UNIVERSITY OF NICE - SOPHIA ANTIPOLIS
These d'Habilitation a Diriger des Recherches

Presented by Paolo Papotti
Department of Data Science, EURECOM
papotti@eurecom.fr

Declarative Rule Discovery for Detecting and Correcting Inconsistencies in Noisy Data

Submitted in total fulfillment of the requirements
of the degree of Habilitation a Diriger des Recherches

Contents

Chapter 1

Introduction

1.1 Motivation

As collecting structured data becomes easier and cheaper, e.g., with the expanding use of sensors and wider adoption of social networks and personal devices, the number and the scope of data-driven applications increase exponentially, from financial decisions and precision medicine to urban planning [57]. Major research efforts investigate how to improve the accuracy of the models and how to make them able to quantify the uncertainty of a result. These models are at the center of the attention, but their outcomes can be perturbed by errors in the data [26]. This means that data can drastically bias predictions that are undesirable or even dangerous [89], such as navigation systems guiding tourists into lakes [77], misdiagnoses for patients [86], and financial models leading to billions of euros losses [59]. Most of the models are validated only on a handful of manually cleaned datasets [70], but in real applications up to 25% of critical data is flawed [99]. While society increasingly depends on data-driven decisions, less attention has been given to the quality of the data that feed the models, and most users of such models are not even aware of the problem.

By data we mean here structured or semi-structured data, stored in data management systems provided with a query language, rather than unstructured contents. To explain what we mean by "errors", consider table D_d derived from a dataset with environmental data in the EU Open Data Portal. The table is one of the 12K available on this website alone, and is not representative of the real size of the datasets, which can go from thousands to millions of records[1].

Each record reports yearly values for an indicator of the waste generated in a country. Dataset D_d shows examples of quality issues, such as duplicate records (second and third rows), missing values (-), and incorrect values (in bold). Assume someone needs to estimate the

[1]`https://data.europa.eu/euodp/data`

Indicator	*Country*	*Instrument*	*2012*	*2014*	*2016*
KG_C1	EL	**PF4EE**	93	109	119
KG_C1	ES	-	63	-	-
KG_C1	**ESP**	PF4EE	-	57	62
G_P2	EL	NCFF	**79**	102	145

(a) Dataset D_d with errors

Indicator	*Country*	*Instrument*	*2012*	*2014*	*2016*
KG_C1	EL	-	93	109	119
KG_C1	ES	PF4EE	63	*57*	*62*
G_P2	EL	NCFF	*97*	102	145

(b) Cleaned dataset D_c

waste generation for 2018. Since a predictive model f would return incorrect estimates y if executed on D_d, she wants to execute it on the clean dataset D_c, i.e., $y = f(D_c)$. To obtain the clean version, a cleaning program p must be executed on D_d, i.e., $D_c = p(D_d)$. Such a program can be manually written in a procedural language (e.g., with the *pandas* library in Python) or obtained from a data cleaning system.

Data cleaning systems take as input high level *specifications*, such as examples of the desired output or declarative rules, and produce the program p to be executed over the dataset [26]. For D_d, a valid rule states that for every country the indicator "G_P2" must have the highest value in 2012. It can be expressed by the following *denial constraint*:

$$r_1 : \neg(t_1.\mathit{Indicator} = \text{“G\_P2”} \wedge t_1.\mathit{Country} = t_2.\mathit{Country} \wedge t_1.2012 < t_2.2012)$$

where t_1, t_2 are universal variables over the records. If two tuples match the conditions in the rule, then there must be at least one erroneous value in one of the tuples. Another rule captures that only countries with waste generation for "KG_C1" below 100 for year 2014 can benefit from the financial instrument "PF4EE":

$$r_2 : \neg(t.\mathit{Indicator} = \text{“KG\_C1”} \wedge t.\mathit{Instrument} = \text{“PF4EE”} \wedge t.2014 > 100)$$

Data cleaning depends on the quality of rules like r_1 and r_2, but writing such specifications, or the corresponding cleaning programs, is a difficult and time consuming job.

Traditionally, data engineers guarantee the quality of the data consumed by an application, e.g., IT staff in companies clean the customer data warehouse. This process is based on the manual definition of data cleaning programs and takes up anywhere from 60 to 80% of the data engineer's time [73, 62, 32]. In the *big data* era, applications are built on data coming from social media, IoT devices, data lakes, and open datasets. These heterogeneous sources have errors from faulty sensors, automatic extractors, and human mistakes. Datasets are so large and increasing so fast that data engineers have to make decisions on how and what to clean. The consequences reflect the struggle with the quality of the data. Without proper cleaning, erroneous decisions can be taken because of data errors trickling down to the models, or because incomplete information is fed to target applications. As the 2014 Turing Award laureate Michael Stonebraker defines it: "Data quality at scale is the 800 pound gorilla in the corner" [11].

After obtaining my Ph.D. in 2007, my work has been dedicated to the development of techniques that support users in improving the quality of their data, and in particular on methods to mine and use *declarative rules* for data cleaning. Two main challenges must be addressed to make data cleaning a principled and scalable process that can be effectively adopted in all domains.

1. Discovering cleaning specifications. Even for a single table, the number of specifications needed to capture all quality issues is extremely large. Rules alone are usually in the thousands [98, 34]. In the example, rules similar to r_1 should be stated for years 2014 and 2016, more rules like r_2 are needed to handle different instruments (e.g., "NCFF"), and at least another rule is needed to identify other issues, such as merging the duplicate "ES" records into one. In general, for every new dataset, a data engineer has to team up with a domain expert, such as a chemist or a neurologist, to define semantically meaningful specifications. These collaborations further increase the cost and the delay due to cleaning. While rules can be executed to deal with the high *volume* of the data, the *diversity* in the data sources aggravate the specification process [11]. Part of my research activities focused on rule discovery techniques, and in particular on data mining techniques that identify expressive specifications in first order logic. Part I of this thesis is dedicated to my contributions to tackle the challenges arising from mining complex rules from large datasets.

2. Identifying errors and repairing data. Even when the quality rules have been carefully setup, a domain expert is involved again to manually update a large portion of the errors identified by the cleaning systems [3]. In fact, rules are able to detect inconsistencies that involve multiple values, i.e., cells of the database that together forms a *violation*. In the example, rule r_1 can identify a violation across the first and the fourth tuples spanning attributes *Indicator*, *Country*, and *2012*. However, a user is needed to identify value **79** as faulty and to change it to the correct value *97*. Going from a - potentially large - set of cells to the one or more values that are recognized as erroneous is no obvious. It is even more challenging to automatically update such erroneous values so that the specifications are satisfied. The process of *repairing* the data requires a number of user interactions that depends on the size of the data, making the human effort orders of magnitude larger compared to the specification task in the first challenge. I devoted part of my research activities in the last few years to develop techniques that make it possible to reduce the human effort in fixing errors in noisy datasets. Part II of this thesis focuses on such approaches.

