair

Figure 3.3. Dynamic Data and Memoryless Repairs

3.4 Support for Continuous Data Cleaning in NADEEF+

We can formalize our problem as follows: Given the current database instance \mathcal{D} , a set of data quality rules Σ and an incremental batch of data changes $\Delta\mathcal{D}(\textit{insertions}, \textit{deletions}$ and/or $\textit{updates})$, clean the data as if it were all available at once, to output a new fixed data instance \mathcal{D}' .

A continuous data cleaning system should avoid reprocessing data during *violations detection*, and its *data repairing* should deal with problems such as: (1) The domino effect via keeping a history of data changes and data lineage; and (2) Memoryless repairs via undoing “wrong” repairs chosen because of incomplete data.

3.4.1 Continuous Violations Detection

Recall the task of violations detection in NADEEF from Section 2.6, the flow needs to be modified accordingly. The process should only inspect data partitions with new or changed tuples to avoid re-performing unnecessary tuple comparisons.

We explain how to adjust the programming interface so that it can handle data changes, how to modify the violations detection flow accordingly, and analyze how this would affect the task complexity.

Changes-Aware Programming Interface

Looking back at our optimized programming interface in Figure 2.5 in the context of dynamic data, we note that: (1) The *scope* operator does not need to be changed as the relevant rows and columns selection criteria would still be the same. (2) Similarly, the **detect** and **fix** operators are not affected; a collection of tuples would produce a violation, and the corresponding candidate fixes would still be the same as well, regardless of the fact that it contains old or new tuples. (3) However, to gain better performance, the *block* and *iterator* operators can be supplied with extra information about the changed tuples. We highlight the changes-aware interface in Figure 3.4.

```

Interface Rule {
    Table scope(Table table);

    Tables block(Table table, Map changes);
    Tuples iterator(Table block, Map changes);

    Violations detect(Tuples tuples);
    Fixes fix(Violation violation);
}

```

Figure 3.4. Changes-Aware Programming Interface in NADEEF+

Continuous Violations Detection Flow

Notice that the flow is modified to handle the newly inserted tuples. Yet, when tuples are deleted, the corresponding violations are simply deleted from the violations table. Moreover, an updated tuple can be considered a newly inserted tuple after a deletion operation. In that sense, we can say that NADEEF+ violations detection can support insertions, deletions and updates as data changes.

We highlight the flow to handle pair-tuple rules. For single-tuple rules, our system needs only to detect the violations for the newly inserted or updated tuples individually and delete the obsolete violations corresponding to the deleted tuples.

Recall that violations are detected in multiple separate parallel threads corresponding to different rules, without any communication between them. Violations are repaired holistically to gain better quality, but detecting violations for a rule can be handled separately in its own flow. Hence, here we focus on changes to the detec-

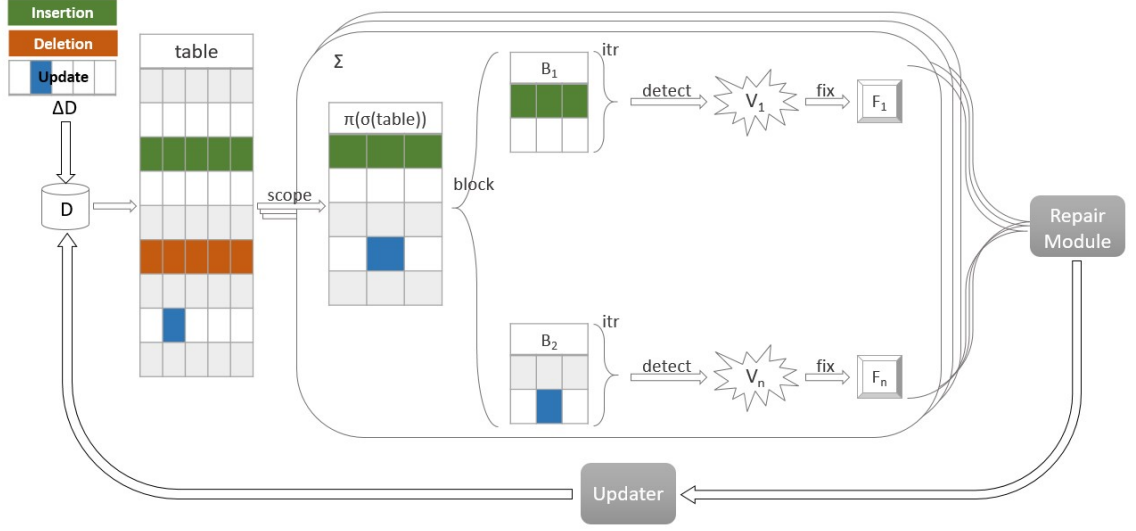


Figure 3.5. Violations Detection Flow in NADEEF+

tion flow for a single rule. It can be directly extended to multiple rules without any loss of generality. The changes-aware detection flow is given in Figure 3.5.

Recall that the *block* operator allows the user to define a way in which a table is partitioned into smaller blocks that are later iterated to find violations, such that any problematic tuples would fall into the same block. As an illustrative example, the *block* operator for an FD is given in Figure 3.6. It should return the set of blocks containing only tuples that are relevant to data changes. Blocks that have no changes are not expected to give any new violations, and hence should be skipped. The values of the left-hand-side (LHS) columns of changed tuples would be transformed into a filtration expression, upon which the table is filtered before it is partitioned. A table is partitioned on the values of the rule LHS, because for an FD, two tuples might cause a violation only if they share the same LHS.

As for the *iterator* operator, it is meant to allow the user to optimize the iteration over a block, in order to gain better performance than the default pair-wise traversal. An example for the FD rule is given in Figure 3.6, in which a block is sorted on the

```

Class FD {
  Tables block(Table table, Map changes) {
    formulate changed tuples into a filter expression exp
    table.filter(exp);
    return table.groupOn(LHS);
  }

  Tuples iterator(Table block, Map changes) {
    block.orderBy(RHS);
    while linearly scanning the block do
      ...
      if tuple is a changed tuple then
        ...
        output tuples for comparison
  }
}

```

Figure 3.6. Sample Changes-Aware Rule

values of the right-hand-side (RHS) columns of its tuples, and then linearly scanned to find violating tuples that have different values; Recall that any two tuples inside a block already have the same left-hand-side columns. To handle data changes, while linearly scanning the block, a tuple-pair is output to the stream if either one of the tuples is new; otherwise, the tuple-pair is already checked before and any possible violation is already detected and repaired.

It is worth noting that these changes-aware operators are also optional, to be implemented for the sake of an optimized performance. Otherwise, NADEEF+ would use the brute-force methods by default. We already provide optimized built-in implementations for a number of data quality rules, *e.g.*, FDs, CFDs and MDs.

Performance Gain

Suppose we have a database table of size N , and a set of newly inserted tuples of size M , for which we need to decide how to find the new resulting violations. We can either merge the whole data altogether and rerun the detection process from scratch, or we can apply our proposed continuous detection flow. On the one hand, rerunning the detection process on the merged data has an $O((N + M)^2)$ time complexity. On the other hand, with continuous violations detection, comparing only the new M tuples with themselves and with the old data, hence avoiding comparing tuples that have been inspected before, gives an $O(M * (N + M))$ performance, which proves very efficient in typical cases when $M \ll N$.

3.4.2 Continuous Data Repairing

Looking back at the core of NADEEF, one of its main contributions is its *extensibility*, where users can plug different repairing algorithms to meet their needs for effectiveness and efficiency.

Recall the two repair modules introduced in Section 2.6 with different goals and use cases: (1) The Weighted SAT-Solver repair module that targets accuracy, but could not scale to reasonable data sizes users face in realistic applications; and (2) The equivalence classes repair module that targets efficiency while maintaining acceptable accuracy, but proved ineffective when it comes to continuous data cleaning as discussed in Section 3.3.

We propose a new repair module that can effectively and efficiently support continuous data cleaning, and that avoids the problems other techniques suffer from, *e.g.*, violations propagation and memoryless repairs.

Dynamic Graphs Repair Module

The equivalence classes repair module employs the disjoint sets (union-find) data structure to support the operations required for data repairing, *i.e.*, the merging of two equivalence classes (*union*), and assigning a target value (fix) to every equivalence class from its candidate values (*find*). In the context of continuous data cleaning, data is changing, and the repairing algorithm needs to adapt accordingly, so that it may reconsider some of the decisions made earlier. However, with no support for *unmerging* in its data structure, the equivalence classes repair module cannot effectively deal with dynamic data, because it assumes that all the data is available and is static, and that all candidate fixes will be provided for it to choose which ones to apply as a data repair.

For the above reasons, we propose our graph-based repair module that can support connecting two data cells when a fix suggests so, and disconnecting them when the underlying data changes imply otherwise.

Graph Connected Components

A *graph connected-component* consists of a set, say C , of cells. In a database, say \mathcal{D} , each cell, say c , has an associated component, denoted by $\text{cc}(c)$. A connected component C is associated with a set of candidate values, denoted by $\text{cand}(C)$, and a unique *target* value, denoted by $\text{targ}(C)$.

Recall back our ongoing example from Figures 1.1 and 2.1, and reconsider the corresponding violations and candidate fixes of φ_1 in Table 2.1. Figure 3.7 gives an example of the proposed dynamic graphs repair module in action. We show a subset of the data again in Figure 3.7(a) highlighting the violating cells. Given a set \mathcal{V} of violations and a set \mathcal{F} of candidate fixes as input, the proposed dynamic graphs repair module would act as follows (See Figure 3.7(b)):

1. **Initialization:** Each cell c involved in \mathcal{V} and \mathcal{F} is a single graph node representing its own component $\text{cc}(c)$, and its candidate value is its current cell value.

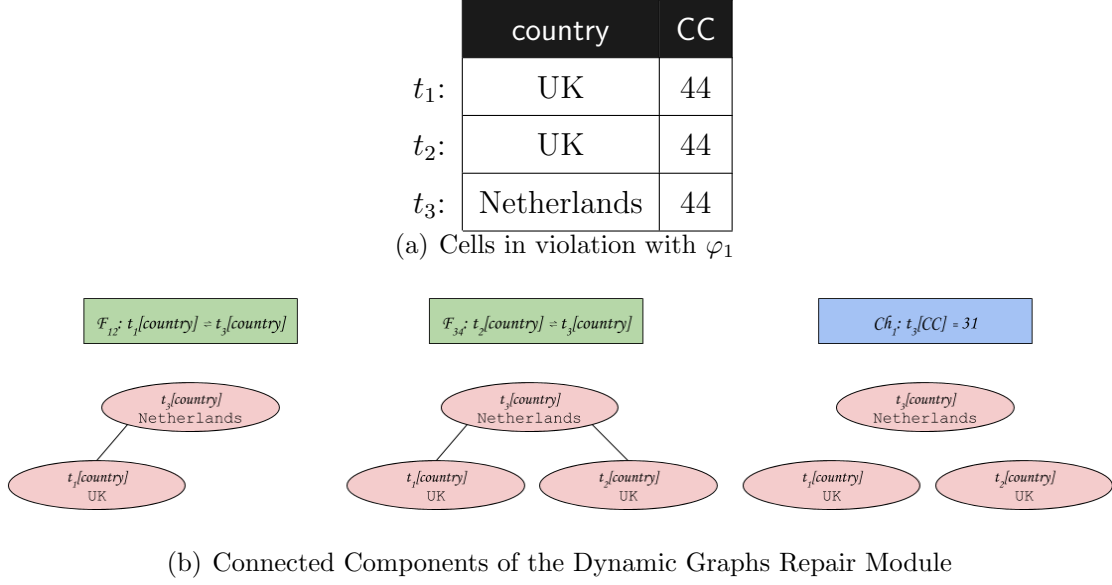


Figure 3.7. Dynamic Graphs Repair Module

2. Connect components:

(a) If there is a candidate fix, say $c_1 \leftarrow c_2$ (or $c_2 \leftarrow c_1$) in \mathcal{F} , then the two nodes c_1 and c_2 will be connected with an edge, and the new set of candidate values is the union of the two sets of candidate values for $\text{cc}(c_1)$ and $\text{cc}(c_2)$, *i.e.*, $\text{cand}(\text{cc}(c_1)) \cup \text{cand}(\text{cc}(c_2))$.