1.2 Plan of the thesis

The thesis is divided in two parts.

- In response to the first challenge, Chapter 2 of the thesis deals with methods that I proposed for **Mining of declarative rules**.
- In response to the second challenge, Chapter 3 of the thesis reports some of my work on **Data repairing with declarative rules**.

The papers included in the thesis contain experiments in various real-world and synthetic datasets. The final chapter of the thesis will draw some conclusions and will discuss some of the ongoing and future work.

Chapter 2

Mining of declarative rules

This part of the thesis focuses on the developments that I proposed to identify semantically valid declarative rules from datasets. In particular, these approaches deal with the problem of discovering data quality rules expressed in different fragments of first order logic, namely denial constraints [27], temporal functional dependencies [2], and Horn rules [80]. The work reported in this part of the thesis proposes ways to identify rules that have support in the data without assuming that the data is already clean. This is a key property for our target application (data repair) and allows users to apply the methods without manually crafting clean samples of their datasets.

Resources and grants: These activities involved my time and that of a number of collaborators. In particular, the research on relational data [2, 27] was conducted during my experience as a scientist at the Qatar Computing Research Institute (QCRI) working in collaboration with interns and post-docs under my co-supervision. Support for these activities came from public funding (national Qatar funding body). The work on rule discovery in knowledge graphs [80] has been conducted with PhD students under my supervision and has been supported by a faculty research award from Google during the time I spent as assistant professor at Arizona State University (ASU).

- **Discovering Denial Constraints.** Specifications, including declarative rules, are metadata for the given dataset that need to be cleaned. The main issue is that the user effort to write down specifications is very high as it requires both technical skills in logic or programming and deep understanding and knowledge of the data domain [73, 62, 32]. To assist domain experts in this task, numerous studies have attempted to discover data quality rules [72], such as Functional Dependencies (FDs) [56, 109], Conditional Functional Dependencies (CFDs) [25, 42, 50], and Matching Dependencies (MDs) [94]. The discovery of rules involving lookup over trusted resources subsumes the traditional problems of matching relational tables [76, 63, 75, 79, 85, 14, 113], Web tables [111, 23, 71], and ontologies [37, 33, 97].

Despite these efforts, there is still a big space of rules that cannot be captured by the constraint types above. There is an infinite space of business rules up to ad-hoc programs for enforcing data quality [64]. However, the more expressive power a rule language has, the harder it is to exploit it, for example, in automated data cleaning algorithms, or in writing SQL queries for consistency checking. It is easy to see that a balance should be achieved between the expressive power of a constraint language in order to deal with a broader space of business rules, and at the same time, the restrictions required to ensure adequate static analysis of the rules and the development of effective cleaning and discovery algorithms.

Denial Constraints (DCs) [39], a universally quantified first order logic formalism, serve as a great compromise between expressiveness and complexity for the following reasons:

1. they are defined on predicates that can be easily expressed in SQL queries for consistency checking;
2. they are more expressive than FDs and CFDs and have been proven to be a useful language for data cleaning in many aspects, such as data repairing and consistent query answering;
3. while their static analysis turns out to be undecidable, we show that it is possible to develop a set of sound inference rules and a linear implication testing algorithm for DCs that enable an efficient adoption of DCs as a rule language.

However, DCs also requires an expensive process with experts in the constraint language at hand in consultation with domain experts. To assist domain experts in obtaining useful rules from their data at hand, we give the formal problem definition of discovering DCs and introduce static analysis for them with three sound axioms that serve as the cornerstone for our implication testing algorithm as well as for our DCs discovery algorithm (FASTDC) [27, 29]. To handle datasets that may have data errors, we extend FASTDC to discover approximate constraints, a feature that is clearly crucial for data cleaning in practice.

- **Temporal Rules Discovery for Web Data Cleaning.** Declarative rules, such as denial constraints, are used for several tasks, including cleaning data. To support domain experts in specifying these specifications, algorithms profile the data and expose rules. However, most discovery techniques have traditionally ignored the time dimension. Recurrent events, such as persons reported in locations, have a duration in which they are valid, and this duration should be part of the rules or the cleaning process would simply fail. For example, while the publisher of a certain edition of a novel does not change over time (title, year and edition jointly identify the publisher), some other functional relationships are true only for a certain amount of time, e.g., a person cannot be reported landing in two countries at the same time (the person name identifies the flight destination in a time window of less than one hour), or the same product cannot have different weights reported at the same release date (product model implies its weight in the time window between two releases).

In this work we study the rule discovery problem for temporal web data [2]. Such a discovery process is challenging because of the nature of web data; extracted facts are (i) sparse over time, (ii) reported with delays, and (iii) often reported with errors over the values because of inaccurate sources or non robust extractors. We handle these challenges with a new discovery approach that is more robust to noise. We focus on mining approximate dependencies over noisy dataset with the goal of identifying useful time-windows, or durations, to mine the interval in which a dependency should be considered valid, e.g., a person is not reported traveling to two countries in a 1-hour window. Without such a time dimension, rules are not usable for events. Our solution uses machine learning methods, such as association measures and outlier detection, for the discovery of the rules, together with an aggressive repair of the data in the mining step itself. Our experimental evaluation over real-world data from Recorded Future[1], an intelligence company that monitors over 700K Web sources, shows that automatically discovered temporal rules can improve the quality of the data with an increase of the average precision in the cleaning process from 0.37 to 0.84, and a 40% relative increase in the average F-measure.

- **Robust Discovery of Positive and Negative Rules in Knowledge-Bases.** Graph databases are becoming popular as a flexible data structure to organize information. This is especially common for RDF knowledge-bases (KBs), graph databases that store information in the form of triples, where a predicate instance (an edge) expresses a binary relation between a subject and an object (two nodes). KB triples, called facts, store information about real-world entities and their relationships, such as "Michelle Obama is married to Barack Obama".

  Depending on the application domain and the effort spent in data curation, KBs can have different levels of data quality. It is safe in general to assume that they can be erroneous and incomplete. This is especially common for KBs bootstrapped by extracting information from sources with minimal or no human intervention. As for the web data, false facts are propagated from the sources to the KBs, or introduced by the extractors. In order to identify data quality issues in KBs with declarative rules, we study in this work the discovery of two types of rules [80, 81]. What we call *positive rules* are useful to identify missing facts and therefore enrich the KB by increasing its coverage. On the other hand, we also discover *negative rules*, which, similarly to DCs, are useful to spot logical inconsistencies and identify erroneous triples.

  Unfortunately, there are three main challenges that make this problem hard. First, as for relational data, errors can be present in the graph. More specifically to this setting, usually KBs do not limit the information of interest with a schema, but let

[1]`https://www.recordedfuture.com/`

users add facts defined on new predicates by simply inserting new triples. Third, since closed world assumption (CWA) does no hold in KBs, it is not possible to assume that a missing fact is false, but we rather label it as unknown (open world assumption). Given these challenges, we formally define the problem of rule discovery over erroneous and incomplete KBs. The input of the problem are two sets of positive and negative examples for every predicate, which we generate. In contrast to the traditional ranking of a large set of rules based on a measure of support, our problem definition aims at the identification of a subset of approximate rules, i.e., rules that do not necessarily hold over all the examples, since data errors and incompleteness are in the nature of KBs. The solution is then the smallest set of rules that cover the majority of input positive examples, and as few input negative examples as possible. To enable value comparison, in the spirit of DCs, we discover rules by judiciously using the memory. The algorithm incrementally materializes the KB as a graph, and discovers rules by navigating only the paths that potentially lead to the best rules. By materializing only the portion of the KB that is needed for the promising rules, the disk-access is minimized and the low memory footprint enables the mining with a richer rule language that includes literal values comparisons.

We experimentally tested the resulting rule discovery system[2] over popular and widely used KBs and found that it delivers accurate rules, with a relative increase in average precision by 45% both in the positive and in the negative settings w.r.t. state-of-the-art systems.

[2] `https://github.com/ppapotti/Rudik`

Discovering Denial Constraints

Xu Chu[*]
University of Waterloo
x4chu@uwaterloo.ca

Ihab F. Ilyas
QCRI
ikaldas@qf.org.qa

Paolo Papotti
QCRI
ppapotti@qf.org.qa

ABSTRACT

Integrity constraints (ICs) provide a valuable tool for enforcing correct application semantics. However, designing ICs requires experts and time. Proposals for automatic discovery have been made for some formalisms, such as functional dependencies and their extension conditional functional dependencies. Unfortunately, these dependencies cannot express many common business rules. For example, an American citizen cannot have lower salary and higher tax rate than another citizen in the same state. In this paper, we tackle the challenges of discovering dependencies in a more expressive integrity constraint language, namely Denial Constraints (DCs). DCs are expressive enough to overcome the limits of previous languages and, at the same time, have enough structure to allow efficient discovery and application in several scenarios. We lay out theoretical and practical foundations for DCs, including a set of sound inference rules and a linear algorithm for implication testing. We then develop an efficient instance-driven DC discovery algorithm and propose a novel scoring function to rank DCs for user validation. Using real-world and synthetic datasets, we experimentally evaluate scalability and effectiveness of our solution.

1. INTRODUCTION

As businesses generate and consume data more than ever, enforcing and maintaining the quality of their data assets become critical tasks. One in three business leaders does not trust the information used to make decisions [12], since establishing trust in data becomes a challenge as the variety and the number of sources grow. Therefore, data cleaning is an urgent task towards improving data quality. Integrity constraints (ICs), originally designed to improve the quality of a database schema, have been recently repurposed towards improving the quality of data, either through checking the validity of the data at points of entry, or by cleaning the dirty data at various points during the processing pipeline [10, 13].Traditional types of ICs, such as key constraints, check constraints, functional dependencies (FDs), and their extension conditional functional dependencies (CFDs) have been proposed for data quality management [7]. However, there is still a big space of ICs that cannot be captured by the aforementioned types.

EXAMPLE 1. *Consider the US tax records in Table 1. Each record describes an individual address and tax information with 15 attributes: first and last name (*FN, LN*), gender (*GD*), area code (*AC*), mobile phone number (*PH*), city (*CT*), state (*ST*), zip code (*ZIP*), marital status (*MS*), has children (*CH*), salary (*SAL*), tax rate (*TR*), tax exemption amount if single (*STX*), married (*MTX*), and having children (*CTX*).*

Suppose that the following constraints hold: (1) area code and phone identify a person; (2) two persons with the same zip code live in the same state; (3) a person who lives in Denver lives in Colorado; (4) if two persons live in the same state, the one earning a lower salary has a lower tax rate; and (5) it is not possible to have single tax exemption greater than salary.

Constraints (1), (2), and (3) can be expressed as a key constraint, an FD, and a CFD, respectively.
(1) : $Key\{AC, PH\}$
(2) : $ZIP \rightarrow ST$
(3) : $[CT = \text{'Denver'}] \rightarrow [ST = \text{'CO'}]$
Since Constraints (4) and (5) involve order predicates ($>, <$), and (5) compares different attributes in the same predicate, they cannot be expressed by FDs and CFDs. However, they can be expressed in first-order logic.
$c_4 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$
$c_5 : \forall t_\alpha \in R, \neg(t_\alpha.SAL < t_\alpha.STX)$
Since first-order logic is more expressive, Constraints (1)-(3) can also be expressed as follows:
$c_1 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$
$c_2 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$
$c_3 : \forall t_\alpha \in R, \neg(t_\alpha.CT = \text{'Denver'} \wedge t_\alpha.ST \neq \text{'CO'})$

The more expressive power an IC language has, the harder it is to exploit it, for example, in automated data cleaning algorithms, or in writing SQL queries for consistency checking. There is an infinite space of business rules up to ad-hoc programs for enforcing correct application semantics. It is easy to see that a balance should be achieved between the expressive power of ICs in order to deal with a broader space of business rules, and at the same time, the restrictions required to ensure adequate static analysis of ICs and the development of effective cleaning and discovery algorithms.

Denial Constraints (DCs) [5, 13], a universally quantified first order logic formalism, can express all constraints in Example 1 as they are more expressive than FDs and CFDs. To clarify the connection between DCs and the different classes of ICs we show in

[*]Work done while interning at QCRI.

Proceedings of the VLDB Endowment, Vol. 6, No. 13
Copyright 2013 VLDB Endowment 2150-8097/13/13... $ 10.00.

TID	FN	LN	GD	AC	PH	CT	ST	ZIP	MS	CH	SAL	TR	STX	MTX	CTX
t_1	Mark	Ballin	M	304	232-7667	Anthony	WV	25813	S	Y	5000	3	2000	0	2000
t_2	Chunho	Black	M	719	154-4816	Denver	CO	80290	M	N	60000	4.63	0	0	0
t_3	Annja	Rebizant	F	636	604-2692	Cyrene	MO	64739	M	N	40000	6	0	4200	0
t_4	Annie	Puerta	F	501	378-7304	West Crossett	AR	72045	M	N	85000	7.22	0	40	0
t_5	Anthony	Landram	M	319	150-3642	Gifford	IA	52404	S	Y	15000	2.48	40	0	40
t_6	Mark	Murro	M	970	190-3324	Denver	CO	80251	S	Y	60000	4.63	0	0	0
t_7	Ruby	Billinghurst	F	501	154-4816	Kremlin	AR	72045	M	Y	70000	7	0	35	1000
t_8	Marcelino	Nuth	F	304	540-4707	Kyle	WV	25813	M	N	10000	4	0	0	0

Table 1: Tax data records.