(b) If there is a candidate fix, say $c_1 \leftarrow x$ (*i.e.*, c_1 should be assigned the value x), then the candidate values of $\text{cc}(c_1)$ will become $\text{cand}(\text{cc}(c_1)) \cup \{x\}$.

3. **Disconnect components:** Whenever a data change is encountered in one tuple, it is marked as a changed tuple. Its cells should be disconnected from other cells by breaking any edges coming into or out of them, because these connections might have become obsolete, representing fixes for violations that no longer exist. Candidate values for each component should be updated accordingly. Changed tuples are passed in the *changes* map to the rule operators and would be inspected again in case they cause any new violations.

4. **Assign a target value:** For each connected component, say C , select one target value $\text{targ}(C)$ from its candidate values $\text{cand}(C)$, such that the total cost of changing all cell values in C to $\text{targ}(C)$ is minimum.

We separate the decision of which cell values should be the same from that of what target value should be assigned to a graph connected-component. We defer the assignment of $\text{targ}(C)$ as late as possible to reduce poor local decisions.

Furthermore, notice that the compression technique discussed previously can be readily applied here. The cost of making a super cell value change in a component is multiplied by $|\text{ext}(c)|$, the cardinality of the super cell.

Solution for Domino Effect

Our proposed repair module would consider not only the values of the repaired data cells, but also their original values and reliable data changes. This way, whenever assigning a target value for a connect component, errors will not propagate from the old data to the new data, and the data would be repaired as if it has been entirely available from the beginning.

Looking back at the example in Figure 3.2 that demonstrates the domino effect in the propagation of violations, the **country** cells of tuples $t_1 : t_5$ would all be in the same connected component because they all violate φ_1 sharing the same CC. But when choosing a target value for this component, NADEEF+ would figure out that assigning UK as their **country** would be the repair requiring the minimum number of changes, acquiring the correct fixed database instance given in Figure 3.2(c), as if all the data has been available in one single chunk in the first place.

Solution for Memoryless Repairs

As explained earlier, data changes may trigger disconnections in the underlying graph data structure. In this case, our proposed repair module would reconsider its relevant old decisions and *undo* or *rollback* any “wrong” ones when the entire data has not been available, or has been missing or incomplete.

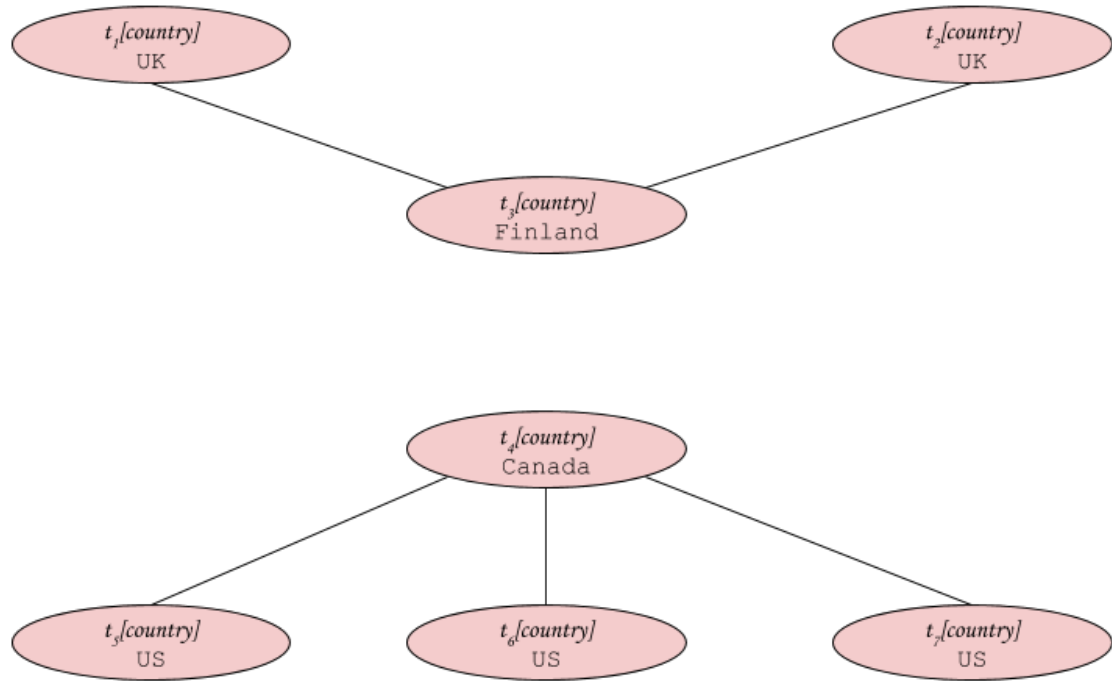


Figure 3.8. Dynamic Data and Memoryful Repairs

Refer to the example in Figure 3.3 that showcases the issue with memoryless repairs and committing to old “wrong” decisions. When t_4 ’s **state** and **zipcode** changes are reported to NADEEF+, the proposed dynamic graphs repair module would eliminate the edge between t_3 and t_4 breaking the underlying graph into two separate connected components, as in Figure 3.8.

When choosing a target value for these components, the repairing algorithm and the **Updater**, for the sake of minimality, would assign UK as the **country** for the tuples $t_1 : t_3$ and USA for the tuples $t_4 : t_7$, obtaining the correct fixed database instance of Figure 3.3(c), again as if all the data has been available from the beginning and no changes have been encountered.

Complexity

In contrast to equivalence classes, dealing with dynamic graphs incurs some extra complexity, when querying for connectivity to find connected components and assign target values. However, the data is actually partitioned into *small* blocks such that the various graph operations, *e.g.*, inserting, deleting and querying for connectivity, are practically fast enough (despite the worst-case $O(N)$ time-complexity). We empirically verify the efficiency of our proposed repair module in the experimental study in Section 3.5.

Moreover, to emphasize NADEEF’s extensibility, we could further extend our repairing algorithm or plug in an enhanced repair module, to use some more efficient poly-logarithmic techniques proposed for dynamic graphs connectivity [46,47] without requiring any other modifications inside the system core.

3.5 Experimental Study

We evaluate our continuous data-cleaning system NADEEF+ against real-world datasets to measure its performance against dynamic data. Our goal is to achieve the same data quality as if the whole data has been available from the beginning, while avoiding reprocessing the data unnecessarily during violations detection, and dealing with the domino effect and memoryless repairs issues while repairing the data.

We monitor how NADEEF+ responds to data changes in two cases:

Minor Data Changes: For example, interactive data cleaning when the user might change a few tuples and wants to explore how this would reflect on the underlying data and rules violations.

Major Data Changes: For example, when the data is not available in whole from the beginning, but rather comes in incremental batches.

3.5.1 Experimental Settings

Datasets

In these experiments, we use the same two real-life datasets HOSP and BUS that we use for evaluating NADEEF’s effectiveness and efficiency (Section 2.7):

1. *HOSP dataset*: Taken from the US Department of Health & Human Services with 100K records and 9 attributes.
2. *BUS dataset*: A one-table dataset obtained by joining 8 tables using primary-foreign key relationships from the UK government public datasets with 160K tuples and 16 attributes.

Algorithms

We run the experiments using the recent open-sourced Java implementation of NADEEF¹ evaluating mainly two scenarios:

1. Assuming all data is available at once and running NADEEF’s violations detection and data repairing (via the equivalence classes repair module EQU) from scratch on the whole data.
2. Acquiring the dynamic data and the corresponding data changes in batches, and running NADEEF+’s continuous violations detection and data repairing (via the dynamic graphs repair module) after each batch.

In all experiments, we verify that the two mentioned scenarios give the same output in terms of cleaning quality as if the whole data has been available all at once: (1) Re-running NADEEF’s cleaning process on the whole data from scratch, or (2) Running NADEEF’s continuous data cleaning on the “cleaned” data with only the new data changes being considered.

All the experiments are conducted on an Ubuntu 18.04 machine with a 3.4GHz Intel CPU and 8GB of memory.

¹<https://github.com/daqcri/NADEEF>

3.5.2 Continuity

Graphs Connected-Components

Table 3.1.
Graphs Connected Components for HOSP Dataset

Count	29
Minimum Size	50
Maximum Size	75
Average Size	52.79

In the first set of experiments we evaluate the extra cost incurred in NADEEF+ due to querying for connectivity in dynamic graphs, in contrast to querying for connectivity in NADEEF’s equivalence classes. Despite the worst case $O(N)$ time complexity, the sizes of the graphs’ connected-components are reasonably small in practice, yielding cheap operations that are not costly for real-world data.

Table 3.1 shows the corresponding results when running data cleaning on the HOSP dataset. Observe that the *small* average size of the graphs’ connected-components would result in cheap operations when inserting or deleting nodes or edges, or querying for connectivity in the underlying graphs.

In other words, in reality, we are not sacrificing the efficiency of the repairing algorithm, in spite of replacing the equivalence classes module in NADEEF with the dynamic graphs module in NADEEF+ to handle dynamic data and support continuous data cleaning.

Minor Data Changes

In this set of experiments, we evaluate how NADEEF+ would handle minor data changes. This is typically encountered with interactive data cleaning, when the user would issue changes for a few tuples, and explore how these changes would affect the current data quality status or trigger new violations.

Table 3.2.
Response Times for Minor Data Changes

Δ Size	HOSP - Time (msec)	BUS - Time (msec)
1	421	395
10	431	1,005
100	858	4,217

Consequently, with the user involvement, response time has to be small enough to allow for interactivity through the data cleaning process. A response is expected momentarily as shown in Table 3.2 for various sizes of data changes for our two datasets. In comparison to the 12.52 minutes needed to clean 100K records of HOSP dataset (resp. 18.53 minutes for BUS dataset) initially, the reported response times are acceptable.

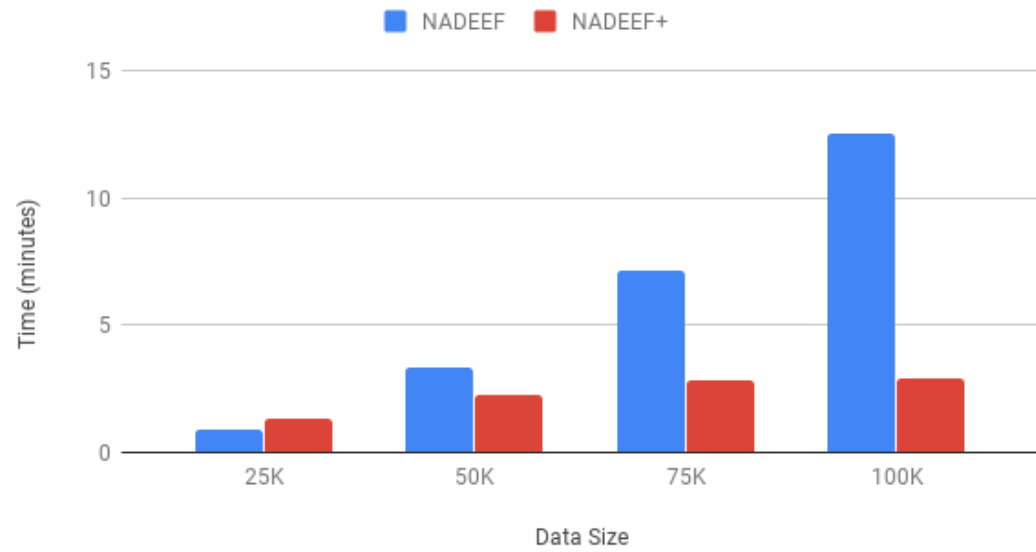
Major Data Changes

In this set of experiments, we evaluate how NADEEF+ would handle the more challenging use case of major data changes. In this scenario, data is not available in whole at once, but rather comes incrementally in batches, a use case that mandates the need for continuous data cleaning in the first place.

In Figure 3.9, data arrives in batches of 25K each. Thus, in a sense, the x-axis may also resemble the progression over time as the data arrives continuously. There is an overhead in the beginning with all the modifications within NADEEF+ that makes it initially less efficient than NADEEF. However, as the data grows, continuously detecting violations and repairing data pays off, and the gain overcomes that overhead.

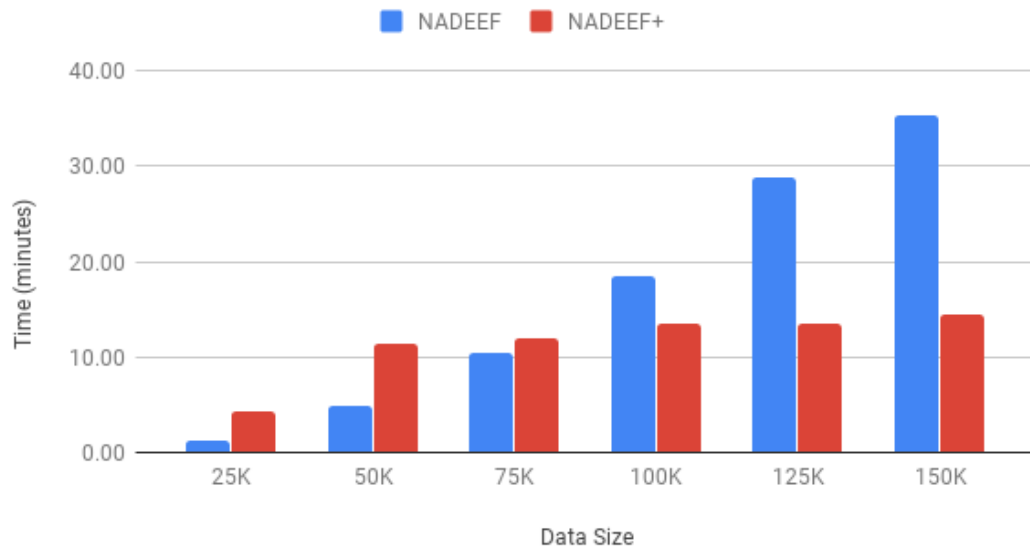
This observation illustrates the much-needed efficient process of continuous data cleaning in NADEEF+ that can integrate major data changes or a new data batch with the old accumulated “cleaned” data in minutes rather than rerunning the whole lengthy process from scratch; a process that can take hours or even days with real-world data sizes.

HOSP



(a) HOSP Dataset

BUS



(b) BUS Dataset

Figure 3.9. Running Times for Major Data Changes

3.6 Concluding Remarks

We present NADEEF+, an extension to NADEEF, to support continuous data cleaning. *Continuity* is a serious challenge that needs to be addressed when dealing with either minor or major data changes. We demonstrate how the proposed system can handle various use cases that traditional data cleaning systems cannot support, *e.g.*, interactive data cleaning and cleaning of dynamic batch data.

On the one hand, existing solutions assume the data is static and available in whole in the first place, and consequently suffer from different critical problems when dealing with dynamic data, *e.g.*, errors propagation and memoryless repairs. On the other hand, NADEEF+ can efficiently and continuously detect violations and repair the data while avoiding these problems.

We propose a modified changes-aware programming interface for continuous violations detection without reprocessing the whole data from scratch. We also propose a dynamic-graphs repair module that can support continuous data repairing. Finally, we experimentally evaluate NADEEF+ on real-world datasets to verify its effectiveness and efficiency against both minor and major changes in dynamic data.

4 DATA CLEANING EXPLANATIONS

4.1 Human-in-the-Loop and the Need for Explainability

Data cleaning has received considerable attention in both industry and academia lately. Rule-based methods [7, 18], including NADEEF [24, 25], have been proposed as effective and efficient solutions. However, several other studies [19, 20, 48, 49] demonstrate that machine learning (ML)-based methods provide comparable results.

Although ML-based methods bring in high-accuracy results, they are usually difficult to understand. Data cleaning systems often aim for higher accuracy and better performance without enough focus on the explainability of the model or the repairs.

Some approaches rely on human-in-the-loop data cleaning [19, 20] because it yields more reliable repairs. They leverage the knowledge they learn from the user, but very few of them try to explain their processes back to the user.

Nowadays, ML affects life-altering actions, *e.g.*, loan approval, job hiring, and medical diagnosis. There is the critical need and motivation for explainability of ML results. Explanations are necessary to build trust in the decision process, and to increase the adoption of (semi-)automated systems. Explanations allow for informed human involvement and for obtaining user feedback. Moreover, explanations help experts and developers debug errors, compare approaches, and improve functionality. Besides, this transparency is now required by new laws and regulations to justify how these decisions are made.

A recent study of explainability in data integration systems [23] concurs that there have been some approaches towards explainability for tasks like schema matching, schema mapping, record linkage and data fusion. However, more attention is usually paid to effectiveness and efficiency at the expense of explainability of the repairing model, the detected violations, or the applied repairs.

4.1.1 Challenges

The following challenges arise when targeting data cleaning explanations:

Interpretability: We need to provide understandable user-friendly explanations to the end-user.

Model-Agnostic Explainability: We cannot assume any knowledge of the underlying data cleaning model internals.

Interactivity: A data cleaning framework should explain how the model would behave if certain features are different, provide flexibility to navigate through different granularity levels of explanations, and allow for comparing between different underlying models.

Audience Diversity: A data cleaning framework has to target different audience types and levels of expertise, and provide different functionalities for either regular users who want to visualize explanations of a cleaning model on a specific dataset, or experts who want to improve the model, engineer its features, or debug its errors.

4.2 Related Work

Several data cleaning approaches have been proposed in literature for the past few decades. Some leverage experts' knowledge via enforcing a set of pre-defined data quality rules on the data, *i.e.*, that data has to obey the rules and be consistent. [7] provides an overview of dependency-theory and different constraints in the context of data cleaning. Our own proposal for NADEEF [24,25] is also an approach towards a generalized rule-based data cleaning system that can deal with different types of rules. The approaches that employ data quality rules are relatively intuitive and explainable by definition.

More recently, other approaches employ machine learning techniques aiming for more accuracy and scalability. For example, Yakout et al. [48] propose a new data repairing approach based on maximizing the likelihood of replacement data given the data distribution that can be modeled using statistical machine learning techniques. HoloClean [49] proposes holistic data repairing driven by probabilistic inference leveraging statistical properties of the input data to ensure scalability to databases with millions of tuples. However, ML-based methods are relatively complicated and more challenging to explain.

Data repairing or cleaning cannot always be a completely automated process. Often, humans are involved to assist in cleaning the data. Sometimes, expert users provide their help by answering some questions or by verifying data repairs. Other approaches adopt *active learning*, where the model chooses the data instances to learn to improve its quality, *e.g.*, GDR [20] a Guided Data Repair framework that incorporates user feedback in the data cleaning process, ALIAS [50] a learning-based system that uses the idea of "reject region" to reduce the training data size, and [51] that targets higher quality and scalability. With the lack of affordable expert users, other approaches have employed *crowdsourcing*, in various data management contexts to enhance the quality with multiple users' feedback. For example, CrowdDB [52] uses crowdsourcing to process queries that neither database systems nor search engines can adequately answer. KATARA [53] employs knowledge bases and crowds to interpret table semantics, identify correct and incorrect data, and generate top-k possible repairs for incorrect data. CrowdER [54] uses a hybrid human-machine approach in which machines are used to do an initial coarse pass over all the data, and then humans verify only the most likely matching pairs.

However, most of the techniques focus on how to learn from the user, and neglect how to teach the user. Although there have been some trials towards explaining and explainable data management systems, as recently investigated in [23], we are still far from user-friendly data cleaning explanations, and several efforts are yet to be made towards enhancing the interpretability of data management systems.

4.3 Properties of Explanations

When discussing the explainability of data cleaning systems, we have to take multiple properties of explanations [23] into account.

First, we have to highlight the *causality* of data cleaning systems. A causal explanation provides enough evidence to support or refute the task outcome. Non-causal explanations might not provide such evidence, but help understand the results via illustrative examples, visualizations or summaries.

Data cleaning systems should put their *audience* as top priority. Different users have different skills and different needs. With the right tools, one can leverage the users' skills and meet their needs. The depth of details and flexibility in explanations should match the users' expectations, whether it is a domain expert that analyzes a whole model, an educated user that is inspecting an area of interest or regular non-experts that are trying to understand the reasoning for some specific data repairing decisions.

This naturally leads to various *formats* of explanations, being it in the form of rules or decision trees, some statistics of the data, visualizations of violations and data repairs, or post-processing of data instances, *e.g.*, summarization and top-K representatives. We highlight various examples in our coverage of explanations in NADEEF and explanations beyond rule-based data cleaning in the following sections.

Moreover, data cleaning explanations should target high *coverage* that measures the amount of evidence *w.r.t.* the data cleaning results, *i.e.*, how much one can replicate the algorithm decisions based on its explanations.

Lastly, and on top of the list, comes the *understandability* of explanations, such that they are interpretable enough and user-friendly for different types of users. Data management tasks are complicated enough that we do not need an extra layer of complex explanations.