Figure 1 a classification based on two criteria: (i) single tuple level vs table level, and (ii) with constants involved in the constraint vs with only column variables. DCs are expressive enough to cover interesting ICs in each quadrant. DCs serve as a great compromise between expressiveness and complexity for the following reasons: (1) they are defined on predicates that can be easily expressed in SQL queries for consistency checking; (2) they have been proven to be a useful language for data cleaning in many aspects, such as data repairing [10], consistent query answering [5], and expressing data currency rules [13]; and (3) while their static analysis turns out to be undecidable [3], we show that it is possible to develop a set of sound inference rules and a linear implication testing algorithm for DCs that enable an efficient adoption of DCs as an IC language, as we show in this paper.

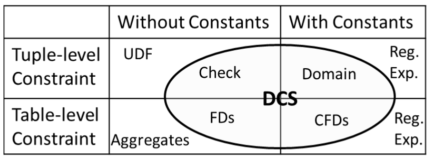


Figure 1: The ICs quadrant.

While DCs can be obtained through consultation with domain experts, it is an expensive process and requires expertise in the constraint language at hand as shown in the experiments. We identified three challenges that hinder the adoption of DCs as an efficient IC language and in discovering DCs from an input data instance:
(1) Theoretical Foundation. The necessary theoretical foundations for DCs as a constraint language are missing [13]. Armstrong Axioms and their extensions are at the core of state-of-the-art algorithms for inferring FDs and CFDs [15, 17], but there is no similar foundation for the design of tractable DCs discovery algorithms.

EXAMPLE 2. *Consider the following constraint, c_6, which states that there cannot exist two persons who live in the same zip code and one person has a lower salary and higher tax rate.*

$$c_6 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$$

c_6 is implied by c_2 and c_4: if two persons live in the same zip code, by c_2 they would live in the same state and by c_4 one person cannot earn less and have higher tax rate in the same state.

In order to systematically identify implied DCs (such as c_6), for example, to prune redundant DCs, a reasoning system is needed.
(2) Space Explosion. Consider FDs discovery on schema R, let $|R| = m$. Taking an attribute as the right hand side of an FD, any subset of remaining $m - 1$ attributes could serve as the left hand side. Thus, the space to be explored for FDs discovery is $m*2^{m-1}$. Consider discovering DCs involving at most two tuples without constants; a predicate space needs to be defined, upon which the space of DCs is defined. The structure of a predicate consists of two different attributes and one operator. Given two tuples, we have $2m$ distinct cells; and we allow six operators $(=, \neq, >, \leq, <, \geq)$. Thus the size of the predicate space **P** is: $|\mathbf{P}| = 6 * 2m * (2m - 1)$. Any subset of the predicate space could constitute a DC. Therefore, the search space for DCs discovery is of size $2^{|\mathbf{P}|}$.

DCs discovery has a much larger space to explore, further justifying the need for a reasoning mechanism to enable efficient pruning, as well as the need for an efficient discovery algorithm. The problem is further complicated by allowing constants in the DCs.
(3) Verification. Since the quality of ICs is crucial for data quality, discovered ICs are usually verified by domain experts for their validity. Model discovery algorithms suffer from the problem of overfitting [6]; ICs found on the input instance I of schema R may not hold on future data of R. This happens also for DCs discovery.

EXAMPLE 3. *Consider DC c_7 on Table 1, which states that first name determines gender.*

$$c_7 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.FN = t_\beta.FN \wedge t_\alpha.GD \neq t_\beta.GD)$$

Even if c_7 is true on current data, common knowledge suggests that it does not hold in general.

Statistical measures have been proposed to rank the constraints and assist the verification step for specific cases. For CFDs it is possible to count the number of tuples that match their tableaux [8]. Similar support measures are used for association rules [2].

Unfortunately, discovered DCs are more difficult to verify and rank than previous formalisms for three reasons: (1) similarly to FDs, in general it is not possible to just count constants to measure support; (2) given the explosion of the space, the number of discovered DCs is much larger than the size of discovered FDs; (3) the semantics of FDs/CFDs is much easier to understand compared to DCs. A novel and general measure of interestingness for DCs is therefore needed to rank discovered constraints.

Contributions. Given the DCs discovery problem and the above challenges, we make the following three contributions:

1. We give the formal problem definition of discovering DCs (Section 3). We introduce static analysis for DCs with three sound axioms that serve as the cornerstone for our implication testing algorithm as well as for our DCs discovery algorithm (Section 4).

2. We present FASTDC, a DCs discovery algorithm (Section 5). FASTDC starts by building a predicate space and calculates evidence sets for it. We establish the connection between discovering minimal DCs and finding minimal set covers for evidence sets. We employ depth-first search strategy for finding minimal set covers and use DC axioms for branch pruning. To handle datasets that may have data errors, we extend FASTDC to discover approximate constraints. Finally, we further extend it to discover DCs involving constant values.

3. We propose a novel scoring function, the *interestingness* of a DC, which combines succinctness and coverage measures of discovered DCs in order to enable their ranking and pruning based on thresholds, thus reducing the cognitive burden for human verification (Section 6).

We experimentally verify our techniques on real-life and synthetic data (Section 7). We show that FASTDC is bound by the number of tuples $|I|$ and by the number of DCs $|\Sigma|$, and that the polynomial part w.r.t. $|I|$ can be parallelized. We show that the implication test substantially reduces the number of DCs in the output, thus reducing users' effort in verifying DCs. We also verify how effective our scoring function is at identifying interesting constraints.

2. RELATED WORK

Our work finds similarities with several bodies of work: static analysis of ICs, dependency discovery, and scoring of ICs.

Whenever a dependency language is proposed, the static analysis should be investigated.Static analysis for FDs has been laid out long ago [1], in which it is shown that static analysis for FDs can be done in linear time w.r.t. the number of FDs and three inference rules are proven to be sound and complete. Conditional functional dependencies were first proposed by Bohannon et al. [7], where implication and consistency problems were shown to be intractable. In addition, a set of sound and complete inference rules were also provided, which were later simplified by Fan [14]. Though denial constraints have been used for data cleaning as well as consistent query answering [5, 10], static analysis has been done only for special fragments, such as currency rules [13].