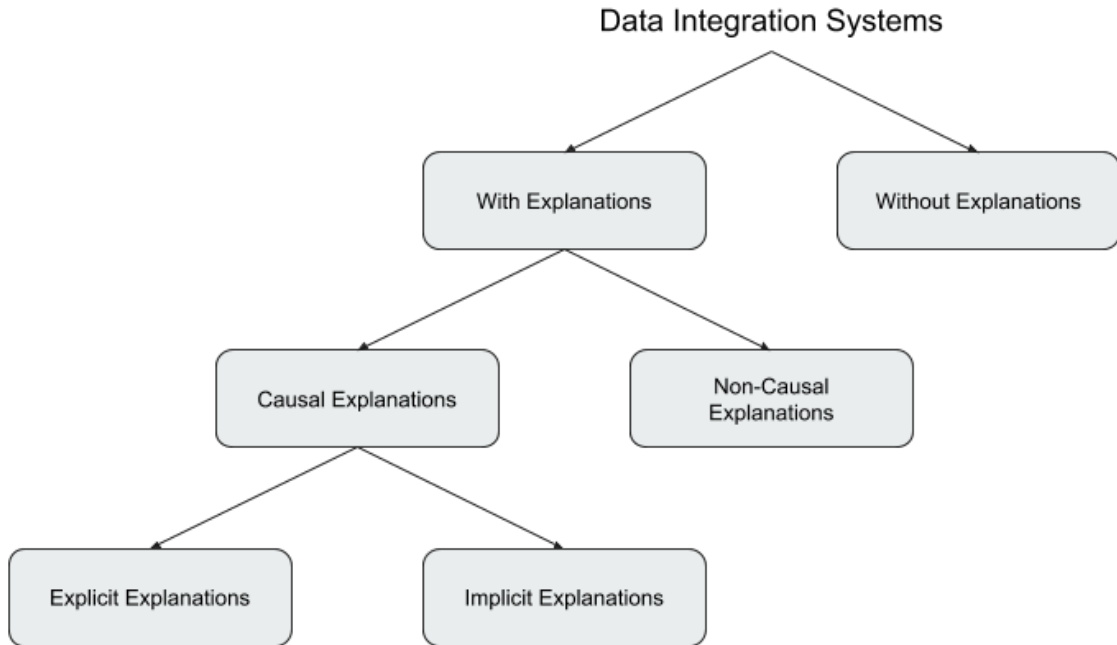


Figure 4.1. Explainability in Data Integration Systems

4.4 Explainability in Data Integration Systems

Wang et al. [23] study the explainability in recent data integration systems, and provide a neat hierarchy of data cleaning explanations, as in Figure 4.1. In this section, we cover these different classes of systems in terms of explainability, and identify where our different pieces of work would fit in this ontology.

The first distinction comes between systems that provide explanations for their decisions vs. those that do not. Systems without explanations account for a broad class of **unexplainable** data integration systems. Examples include several probabilistic-based [49] and ML-based [20, 48] approaches.

Then, comes the next dimension within the class of systems with explanations that separates their explanations into two sub-classes: (1) *Causal explanations* that could answer "how" or "why" a specific decision has been made by providing the reasoning behind different steps of a data integration process; and (2) *Non-causal explanations* that focus on "what" the decision is without giving much attention to why it has been made.

While causal explanations try to justify the process decisions, non-causal explanations illustrate them through simplifications, summarizations, examples, visualizations, etc. These **illustrative** techniques are common practices in systems that involve the human-in-the-loop [50, 55–57].

The final branching within the class of systems with causal explanations distinguishes between explaining and explainable systems. On the one hand, **explaining** systems provide *explicit* explanations of the data and the task, with goals such as conciseness, so they often provide understandable and simple explanations [58–60]. On the other hand, in **explainable** systems, *implicit* explanations come as a by-product; these systems focus more on the accuracy and scalability of their tasks, and this often results in hard and complex explanations [61–63].

In this dissertation, we highlight how we target the challenge of *explainability* in data cleaning systems that we outline in the introduction of Chapter 1. We locate NADEEF under the class of *explaining* data cleaning systems, because it provides clear explicit explanations to the end user highlighting the violations of the rules and the corresponding data repairs.

We propose EXPLAINER, a tool for model-agnostic entity resolution explanations, to illustrate explainability beyond rule-based data cleaning, and showcase our approach towards interpreting some *unexplainable* data cleaning systems.

4.5 Data Cleaning Explanations in NADEEF

Recall that NADEEF employs a data-quality dashboard to keep full lineage and auditing information about violations and repairs to help understand the health of the database as well as its data cleaning process. Through this dashboard, NADEEF provides explanations for both detected violations and data repairs.

As for the violations, it explains why a tuple or a tuple-pair is detected as an error by which data quality rule via providing the violating cells. This helps explain the errors and how they have been introduced in the first place, whether through the initial data, after some data changes, or even due to data repairs *w.r.t.* to other data quality rules.

As for the repairs, it highlights each data repair the **Updater** selects, from what and to which value the cell is changed, to resolve which violation, and according to the semantics of which rule. This helps verify the validity of the repair and explains how it has been derived.

4.5.1 Violations Explanations

Table 4.1.
Violations (Explanations) in NADEEF

Rule	Violation	Values
φ_1	$V_1: \{t_1[\text{CC, country}], t_3[\text{CC, country}]\}$	$\{[44, \text{UK}], [44, \text{Netherlands}]\}$
φ_1	$V_2: \{t_2[\text{CC, country}], t_3[\text{CC, country}]\}$	$\{[44, \text{UK}], [44, \text{Netherlands}]\}$
φ_2	$V_3: \{t_3[\text{FN, LN, street, city, country, CC, phone}],$ $c_1[\text{FN, LN, street, city, country, CC, tel}]\}$	$\{[\text{David}, \dots 44, 66700541],$ $[\text{David}, \dots 44, 66700543]\}$
φ_3	$V_4: \{t_1[\text{FN, LN, CC, phone, when, where}],$ $t_3[\text{FN, LN, CC, phone, when, where}]\}$	$\{[\text{David}, \dots \text{Netherlands}],$ $[\text{David}, \dots \text{'NY, USA'}]\}$

Refer to our ongoing example in Figures 1.1 and 2.1, and the corresponding violations and candidate fixes in Table 2.1. By definition, a violation is a set of cells that together cause some error *w.r.t.* to the static semantics of a specific data quality rule. We show again the violations of our example in Table 4.1 to highlight that each violation indicates what rule is violated and defines exactly which cells together cause the violation as a form of explicit explanations for the detected errors.

In NADEEF, we have further implemented some charts and graphs to summarize and explain violations as in Figure 4.2.

Overview: This indicates the amount of data that is either involved in violations or is considered clean.

Error Distribution on Rules: This shows how many violations are detected for each rule, and how many data tables are involved.

Error Distribution on Attributes: This shows the number of values that are involved in violations for each attribute. This is to reflect the *dirtyiness* relative to various attributes.

Violations Graph: This reflects the violations *w.r.t.* each data quality rule. In this graph, each node represents a rule. There is an edge between two nodes indicating that there are common cells that are involved in the violations for each rule (*i.e.*, node). The thickness of an edge indicates the number of common cells involved in the violation over both rules (*i.e.*, two ends of the edge).

For each, the users can drill down to see the corresponding part in the violations table, or they can specify search predicates to select the data they want to further explore, as shown in the bottom of Figure 4.2.

By leveraging the data quality dashboard, NADEEF allows easy interaction with the users such that they can easily switch between different visualization modalities and identify errors based on their expertise and knowledge of the data.

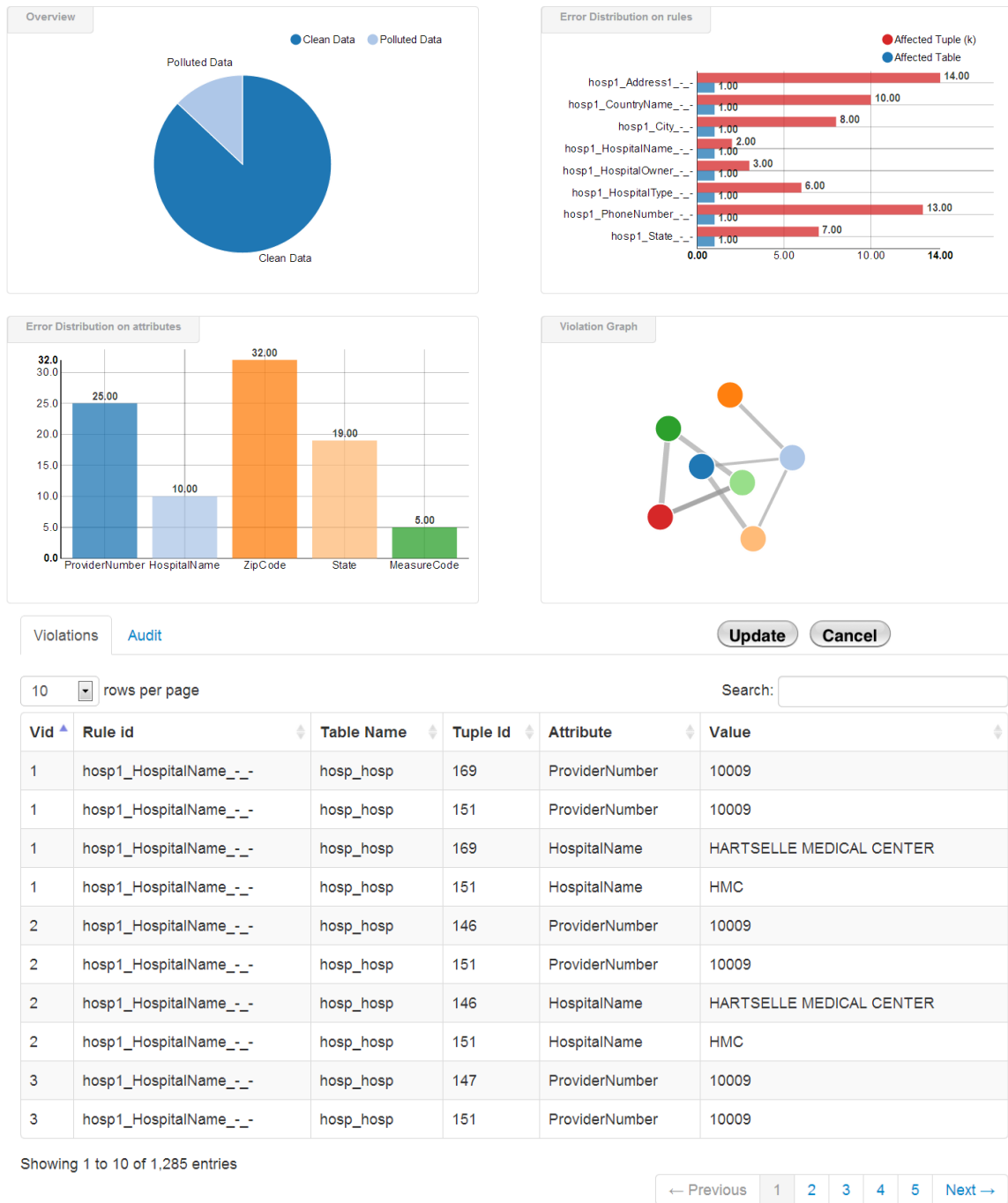


Figure 4.2. NADEEF's Data Quality Dashboard

4.5.2 Repairs Explanations

Table	Tid	Attribute	Old	New	User	Timestamp
hospital	169	ProviderNumber	10009	10008	User1	02/01/13 14:23:14
hospital	151	ProviderNumber	10009	10008	nadeef	02/01/13 14:23:30

Figure 4.3. Repairs Auditing in NADEEF

Moreover, NADEEF provides a data auditing facility such that, after data repairs, the users may inspect the different changes made to the data. Figure 4.3 shows some examples. The first five attributes identify the updated cells, the attribute **User** identifies who or what changed the data, either a specific user, a data change or NADEEF, and **Timestamp** specifies when this update has been committed to the database.