In the context of constraints discovery, FDs attracted the most attention and whose methodologies can be divided into schema-driven and instance-driven approaches. TANE is a representative for the schema-driven approach [17]. It adopts a level-wise candidate generation and pruning strategy and relies on a linear algorithm for checking the validity of FDs. TANE is sensitive to the size of the schema. FASTFD is a an instance-driven approach [19], which first computes agree-sets from data, then adopts a heuristic-driven depth-first search algorithm to search for covers of agree-sets. FASTFD is sensitive to the size of the instance. Both algorithms were extended in [15] for discovering CFDs. CFDs discovery is also studied in [8], which not only is able to discover exact CFDs but also outputs approximate CFDs and dirty values for approximate CFDs, and in [16], which focuses on generating a near-optimal tableaux assuming an embedded FD is provided. The lack of an efficient DCs validity checking algorithm makes the schema-driven approach for DCs discovery infeasible. Therefore, we extend FASTFD for DCs discovery.

Another aspect of discovering ICs is to measure the importance of ICs according to a scoring function. In FDs discovery, Ilyas et al. examined the statistical correlations for each column pair to discover soft FDs [18]. In CFDs discovery some measures have been proposed, including support, which is defined as the percentage of the tuples in the data that match the pattern tableaux, conviction, and χ^2 test [8, 15]. Our scoring function identifies two principles that are widely used in data mining, and combines them into a unified function, which is fundamentally different from previous scoring functions for discovered ICs.

3. DENIAL CONSTRAINTS AND DISCOVERY PROBLEM

In this section, we first review the syntax and semantics of DCs. Then, we define minimal DCs and state their discovery problem.

3.1 Denial Constraints (DCs)

Syntax. Consider a database schema of the form $\mathbb{S} = (\mathbb{U}, \mathbb{R}, \mathbb{B})$, where $\mathbb{U}$ is a set of database domains, $\mathbb{R}$ is a set of database predicates or relations, and $\mathbb{B}$ is a set of finite built-in operators. In this paper, $\mathbb{B} = \{=, <, >, \neq, \leq, \geq\}$. $\mathbb{B}$ must be *negation closed*, such that we could define the *inverse* of operator ϕ as $\overline{\phi}$.

We support the subset of integrity constraints identified by *denial constraints* (DCs) over relational databases. We introduce a notation for DCs of the form $\varphi : \forall t_\alpha, t_\beta, t_\gamma, \ldots \in R, \urcorner(P_1 \wedge \ldots \wedge P_m)$, where P_i is of the form $v_1 \phi v_2$ or $v_1 \phi c$ with $v_1, v_2 \in t_x.A, x \in \{\alpha, \beta, \gamma, \ldots\}, A \in R$, and c is a constant. For simplicity, we assume there is only one relation R in $\mathbb{R}$.

For a DC φ, if $\forall P_i, i \in [1, m]$ is of the form $v_1 \phi v_2$, then we call such DC *variable denial constraint* (VDC), otherwise, φ is a *constant denial constraint* (CDC).

The *inverse* of predicate P: $v_1 \phi_1 v_2$ is $\overline{P}$: $v_1 \phi_2 v_2$,with $\phi_2 = \overline{\phi_1}$. If P is true, then $\overline{P}$ is false. The set of *implied* predicates of P is $Imp(P) = \{Q | Q : v_1 \phi_2 v_2\}$, where $\phi_2 \in Imp(\phi_1)$. If P is true, then $\forall Q \in Imp(P)$, Q is true. The inverse and implication of the six operators in $\mathbb{B}$ is summarized in Table 2.

ϕ	$=$	$\neq$	$>$	$<$	$\geq$	$\leq$
$\overline{\phi}$	$\neq$	$=$	$\leq$	$\geq$	$<$	$>$
$Imp(\phi)$	$=, \geq, \leq$	$\neq$	$>, \geq, \neq$	$<, \leq, \neq$	$\geq$	$\leq$

Table 2: Operator Inverse and Implication.

Semantics. A DC states that all the predicates cannot be true at the same time, otherwise, we have a violation. Single-tuple constraints (such as check constraints), FDs, and CFDs are special cases of unary and binary denial constraints with equality and inequality predicates. Given a database instance I of schema $\mathbb{S}$ and a DC φ, if I satisfies φ, we write $I \models \varphi$, and we say that φ is a *valid DC*. If we have a set of DC Σ, $I \models \Sigma$ if and only if $\forall \varphi \in \Sigma, I \models \varphi$.

A set of DCs Σ implies φ, i.e., $\Sigma \models \varphi$, if for every instance I of $\mathbb{S}$, if $I \models \Sigma$, then $I \models \varphi$.

In the context of this paper, we are only interested in DCs with at most two tuples. DCs involving more tuples are less likely in real life, and incur bigger predicate space to search as shown in Section 5. The universal quantifier for DCs with at most two tuples are $\forall t_\alpha, t_\beta$. We will omit universal quantifiers hereafter.

3.2 Problem Definition

Trivial, Symmetric, and Minimal DC. A DC $\urcorner(P_1 \wedge \ldots \wedge P_n)$ is said to be *trivial* if it is satisfied by any instance. In the sequel, we only consider nontrivial DCs unless otherwise specified. The *symmetric* DC of a DC φ_1 is a DC φ_2 by substituting t_α with t_β, and t_β with t_α. If φ_1 and φ_2 are symmetric, then $\varphi_1 \models \varphi_2$ and $\varphi_2 \models \varphi_1$. A DC φ_1 is *set-minimal*, or *minimal*, if there does not exist φ_2, s.t. $I \models \varphi_1, I \models \varphi_2$, and $\varphi_2.Pres \subset \varphi_1.Pres$. We use $\varphi.Pres$ to denote the set of predicates in DC φ.

EXAMPLE 4. *Consider three additional DCs for Table 1.*
$c_8 : \urcorner(t_\alpha.SAL = t_\beta.SAL \wedge t_\alpha.SAL > t_\beta.SAL)$
$c_9 : \urcorner(t_\alpha.PH = t_\beta.PH)$
$c_{10} : \urcorner(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL > t_\beta.SAL \wedge t_\alpha.TR < t_\beta.TR)$
c_8 is a trivial DC, since there cannot exist two persons that have the same salary, and one's salary is greater than the other. If we remove tuple t_7 in Table 1, c_9 becomes a valid DC, making c_1 no longer minimal. c_{10} and c_4 are symmetric DCs.

Problem Statement. Given a relational schema R and an instance I, the *discovery problem* for DCs is to find all valid minimal DCs that hold on I. Since the number of DCs that hold on a dataset is usually very big, we also study the problem of ranking DCs with an objective function described in Section 6.

4. STATIC ANALYSIS OF DCS

Since DCs subsume FDs and CFDs, it is natural to ask whether we can perform reasoning the same way. An inference system for DCs enables pruning in a discovery algorithm. Similarly, an implication test is required to reduce the number of DCs in the output.

4.1 Inference System

Armstrong Axioms are the fundamental building blocks for implication analysis for FDs [1]. We present three symbolic inference rules for DCs, denoted as $\mathcal{I}$, analogous to such Axioms.

Triviality: $\forall P_i, P_j$, if $\overline{P_i} \in Imp(P_j)$, then $\neg(P_i \wedge P_j)$ is a trivial DC.

Augmentation: If $\neg(P_1 \wedge \ldots \wedge P_n)$ is a valid DC, then $\neg(P_1 \wedge \ldots \wedge P_n \wedge Q)$ is also a valid DC.

Transitivity: If $\neg(P_1 \wedge \ldots \wedge P_n \wedge Q_1)$ and $\neg(R_1 \wedge \ldots \wedge R_m \wedge Q_2)$ are valid DCs, and $Q_2 \in Imp(\overline{Q_1})$, then $\neg(P_1 \wedge \ldots \wedge P_n \wedge R_1 \wedge \ldots \wedge R_m)$ is also a valid DC.

Triviality states that, if a DC has two predicates that cannot be true at the same time ($\overline{P_i} \in Imp(P_j)$), then the DC is trivially satisfied. Augmentation states that, if a DC is valid, adding more predicates will always result in a valid DC. Transitivity states, that if there are two DCs and two predicates (one in each DC) that cannot be false at the same time ($Q_2 \in Imp(\overline{Q_1})$), then merging two DCs plus removing those two predicates will result in a valid DC.

Inference system $\mathcal{I}$ is a syntactic way of checking whether a set of DCs Σ implies a DC φ. It is sound in that if by using $\mathcal{I}$ a DC φ can be derived from Σ, i.e., $\Sigma \vdash_{\mathcal{I}} \varphi$, then Σ implies φ, i.e., $\Sigma \models \varphi$. The completeness of $\mathcal{I}$ dictates that if $\Sigma \models \varphi$, then $\Sigma \vdash_{\mathcal{I}} \varphi$. We identify a specific form of DCs, for which $\mathcal{I}$ is complete. The specific form requires that each predicate of a DC is defined on two tuples and on the same attribute, and that all predicates must have the same operator θ except one that must have the reverse of θ.

THEOREM 1. *The inference system $\mathcal{I}$ is sound. It is also complete for VDCs of the form $\forall t_\alpha, t_\beta \in R, \neg(P_1 \wedge \ldots \wedge P_m \wedge Q)$, where $P_i = t_\alpha.A_i \theta t_\beta.A_i, \forall i \in [1, m]$ and $Q = t_\alpha.B\overline{\theta}t_\beta.B$ with $A_i, B \in \mathbb{U}$.*

Formal proof for Theorem 1 is reported in the extended version of this paper [9]. The completeness result of $\mathcal{I}$ for that form of DCs generalizes the completeness result of Armstrong Axioms for FDs. In particular, FDs adhere to the form with θ being $=$. The partial completeness result for the inference system has no implication on the completeness of the discovery algorithms described in Section 5. We will discuss in the experiments how, although not complete, the inference system $\mathcal{I}$ has a huge impact on the pruning power of the implication test and on the FASTDC algorithm.

4.2 Implication Problem

Implication testing refers to the problem of determining whether a set of DCs Σ implies another DC φ. It has been established that the complexity of the implication testing problem for DCs is coNP-Complete [3]. Given the intractability result, we have devised a linear, sound, but not complete, algorithm for implication testing to reduce the number of DCs in the discovery algorithm output.

In order to devise an efficient implication testing algorithm, we define the concept of *closure* in Definition 1 for a set of predicates **W** under a set of DCs Σ. A predicate P is in the closure if adding $\overline{P}$ to **W** would constitute a DC implied by Σ. It is in spirit similar to the closure of a set of attributes under a set of FDs.

DEFINITION 1. *The closure of a set of predicates **W**, w.r.t. a set of DCs Σ, is a set of predicates, denoted as $Clo_\Sigma(\mathbf{W})$, such that $\forall P \in Clo_\Sigma(\mathbf{W})$, $\Sigma \models \neg(\mathbf{W} \wedge \overline{P})$.*

Algorithm 1 GET PARTIAL CLOSURE:

```
Input: Set of DCs $\Sigma$, Set of Predicates $\mathbf{W}$
Output: Set of predicates called closure of $\mathbf{W}$ under $\Sigma$ : $Clo_\Sigma(\mathbf{W})$
 1: for all $P \in \mathbf{W}$ do
 2:     $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(P)$
 3:     $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(Clo_\Sigma(\mathbf{W}))$
 4: for each $P$, create a list $L_P$ of DCs containing $P$
 5: for each $\varphi$, create a list $L_\varphi$ of predicates not yet in the closure
 6: for all $\varphi \in \Sigma$ do
 7:     for all $P \in \varphi.Pres$ do
 8:         $L_P \leftarrow L_P + \varphi$
 9: for all $P \notin Clo_\Sigma(\mathbf{W})$ do
10:     for all $\varphi \in L_P$ do
11:         $L_\varphi \leftarrow L_\varphi + P$
12: create a queue $J$ of DC with all but one predicate in the closure
13: for all $\varphi \in \Sigma$ do
14:     if $|L_\varphi| = 1$ then
15:         $J \leftarrow J + \varphi$
16: while $|J| > 0$ do
17:     $\varphi \leftarrow J.pop()$
18:     $P \leftarrow L_\varphi.pop()$
19:     for all $Q \in Imp(\overline{P})$ do
20:         for all $\varphi \in L_Q$ do
21:             $L_\varphi \leftarrow L_\varphi - Q$
22:             if $|L_\varphi| = 1$ then
23:                 $J \leftarrow J + \varphi$
24:     $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(\overline{P})$
25:     $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(Clo_\Sigma(\mathbf{W}))$
26: return $Clo_\Sigma(\mathbf{W})$
```

Algorithm 1 calculates the partial closure of **W** under Σ, whose proof of correctness is provided in [9]. We initialize $Clo_\Sigma(\mathbf{W})$ by adding every predicate in **W** and their implied predicates due to Axiom Triviality (Line 1-2). We add additional predicates that are implied by $Clo_\Sigma(\mathbf{W})$ through basic algebraic transitivity (Line 3). The closure is enlarged if there exists a DC φ in Σ such that all but one predicates in φ are in the closure (Line 15-23). We use two lists to keep track of exactly when such condition is met (Line 3-11).