Table 4.2.
Repairs (Explanations) in NADEEF

table	tuple_id	attribute	old_value	new_value	rule	violation	fix
transactions	3	country	Netherlands	UK	φ_1	V_1	F_2
transactions	3	phone	66700541	66700543	φ_2	V_3	F_5

Specifically, this auditing information functions as explanations for the data repairs. It shows how each specific repair has been derived, via logging the data-value changes, linking to the corresponding rules, violations and candidate fixes. Table 4.2 shows the explanations of the data repairs for our continuing example.

4.6 Explanations beyond Rule-based Data Cleaning

In this chapter, we try to address explainability beyond rule-based data cleaning via interpreting some *unexplainable* data cleaning systems. We target entity resolution as a use-case, and propose a tool that can explain entity resolution classifiers with different levels of granularity.

4.6.1 Use Case: Entity Resolution Explanations

Entity Resolution (ER, for short), *a.k.a.* Record Linkage, Entity Matching, or Duplicate Detection, is a fundamental data cleaning and integration problem that has received notable attention in the past few decades. It identifies pairs of data instances that refer to the same real-world entity, and it has been the subject of many investigations [31,64]. Several recent studies [65–67] show that machine learning (ML)-based methods often provide state-of-the-art results for ER.

While rule-based methods have been used in many practical scenarios and are often easy to understand, machine-learning-based methods provide the best accuracy, but the state-of-the-art classifiers are very opaque. There has been some work towards understanding and debugging the early stages of the entity resolution pipeline, *e.g.*, blocking and generating features (similarity scores). However, there are no such efforts for explaining the model or its predictions.

A key impediment to using these ML-based solutions in practice is that end-users are given the output (*i.e.*, the matching tuples) without sufficient explanation of why these tuples are matching. This state of affairs may hinder the use of these ML-based solutions even if they deliver the best results.

The aforementioned study of explainability in data integration systems [23] notes that there have been several approaches towards *explaining* systems (*i.e.*, explicit causal explanations) for tasks like schema matching, schema mapping and data fusion. However, none of the current ER systems explicitly explain their results.

4.6.2 Contributions

In this chapter on Data Cleaning eXplanations ¹, we propose EXPLAINER, a tool to understand and explain entity resolution classifiers with different granularity levels of explanations. It takes two input datasets to be deduplicated along with an ML model that is trained for such task, and in turn helps users understand the outcome of the ML model from various angles. We demonstrate how EXPLAINER can handle different scenarios for a variety of classifiers.

For this purpose, we adapt general-purpose explanation tools into the context of ER. More specifically, we leverage LIME [68] and Anchors [69] for *local explanations*, Bayesian Rule Lists (BRL) [70] for *global explanations*, and Skater [71] for an industry-level hybrid of both flavors. While these frameworks provide useful instance-level or model-level explanations, these are not sufficient in the context of ER. In EXPLAINER, we build upon them and extend their functionalities to provide more profound explanations and deeper analyses of their collective outcomes.

EXPLAINER provides the following functionalities for explaining ML-based ER:

- **Global Explanations:** We post-process local explanations to help explain the whole model and how different features drive its predictions. In this regard, along with the tools-provided global explanations, we also derive feature importance, visualize predictions (explanations) against features values (contributions), and select representatives for the different explanations clusters.
- **Model Analysis:** We provide a mechanism to analyze where the model works well (true positives and true negatives), and where it does not (false positives and false negatives). We also mine the explanations for interesting patterns, such as features frequent itemsets and rules, *e.g.*, association rules, that can demystify the model decisions.
- **Differential Analysis:** We compare the explanations of two different classifiers with a focus on where their predictions disagree.

¹<https://dcx.cs.purdue.edu>

4.6.3 Overview of EXPLAINER

We give an overview of the typical ER pipeline, and how EXPLAINER weaves in to explain the model and its predictions. Notice that we are not investigating the ER pipeline itself (*e.g.*, blocking and feature selection), but are rather focusing on interpreting its predictions.

Entity Resolution

Let R and R' be two relations with aligned schema $\{A_1, A_2, \dots, A_m\}$. Furthermore, let $t[A_j]$ be the value of Attribute A_j on Tuple t . Given all distinct tuple pairs $(t, t') \in R \times R'$, ER aims to identify the pairs of tuples that refer to the same real-world entities.

Blocking

Typically, ER solutions first run *blocking* methods that generate a candidate set $C \subseteq R \times R'$ that includes tuple pairs that are likely to match.

Training/Testing Data

Most, if not all, ER solutions need *training/testing data* that can be formalized as follows: A labeled dataset is a set of triplets $L \subseteq R \times R' \times \{0, 1\}$, where Triplet $(t, t', 1)$ (resp. $(t, t', 0)$) denotes that Tuples t and t' are (resp. are not) duplicates.

Explanations for Entity Resolution

While machine learning provides amazing results in many applications, an oft-time reservation against its use is the lack of transparency and understanding of why a decision is made by a given ML algorithm. Thus, explaining ML algorithms has been the subject of intense research activity.

Some models (*e.g.*, Decision Trees and Linear Models) are interpretable, but many other models are harder to understand. To explain a black-box model, model-agnostic

tools either learn an interpretable model on the predictions of the underlying model, or alter the model inputs and monitor its respective reactions.

Some tools focus on how different features contribute to every single instance prediction (*local explanations*), while other tools compute the combined feature importance, or summarize the model as a whole (*global explanations*).

In this chapter, we leverage various general-purpose explanation tools for explaining ML-based ER. These tools have been used to provide local and/or global explanations for several applications in both academic and industrial domains. LIME [68] explains the predictions of any classifier by approximating the classifier locally with an interpretable model for perturbed inputs. It also presents a set of representative instances, selected via submodular optimization, as an explanation of the whole model. Anchors [69] is another tool that targets local explanations by providing explanations based on *if-then* rules (anchors) that sufficiently anchors the prediction locally, such that changes to the rest of the feature values of the instance do not change its predicted class. BRL [70] aims to output global explanations that consist of a series of *if-then-else* statements. These rules discretize the feature space into a series of simple interpretable decision statements. Finally, Skater [71], by datascience.com, is a professional approach towards interpretation for black box models both globally and locally. It uses a combination of algorithms to clarify the relationships between the data a model receives and the output it produces.

While these tools provide local and global explanations, we build upon their collective multiple-granularity explanations to support further use-cases in the ER context. EXPLAINER takes in the tuple pairs, labeled data, features and trained model, and processes the explanations from the underlying tools to output more profound analyses. An overview of the ER pipeline and EXPLAINER architecture is given in Figure 4.4.

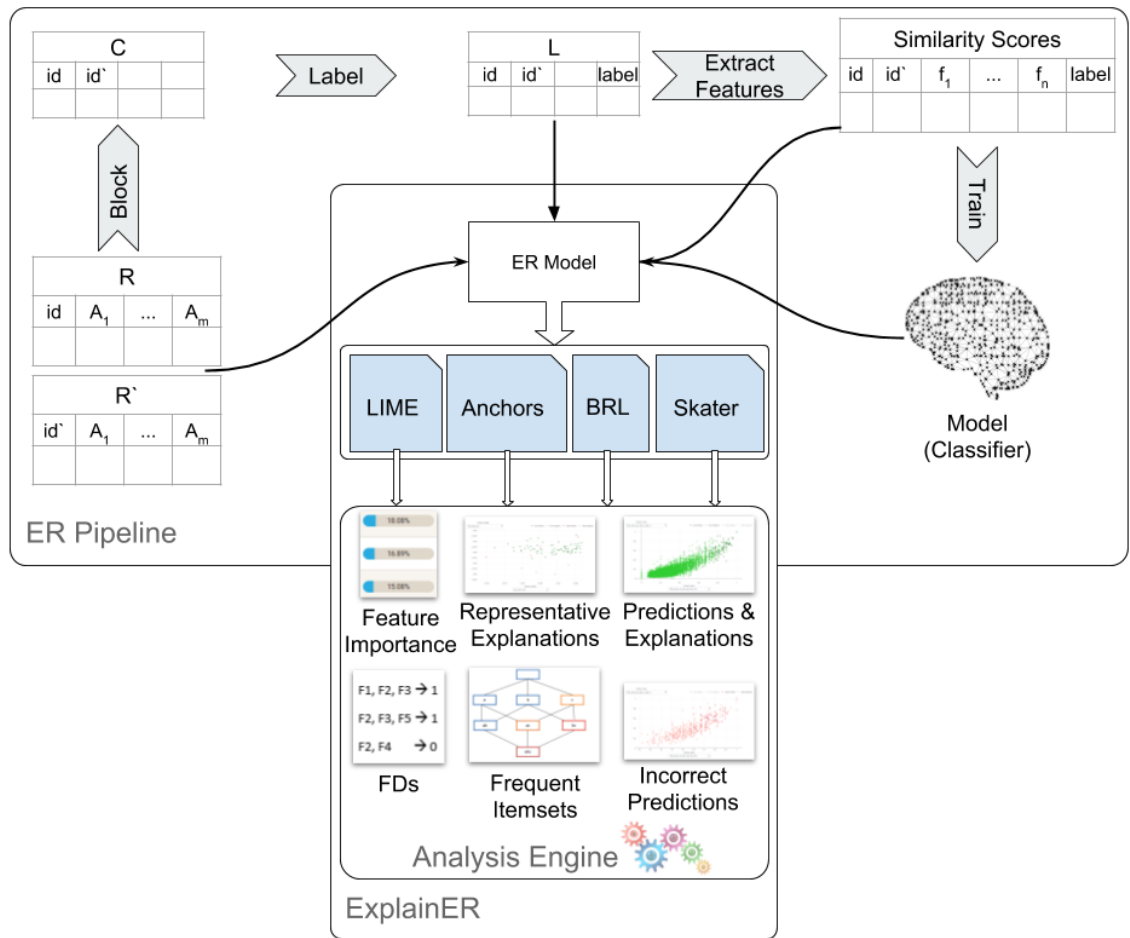


Figure 4.4. Entity Resolution Pipeline and EXPLAINER Architecture

4.6.4 Case Studies

We showcase EXPLAINER in action. The users can choose among various datasets and classifiers, for which EXPLAINER provides explanations at multiple granularities. These help users understand the dataset and the classifier, figure out the important features and fine-tune them, and inspect when and why the model performs badly and address the model's shortcomings.

We experiment on several multi-domain benchmark datasets [72, 73] that have been frequently used in ER research, along with different classifiers from Magellan [65]. Since EXPLAINER deals with classifiers as black-boxes and does not assume any knowledge of the underlying models, it can be easily extended to work with others without loss of generality.

We highlight the following scenarios and case studies: (1) *Global Explanations* to present a global interpretation of an ER model via post-processing the instance-level explanations; (2) *Model Analysis* to provide a deeper understanding, and highlight when the model works well and when it does not; and (3) *Differential Analysis* to compare two different classifiers with a focus on where they differ.