EXAMPLE 5. *Consider Σ=$\{c_1, \ldots, c_5\}$ and **W** = $\{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL\}$.*

The initialization step in Line(1-3) results in $Clo_\Sigma(\mathbf{W}) = \{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL, t_\alpha.SAL \leq t_\beta.SAL\}$. As all predicates but $t_\alpha.ST \neq t_\beta.ST$ of c_2 are in the closure, we add the implied predicates of the reverse of $t_\alpha.ST \neq t_\beta.ST$ to it and $Clo_\Sigma(\mathbf{W}) = \{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL, t_\alpha.SAL \leq t_\beta.SAL, t_\alpha.ST = t_\beta.ST\}$. As all predicates but $t_\alpha.TR > t_\beta.TR$ of c_4 are in the closure (Line 22), we add the implied predicates of its reverse, $Clo_\Sigma(\mathbf{W}) = \{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL, t_\alpha.SAL \leq t_\beta.SAL, t_\alpha.TR \leq t_\beta.TR\}$. No more DCs are in the queue (Line 16).

Since $t_\alpha.TR \leq t_\beta.TR \in Clo_\Sigma(\mathbf{W})$, we have $\Sigma \models \neg(\mathbf{W} \wedge t_\alpha.TR > t_\beta.TR)$, i.e., $\Sigma \models c_6$.

Algorithm 2 tests whether a DC φ is implied by a set of DCs Σ, by computing the closure of $\varphi.Pres$ in φ under Γ, which is Σ enlarged with symmetric DCs. If there exists a DC ϕ in Γ, whose predicates are a subset of the closure, φ is implied by Σ. The proof of soundness of Algorithm 2 is in [9], which also shows a counterexample where φ is implied by Σ, but Algorithm 2 fails.

EXAMPLE 6. *Consider a database with two numerical columns, High (H) and Low (L). Consider two DCs c_{11}, c_{12}.*

Algorithm 2 IMPLICATION TESTING

```
Input: Set of DCs $\Sigma$, one DC $\varphi$
Output: A boolean value, indicating whether $\Sigma \models \varphi$
1: if $\varphi$ is a trivial DC then
2:     return true
3: $\Gamma \leftarrow \Sigma$
4: for $\phi \in \Sigma$ do
5:     $\Gamma \leftarrow \Gamma +$ symmetric DC of $\phi$
6: $Clo_\Gamma(\varphi.Pres) = getClosure(\varphi.Pres, \Gamma)$
7: if $\exists \phi \in \Gamma$, s.t. $\phi.Pres \subseteq Clo_\Gamma(\varphi.Pres)$ then
8:     return true
```

$c_{11} : \forall t_\alpha, (t_\alpha.H < t_\alpha.L)$
$c_{12} : \forall t_\alpha, t_\beta, (t_\alpha.H > t_\beta.H \wedge t_\beta.L > t_\alpha.H)$

*Algorithm 2 identifies that c_{11} implies c_{12}. Let $\Sigma = \{c_{11}\}$ and **W** = $c_{12}.Pres$. $\Gamma = \{c_{11}, c_{13}\}$, where c_{13}: $\forall t_\beta, (t_\beta.H < t_\beta.L)$. $Clo_\Gamma(\boldsymbol{W}) = \{t_\alpha.H > t_\beta.H, t_\beta.L > t_\alpha.H, t_\beta.H < t_\beta.L\}$, because $t_\beta.H < t_\beta.L$ is implied by $\{t_\alpha.H > t_\beta.H, t_\beta.L > t_\alpha.H\}$ through basic algebraic transitivity (Line 3).*

Since $c_{13}.Pres \subset Clo_\Gamma(\boldsymbol{W})$, the implication holds.

5. DCS DISCOVERY ALGORITHM

Algorithm 3 describes our procedure for discovering minimal DCs. Since a DC is composed of a set of predicates, we build a predicate space **P** based on schema R (Line 1). Any subset of **P** could be a set of predicates for a DC.

Algorithm 3 FASTDC

```
Input: One relational instance $I$, schema $R$
Output: All minimal DCs $\Sigma$
1: P $\leftarrow$ BUILD PREDICATE SPACE$(I, R)$
2: $Evi_I \leftarrow$ BUILD EVIDENCE SET$(I, \mathbf{P})$
3: MC $\leftarrow$ SEARCH MINIMAL COVERS$(Evi_I, Evi_I, \emptyset, >_{init}, \emptyset)$
4: for all X $\in$ MC do
5:     $\Sigma \leftarrow \Sigma + \urcorner(\overline{\mathbf{X}})$
6: for all $\varphi \in \Sigma$ do
7:     if $\Sigma - \varphi \models \varphi$ then
8:         remove $\varphi$ from $\Sigma$
```

Given **P**, the space of candidate DCs is of size $2^{|\mathbf{P}|}$. It is not feasible to validate each candidate DC directly over I, due to the quadratic complexity of checking all tuple pairs. For this reason, we extract evidence from I in a way that enables the reduction of DCs discovery to a search problem that computes valid minimal DCs without checking each candidate DC individually.

The evidence is composed of sets of satisfied predicates in **P**, one set for every pair of tuples (Line 2). For example, assume two satisfied predicates for one tuple pair: $t_\alpha.A = t_\beta.A$ and $t_\alpha.B = t_\beta.B$. We use the set of satisfied predicates to derive the valid DCs that do not violate this tuple pair. In the example, two sample DCs that hold on that tuple pair are $\urcorner(t_\alpha.A \neq t_\beta.A)$ and $\urcorner(t_\alpha.A = t_\beta.A \wedge t_\alpha.B \neq t_\beta.B)$. Let Evi_I be the sets of satisfied predicates for all pairs of tuples, deriving valid minimal DCs for I corresponds to finding the minimal sets of predicates that cover Evi_I (Line 3)[1]. For each minimal cover **X**, we derive a valid minimal DC by inverting each predicate in it (Lines 4-5). We remove implied DCs from Σ with Algorithm 2 (Lines 6-8).

Section 5.1 describes the procedure for building the predicate space **P**. Section 5.2 formally defines Evi_I, gives a theorem that reduces the problem of discovering all minimal DCs to the problem of finding all minimal covers for Evi_I, and presents a procedure for building Evi_I. Section 5.3 describes a search procedure for finding minimal covers for Evi_I. In order to reduce the execution time, the search is optimized with a dynamic ordering of predicates and branch pruning based on the axioms we developed in Section 4. In order to enable further pruning, Section 5.4 introduces an optimization technique that divides the space of DCs and performs DFS on each subspace. We extend FASTDC in Section 5.5 to discover approximate DCs and in Section 5.6 to discover DCs with constants.

[1]For sake of presentation, parameters are described in Section 5.3

5.1 Building the Predicate Space

Given a database schema R and an instance I, we build a predicate space **P** from which DCs can be formed. For each attribute in the schema, we add two equality predicates $(=, \neq)$ between two tuples on it. In the same way, for each numerical attribute, we add order predicates $(>, \leq, <, \geq)$. For every pair of attributes in R, they are *joinable* (*comparable*) if equality (order) predicates hold across them, and add cross column predicates accordingly.