Global Explanations Several frameworks provide local explanations for predictions on instance-level. Our goal is to provide the end-user with a global understanding of the ER model as a whole. To this end, EXPLAINER uses different channels to communicate various aspects and properties of the model to the end user:




Name	Left Attribute	Right Attribute	Similarity Function	Importance
year_year_exm	year	year	Exact Match	 18.08%
title_title_mel	title	title	Monge-Elkan Algorithm	 16.89%
year_year_lev_dist	year	year	Levenshtein Distance	 15.08%

Figure 4.5. Global Explanations in EXPLAINER - Feature Importance

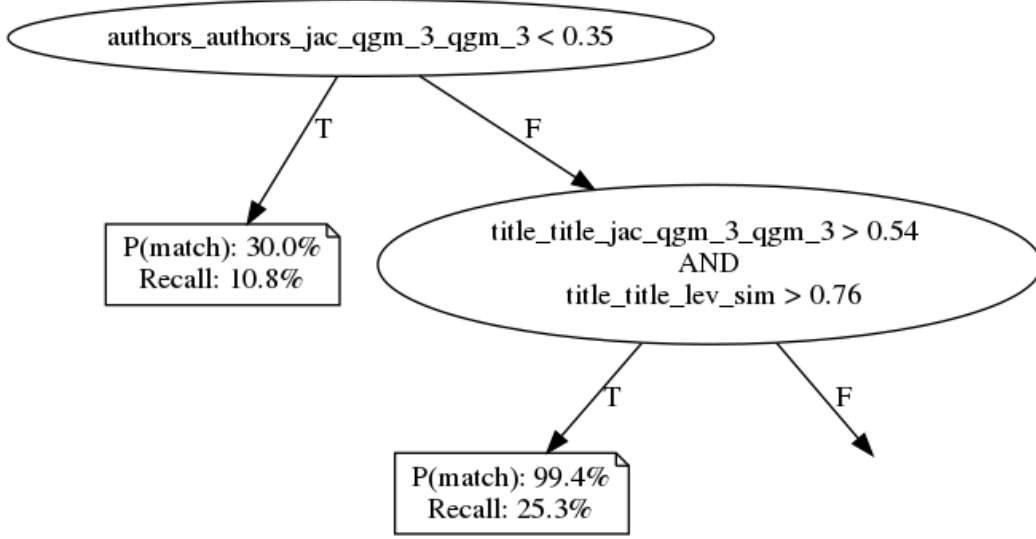
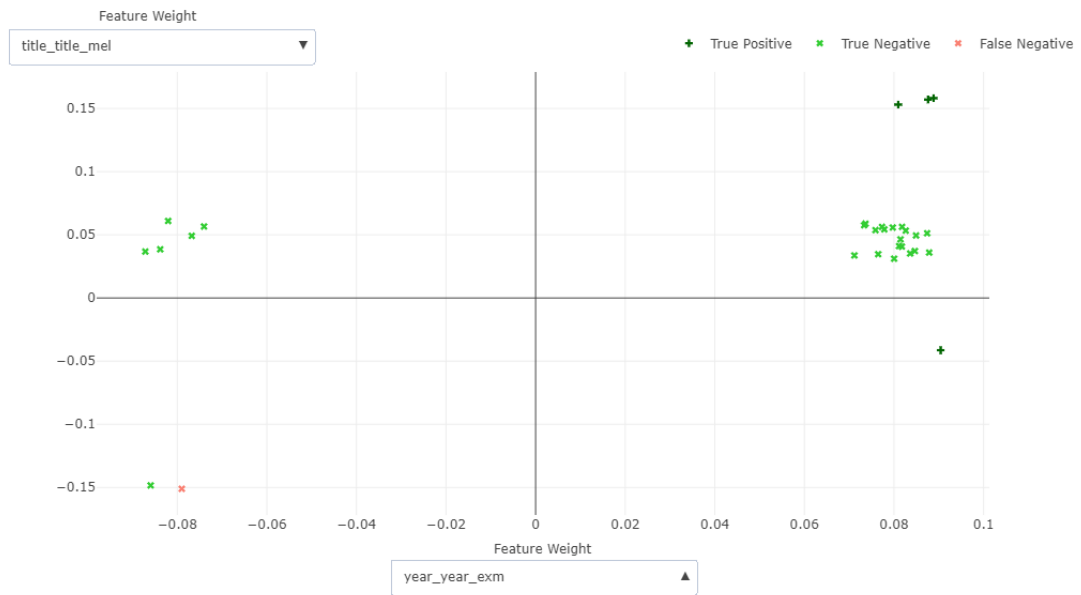


Figure 4.6. Global Explanations in EXPLAINER - Bayesian Rule List (BRL)

- *Feature Importance* provided by Skater [71] or derived from feature contributions over all local explanations (Figure 4.5). This helps understand how significant each feature is, and how it affects the model decisions.
- Approximating the model as a BRL [70] (Figure 4.6). This helps compare features' importance, and visualize how each guides the model through the feature space.
- Plotting all predictions (training and testing data) against feature values to visualize different clusters, and inspect predictions in each cluster.
- Plotting all local explanations against feature weights (Figure 4.7(a)) to visualize how effectively different features can distinguish matches from non-matches.
- Choosing a set of *representative explanations* as a summary of all explanations (Figure 4.7(b)), as an attempt to globally understand the classification model. While SP-LIME [68] calculates top- K diverse explanations, it assumes equal contributions for features, and does not distinguish between important and



(a) Feature Weights for All Explanations



(b) Feature Weights for Representative Explanations

Figure 4.7. Global Explanations in EXPLAINER - Feature Weights for Explanations

unimportant ones. We propose to use the K-Medoids algorithm [74] in order to take feature weights into consideration as well. The K-Medoids algorithm chooses representatives among the explanations, in an attempt to minimize the distance (in feature weights) between those in the same cluster.

The above system aspects and properties can help the non-expert user have an overall perspective of the ER model, and identify important features and their corresponding contributions to the classifier. Moreover, they allow experts to carry out feature engineering, and proceed further with improving and fine-tuning the model.

Model Analysis

The goal is to have a deeper understanding of the model and features via analysis on the explanations. Some of the underlying tools provide local explanations for individual predictions, but we exploit these further to construct more advanced informed interpretations:

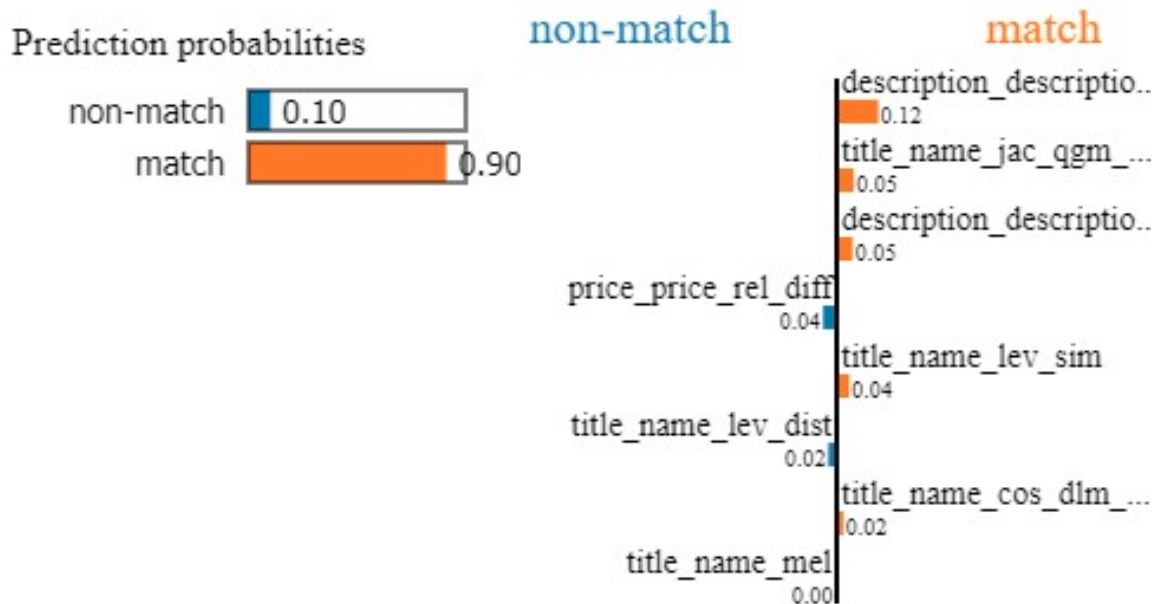


Figure 4.8. Model Analysis in EXPLAINER - Local Explanation by LIME

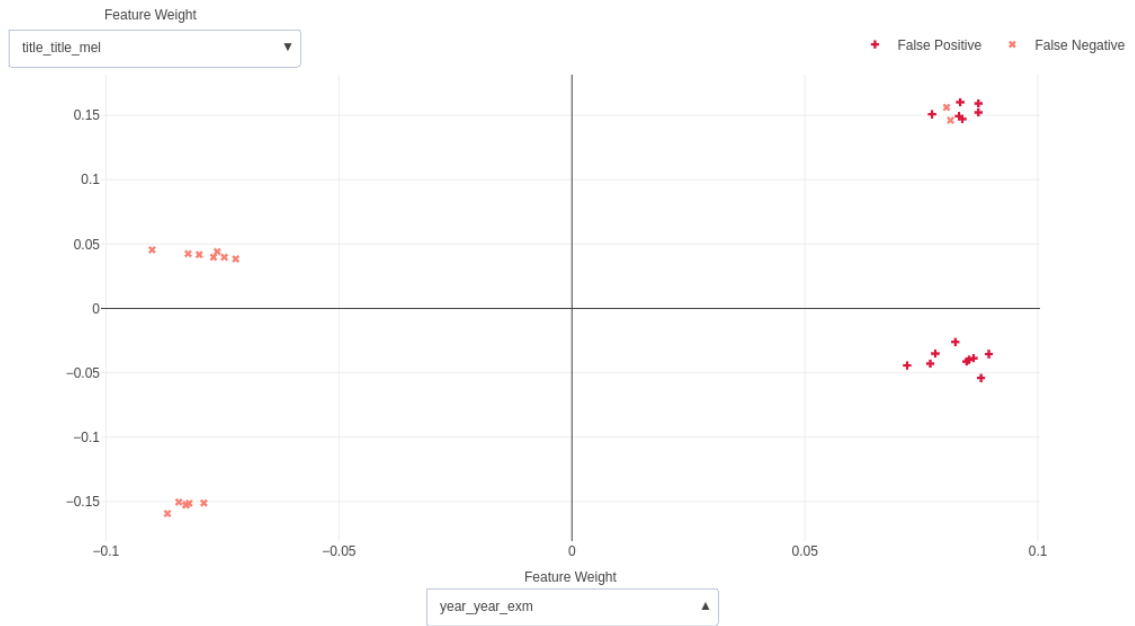


Figure 4.9. Model Analysis in EXPLAINER - Incorrect Predictions

Feature Itemset	Support
year_year_anm + authors_authors_mel + title_title_cos_dlm_dc0_dlm_dc0	92.34%
authors_authors_cos_dlm_dc0_dlm_dc0 + year_year_exm + authors_authors_mel	86.23%
authors_authors_jac_qgm_3_qgm_3 + year_year_exm	79.15%

Figure 4.10. Model Analysis in EXPLAINER - Features Frequent Itemsets

- Inspecting every individual explanation by LIME [68] (Figure 4.8) and Anchors [69]. This can help answer why a specific instance is a match (true positive) or a non-match (true negative), and more interestingly explain erroneous predictions (false positives and false negatives).

- Visualizing representative explanations of incorrect predictions (Figure 4.9), *i.e.*, false positives and false negatives, to highlight where the model fails.
- To highlight possible correlations between features and explain which sets of features contributed together towards a prediction, we propose to mine *frequent itemsets* and *association rules* [75] from explanations formatted as vectors of feature weights (Figure 4.10). Through this mining, we can highlight the correlation between features and explain which sets of features contribute together towards a prediction.
- Mining *FD (CFD) rules* [76] from explanations that are formatted as binary feature vectors to compare against rule-based global explanations and feature importance results.

The above approaches towards batch explanations allow to zoom in beyond local explanations and contrast against global explanations. Additionally, they can help spot and address issues in the dataset itself, *e.g.*, dealing with heterogeneous values in the same attribute or detecting the more serious issue of wrong labels in the dataset.

Differential Analysis

In this case, we have two different models, say $M1$ and $M2$, and the goal is to understand how they compare against each other. We investigate each model's predictions, contrast the two models together, generate explanations for each pair of tuples for both models, and highlight where they disagree.

The user specifies an input dataset and two classifiers, and is then provided with different comparisons of their predictions and explanations (Figure 4.11). Like other scenarios, we can still explain each model from a global perspective (*e.g.*, feature importance, BRL and representative explanations), or look closely at the explanations and the model analysis (*e.g.*, local explanations, incorrect predictions and features frequent itemsets), but more importantly, we can easily inspect these predictions where the two models agree and where they disagree (Figure 4.12).

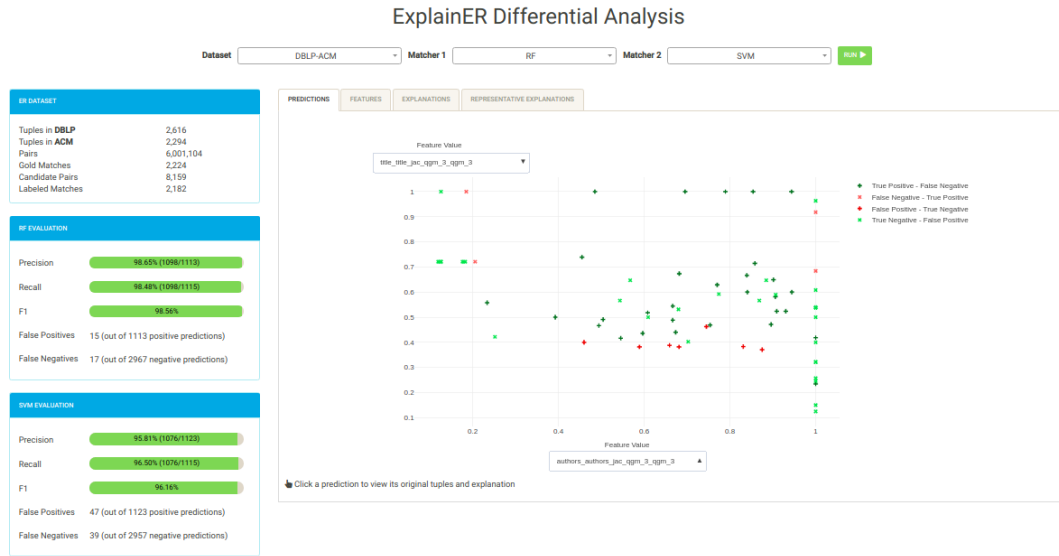


Figure 4.11. Differential Analysis in EXPLAINER - User Interface

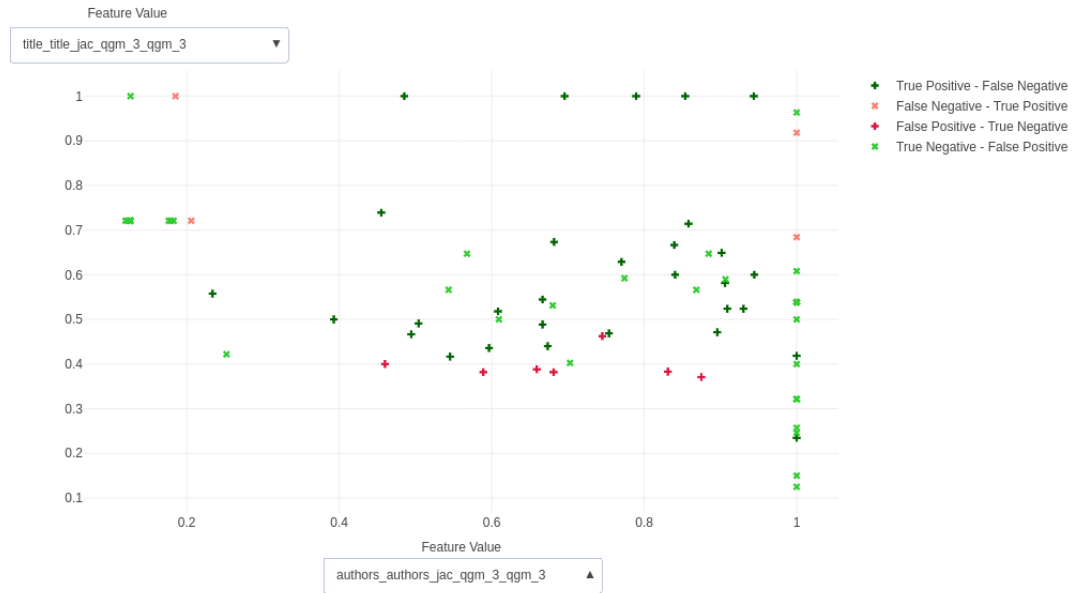


Figure 4.12. Differential Analysis in EXPLAINER - Predictions Disagreement

4.7 Concluding Remarks

In this chapter, we highlight how involving the human-in-the-loop mandates the need for explainability, and discuss the challenges arising from that involvement. We investigate explainability in state-of-the-art data integration systems, explore how they can be classified *w.r.t.* providing explanations, and discuss various properties of those explanations.

As for our dissertation proposal, we show how NADEEF can be categorized as an *explaining* data cleaning system that provides explicit causal explanations for its detected violations and selected data repairs.

We further address explainability beyond rule-based data cleaning and introduce EXPLAINER, a tool to study ER classifiers as a use-case of unexplainable data-cleaning systems. We showcase how EXPLAINER can explain an ER model in different levels of granularity in order to help the user understand the whole model, analyze its predictions, investigate its strengths and weaknesses, or compare it against other models.

5 CONCLUSIONS

Summary

In this dissertation, we propose NADEEF, an end-to-end **generalized, extensible, continuous** and **explaining** data cleaning system, that allows the user to specify multiple types of data quality rules, offers various algorithms for detecting the errors and cleaning the data, can adapt to data changes, and provides the end-user with user-friendly explanations of its cleaning process via auditing the detected violations and data repairs.

In Chapter 2, we present NADEEF, a *generalized* and *extensible* easy-to-deploy data cleaning system. NADEEF distinguishes between a *programming interface* and a *core* to achieve generality and extensibility. The programming interface can be used to express many types of data-quality rules beyond the well-known FDs, CFDs, MDs and ETL rules, and it allows users to specify data-quality rules by writing code that implements predefined classes. These classes uniformly define *what* is wrong with the data, and (possibly) *how* to fix it. The core algorithms can interleave multiple types of rules to detect and repair data errors holistically. Moreover, users can easily customize NADEEF by defining new types of rules, or by extending the core.

In Chapter 3, we extend our data cleaning system to support continuity, and handle both minor and major data changes. With dynamic evolving data in today’s big-data era, continuous data cleaning is not a luxury, but rather a must to handle use-cases that other traditional data cleaning systems cannot handle, e.g., interactive data cleaning, and cleaning of dynamic data (streams) in which the data is not available as a whole, but arrives in increments. We propose NADEEF+ as a *continuous* data-cleaning solution, that can efficiently and incrementally detect violations in dynamic data and repair it while mending the inevitable side effects.

In Chapter 4, we highlight the need for explainability due to human involvement. We discuss explainability in state-of-the-art data integration systems and different properties of explanations. We point out how NADEEF targets explainability via providing explicit causal explanations for violations and repairs, which places it under the umbrella of scarce *explaining* data-cleaning systems. We further address explainability beyond rule-based data cleaning and presented EXPLAINER, our tool for ER explanations, as a use case of unexplainable data cleaning systems. We showcase how EXPLAINER helps the user understand the whole model, analyze its predictions or compare it against other models.

Other Challenges

When envisioning an end-to-end data cleaning system, one ought to take other important challenges into consideration as well. With human involvement, interactivity is a top priority; users should be able to investigate the data, monitor its health and quality, and explore samples and summaries of the errors and repairs. This calls for more research and future work in tracks such as data profiling, data discovery, data visualization and sampling/summarizing techniques.

Besides, one missing aspect is seeking the wisdom of the crowd through what we call user-assisted data cleaning. Users can be classified and only problems relevant to their knowledge are addressed to them. Classification criteria can include languages, location, age, interests, areas of expertise, work experience, education, etc. This would increase the probability of getting correct answers, increase the users potential to help, and employ users feedback more adequately so that such a valuable resource is not wasted in vain. This drives more research in the areas of crowdsourcing, user profiling and expertise matching.

Scalability is another necessary aspect. A modern data cleaning system has to scale to real world big data with millions, if not billions, of tuples. With cloud computing on the rise in several applications, this triggers a lot of interest in the study of distributed and parallel environments, data partitioning and communication cost optimizations in the realm of data quality and data cleaning.

Open Research Problems

Although addressing these other challenges has been out of the scope of this dissertation, we have designed and prototyped NADEEF as an extensible data cleaning system that can allow replacing its core detection and repairing modules to support different use cases, including those we have just touched upon but have not investigated their details in the scope of this dissertation.

Several extensions make for future work that might be of much interest to other researchers or students seeking promising ideas in data quality and data cleaning:

- Involving the human-in-the-loop; users can submit feedback to guide the repairing process or validate the decisions made by the system core and modules. This would require smart techniques to summarize and visualize the data, violations and repairs.
- For user-assisted data cleaning, NADEEF needs to be modified into a multi-user framework. It should profile its users, and classify their areas of expertise, seek users' help based on their knowledge and edit-history, and also resolve any prospective conflicts among their edits.
- Partitioning a big data-cleaning problem into multiple small ones such that each problem 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, and running NADEEF in a parallel and distributed environment.
- Incorporating various indices and blocking techniques in the framework to efficiently support similarity comparisons, and extending the expressions supported by the `fix()` operator to be more general, *e.g.*, by including inequality.
- To handle large volumes of data, NADEEF can be transformed from memory-based to disk-based, through integration with an open-source database management system (DBMS), *e.g.*, PostgreSQL.

REFERENCES

REFERENCES

- [1] Wayne W Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. *The Data Warehousing Institute*, pages 1–36, 2002.
- [2] Hollis Tibbetts. \$3 trillion problem: Three best practices for today’s dirty data pandemic. <http://hollistibbetts.sys-con.com/node/1975126>, 2011.
- [3] Nikki Swartz. Gartner warns firms of ‘dirty data’. *Information Management Journal*, 2007.
- [4] Colleen Graham. Forecast: Data quality tools, worldwide, 2006-2011. <https://www.gartner.com/doc/507207>, 2007.
- [5] Wenfei Fan, Floris Geerts, Shuai Ma, Nan Tang, and Wenyuan Yu. Data quality problems beyond consistency and deduplication. In *In Search of Elegance in the Theory and Practice of Computation*, pages 237–249. Springer, 2013.
- [6] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment*, 9(12):993–1004, 2016.
- [7] Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 159–170. ACM, 2008.
- [8] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems*, pages 68–79. ACM, 1999.
- [9] Leopoldo Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121, 2011.
- [10] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *Proceedings of the 33rd international conference on Very large data bases*, pages 315–326. VLDB Endowment, 2007.
- [11] Jef Wijsen. Database repairing using updates. *ACM Transactions on Database Systems (TODS)*, 30(3):722–768, 2005.
- [12] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *Proceedings of the VLDB Endowment*, 2(1):407–418, 2009.

- [13] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Interaction between record matching and data repairing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 469–480. ACM, 2011.
- [14] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):6, 2008.
- [15] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of Data*, pages 143–154. ACM, 2005.
- [16] Loreto Bravo, Wenfei Fan, and Shuai Ma. Extending dependencies with conditions. In *Proceedings of the 33rd international conference on Very large data bases*, pages 243–254. VLDB Endowment, 2007.
- [17] Solmaz Kolahi and Laks VS Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*, pages 53–62. ACM, 2009.
- [18] Xu Chu, Ihab F Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *IEEE 29th International Conference on Data Engineering (ICDE)*, pages 458–469. IEEE, 2013.
- [19] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. Eracer: a database approach for statistical inference and data cleaning. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 75–86. ACM, 2010.
- [20] Mohamed Yakout, Ahmed K Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F Ilyas. Guided data repair. *Proceedings of the VLDB Endowment*, 4(5):279–289, 2011.
- [21] Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J Miller. Continuous data cleaning. In *IEEE 30th International Conference on Data Engineering (ICDE)*, pages 244–255. IEEE, 2014.
- [22] Paulo Oliveira, Fátima Rodrigues, and Pedro Henriques. Smartclean: an incremental data cleaning tool. In *9th International Conference on Quality Software*, pages 452–457. IEEE, 2009.
- [23] Xiaolan Wang, Laura Haas, and Alexandra Meliou. Explaining data integration. *Data Engineering*, page 47, 2018.
- [24] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F Ilyas, Mourad Ouzzani, and Nan Tang. Nadeef: a commodity data cleaning system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 541–552. ACM, 2013.
- [25] Amr Ebaid, Ahmed Elmagarmid, Ihab F Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, Nan Tang, and Si Yin. Nadeef: A generalized data cleaning system. *Proceedings of the VLDB Endowment*, 6(12):1218–1221, 2013.

- [26] Vijayshankar Raman and Joseph M Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.
- [27] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *The International Journal on Very Large Data Bases*, 21(2):213–238, 2012.
- [28] Ivan P Fellegi and David Holt. A systematic approach to automatic edit and imputation. *Journal of the American Statistical association*, 71(353):17–35, 1976.
- [29] Monica Scannapieco. *Data Quality: Concepts, Methodologies and Techniques. Data-Centric Systems and Applications*. Springer, 2006.
- [30] Felix Naumann, Alexander Bilke, Jens Bleiholder, and Melanie Weis. Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Eng. Bull.*, 29(2):21–31, 2006.
- [31] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering*, 19(1):1–16, 2007.
- [32] Mauricio A Hernández and Salvatore J Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data mining and knowledge discovery*, 2(1):9–37, 1998.
- [33] Steven Euijong Whang, Omar Benjelloun, and Hector Garcia-Molina. Generic entity resolution with negative rules. *The VLDB Journal*, 18(6):1261, 2009.
- [34] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. Ajax: an extensible data cleaning tool. In *ACM SIGMOD Record*, volume 29, page 590. ACM, 2000.
- [35] Mohamed G Elfeky, Vassilios S Verykios, and Ahmed K Elmagarmid. Tailor: A record linkage toolbox. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 17–28. IEEE, 2002.
- [36] Gabriel M Kuper. Aggregation in constraint databases. In *PPCP*, volume 93, pages 166–173. Citeseer, 1993.
- [37] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl. Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 53–62. ACM, 2012.
- [38] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE transactions on knowledge and data engineering*, 24(9):1537–1555, 2012.
- [39] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- [40] Yogesh S Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–375. Springer, 2004.

- [41] Mary Jo Waller. *Comparison of Two Incremental Merge/purge Strategies*. PhD thesis, University of Illinois Springfield, 1998.
- [42] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. Progressive duplicate detection. *IEEE Transactions on knowledge and data engineering*, 27(5):1316–1329, 2015.
- [43] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. Incremental record linkage. *Proceedings of the VLDB Endowment*, 7(9):697–708, 2014.
- [44] Hui Xie, Hongzhi Wang, Jianzhong Li, and Hong Gao. A data cleaning framework based on user feedback. In *Web-Age Information Management*, pages 514–520. Springer, 2013.
- [45] Shawn R Jeffery, Gustavo Alonso, Michael J Franklin, Wei Hong, and Jennifer Widom. A pipelined framework for online cleaning of sensor data streams. In *Data Engineering, 2006. ICDE’06. Proceedings of the 22nd International Conference on*, pages 140–140. IEEE, 2006.
- [46] Jacob Holm, Kristian De Lichtenberg, Mikkel Thorup, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.
- [47] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142. Society for Industrial and Applied Mathematics, 2013.
- [48] Mohamed Yakout, Laure Berti-Équille, and Ahmed K Elmagarmid. Don’t be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 553–564. ACM, 2013.
- [49] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proceedings of the VLDB Endowment*, 10(11):1190–1201, 2017.
- [50] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–278. ACM, 2002.
- [51] Arvind Arasu, Michaela Götz, and Raghav Kaushik. On active learning of record matching packages. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 783–794. ACM, 2010.
- [52] Michael J Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. Crowddb: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 61–72. ACM, 2011.
- [53] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1247–1261. ACM, 2015.

- [54] Jiannan Wang, Tim Kraska, Michael J Franklin, and Jianhua Feng. Crowder: crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.
- [55] David Aumüller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 906–908. Acm, 2005.
- [56] Ling Ling Yan, Renée J Miller, Laura M Haas, and Ronald Fagin. Data-driven understanding and refinement of schema mappings. In *ACM SIGMOD Record*, volume 30, pages 485–496. ACM, 2001.
- [57] Yunfan Chen, Lei Chen, and Chen Jason Zhang. Crowdfusion: A crowdsourced approach on data fusion refinement. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 127–130. IEEE, 2017.
- [58] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Halevy, and Pedro Domingos. imap: discovering complex semantic matches between database schemas. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 383–394. ACM, 2004.
- [59] Boris Glavic, Gustavo Alonso, Renée J Miller, and Laura M Haas. Tramp: Understanding the behavior of schema mappings through provenance. *Proceedings of the VLDB Endowment*, 3(1-2):1314–1325, 2010.
- [60] Xin Luna Dong and Divesh Srivastava. Compact explanation of data fusion decisions. In *Proceedings of the 22nd international conference on World Wide Web*, pages 379–390. ACM, 2013.
- [61] Jayant Madhavan, Philip A Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *vldb*, volume 1, pages 49–58, 2001.
- [62] Angelika Kimmig, Alex Memory, Lise Getoor, et al. A collective, probabilistic approach to schema mapping. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 921–932. IEEE, 2017.
- [63] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Integrating conflicting data: the role of source dependence. *Proceedings of the VLDB Endowment*, 2(1):550–561, 2009.
- [64] Peter Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer Science & Business Media, 2012.
- [65] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. Magellan: Toward building entity matching management systems. *Proceedings of the VLDB Endowment*, 9(12):1197–1208, 2016.
- [66] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data*, pages 19–34, 2018.
- [67] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. Distributed representations of tuples for entity resolution. *Proceedings of the VLDB Endowment*, 11(11):1454–1467, 2018.

- [68] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should I trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.
- [69] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *AAAI Conference on Artificial Intelligence*, 2018.
- [70] Benjamin Letham, Cynthia Rudin, Tyler H McCormick, David Madigan, et al. Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model. *The Annals of Applied Statistics*, 9(3):1350–1371, 2015.
- [71] Pramit Choudhary, Aaron Kramer, and contributors datascience.com team. Skater: Model Interpretation Library, March 2018.
- [72] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Benchmark datasets for entity resolution. https://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution.
- [73] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, and Pradap Konda. The magellan data repository. <https://sites.google.com/site/anhaidgroup/useful-stuff/data>.
- [74] Leonard Kaufman and Peter Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.
- [75] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th international conference on Very Large Data Bases, VLDB*, volume 1215, pages 487–499, 1994.
- [76] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):683–698, 2011.

VITA

VITA

Amr Ebaid was born and raised in Damanhur, Egypt. He received his BSc degree in Computer Science from Alexandria University, Egypt in 2007. He received his MSc degree in Computer and Systems Engineering from Alexandria University, Egypt in 2010, with a thesis titled “Bipartite Graph Matching for Graph and Subgraph Isomorphism”.

He then joined the PhD program in Purdue University and has been working on data quality and data cleaning, supervised by Professors Walid Aref, Ahmed Elmagarmid and Mourad Ouzzani. He got an MSc degree in Computer Science from Purdue University in 2018.

He also had multiple internships at different research institutes and corporations: HP Labs 2015, Yelp Inc. 2014, Qatar Computing Research Institute (QCRI) 2012 and 2013. Google Inc. 2009 and 2011.

He has been awarded the Purdue Graduate Teacher Certificate (GTC) for teaching and professional development in 2016, the Excellence Award for Academic Distinction from Alexandria University from 2002 to 2007, and silver and bronze medals in the Egyptian Olympiad in Informatics (EOI) 2005 and 2004.

He had been a Graduate School Global Ambassador in Purdue University from 2013 to 2017, Vice-President of the Egyptian Students Association in Purdue University from 2015 to 2016, Google Student Ambassador in Alexandria University from 2008 to 2009, and Co-Founder and Vice-Chairman of the ACM Student Chapter in Alexandria University from 2005 to 2006.

Publications

- A. Mustafa, **A. Ebaid** and J. Teller. “Benefits of a multiplesolution approach in land change models”. Transactions in GIS 2018.
- A. Mustafa, I. Saadi, **A. Ebaid**, M. Cools and J. Teller. “Comparison among three automated calibration methods for cellular automata land use change model: GA, PSO and MCMC”. AGILE 2018.
- M. A. Zahran and **A. Ebaid**. “Exploiting Textual and Citation Information to Identify and Summarize Influential Publications”. FLAIRS 2018.
- **A. Ebaid**, A. Elmagarmid, I. Ilyas, M. Ouzzani, J. A. Quijane-Ruiz, N. Tang and S. Yin. “NADEEF: A Generalized Data Cleaning System”. VLDB 2013.
- M. Dallachiesa, **A. Ebaid**, A. Eldawy, A. Elmagarmid, I. Ilyas, M. Ouzzani and N. Tang. “NADEEF: a Commodity Data Cleaning System”. SIGMOD 2013.

Under Review

- **A. Ebaid**, S. Thirumuruganathan, W. G. Aref, A. Elmagarmid and M. Ouzzani. “EXPLAINER: Entity Resolution Explanations”. ICDE 2019.