Profiling algorithms [11] can be used to detect joinable and comparable columns. We consider two columns joinable if they are of same type and have common values[2]. Two columns are comparable if they are both of numerical types and the arithmetic means of two columns are within the same order of magnitude.

EXAMPLE 7. *Consider the following Employee table with three attributes: Employee ID (*I*), Manager ID (*M*), and Salary(*S*).*

TID	*I(String)*	*M(String)*	*S(Double)*
t_9	*A1*	*A1*	*50*
t_{10}	*A2*	*A1*	*40*
t_{11}	*A3*	*A1*	*40*

*We build the following predicate space **P** for it.*

$P_1 : t_\alpha.I = t_\beta.I$	$P_5 : t_\alpha.S = t_\beta.S$	$P_9 : t_\alpha.S < t_\beta.S$
$P_2 : t_\alpha.I \neq t_\beta.I$	$P_6 : t_\alpha.S \neq t_\beta.S$	$P_{10} : t_\alpha.S \geq t_\beta.S$
$P_3 : t_\alpha.M = t_\beta.M$	$P_7 : t_\alpha.S > t_\beta.S$	$P_{11} : t_\alpha.I = t_\alpha.M$
$P_4 : t_\alpha.M \neq t_\beta.M$	$P_8 : t_\alpha.S \leq t_\beta.S$	$P_{12} : t_\alpha.I \neq t_\alpha.M$
$P_{13} : t_\alpha.I = t_\beta.M$	$P_{14} : t_\alpha.I \neq t_\beta.M$	

5.2 Evidence Set

Before giving formal definitions of Evi_I, we show an example of the satisfied predicates for the Employee table Emp above:
$Evi_{Emp} = \{\{P_2, P_3, P_5, P_8, P_{10}, P_{12}, P_{14}\},$
$\{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}, \{P_2, P_3, P_6, P_7, P_{10}, P_{11}, P_{13}\}\}$.
Every element in Evi_{Emp} has at least one pair of tuples in I such that every predicate in it is satisfied by that pair of tuples.

DEFINITION 2. *Given a pair of tuple $\langle t_x, t_y\rangle \in I$, the satisfied predicate set for $\langle t_x, t_y\rangle$ is $SAT(\langle t_x, t_y\rangle) = \{P | P \in \boldsymbol{P}, \langle t_x, t_y\rangle \models P\}$, where **P** is the predicate space, and $\langle t_x, t_y\rangle \models P$ means $\langle t_x, t_y\rangle$ satisfies P.*

The evidence set *of I is $Evi_I = \{SAT(\langle t_x, t_y\rangle) | \forall \langle t_x, t_y\rangle \in I\}$.*

A set of predicates $\boldsymbol{X} \subseteq \boldsymbol{P}$ is a minimal set cover *for Evi_I if $\forall E \in Evi_I, \boldsymbol{X} \cap E \neq \emptyset$, and $\nexists \boldsymbol{Y} \subset \boldsymbol{X}$, s.t. $\forall E \in Evi_I, \boldsymbol{Y} \cap E \neq \emptyset$.*

The minimal set cover for Evi_I is a set of predicates that intersect with every element in Evi_I. Theorem 2 transforms the problem of minimal DCs discovery into the problem of searching for minimal set covers for Evi_I.

THEOREM 2. *$\urcorner(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid minimal DC if and only if $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ is a minimal set cover for Evi_I.*

[2]We show in the experiments that requiring at least 30% common values allows to identify joinable columns without introducing a large number of unuseful predicates. Joinable columns can also be discovered from query logs, if available.

Proof. Step 1: we prove if $\mathbf{X} \subseteq \mathbf{P}$ is a cover for Evi_I, $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid DC. According to the definition, Evi_I represents all the pieces of evidence that might violate DCs. For any $E \in Evi_I$, there exists $X \in \mathbf{X}$, s.t. $X \in E$; thus $\overline{X} \notin E$. I.e., the presence of $\overline{X}$ in $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ disqualifies E as a possible violation.

Step 2: we prove if $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid DC, then $\mathbf{X} \subseteq \mathbf{P}$ is a cover. According to the definition of valid DC, there does not exist tuple pair $\langle t_x, t_y \rangle$, s.t. $\langle t_x, t_y \rangle$ satisfies $\overline{X_1}, \ldots, \overline{X_n}$ simultaneously. In other words, $\forall \langle t_x, t_y \rangle$, $\exists \overline{X_i}$, s.t. $\langle t_x, t_y \rangle$ does not satisfy $\overline{X_i}$. Therefore, $\forall \langle t_x, t_y \rangle$, $\exists X_i$, s.t. $\langle t_x, t_y \rangle \models X_i$, which means any tuple pair's satisfied predicate set is covered by $\{X_1, \ldots, X_n\}$.

Step 3: if $\mathbf{X} \subseteq \mathbf{P}$ is a minimal cover, then the DC is also minimal. Assume the DC is not minimal, there exists another DC φ whose predicates are a subset of $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$. According to Step 2, $\varphi.Pres$ is a cover, which is a subset of $\mathbf{X} = \{X_1, \ldots, X_n\}$. It contradicts with the assumption that $\mathbf{X} \subseteq \mathbf{P}$ is a minimal cover.

Step 4: if the DC is minimal, then the corresponding cover is also minimal. The proof is similar to Step 3. $\diamond$

EXAMPLE 8. *Consider Evi_{Emp} for the table in Example 7.*

$X_1 = \{P_2\}$ is a minimal cover, thus $\neg(\overline{P_2})$, i.e., $\neg(t_\alpha.I = t_\beta.I)$ is a valid DC, which states I *is a key.*

$X_2 = \{P_{10}, P_{14}\}$ is another minimal cover, thus $\neg(\overline{P_{10}} \wedge \overline{P_{14}})$, i.e., $\neg(t_\alpha.S < t_\beta.S \wedge t_\alpha.I = t_\beta.M)$ is another valid DC, which states that a manager's salary cannot be less than her employee's.

The procedure to compute Evi_I follows directly from the definition: for every tuple pair in I, we compute the set of predicates that tuple pair satisfies, and we add that set into Evi_I. This operation is sensitive to the size of the database, with a complexity of $O(|\mathbf{P}| \times |I|^2)$. However, for every tuple pair, computing the satisfied set of predicates is independent of each other. In our implementation we use the *Grid Scheme* strategy, a standard approach to scale in entity resolution [4]. We partition the data into B blocks, and define each task as a comparison of tuples from two blocks. The total number of tasks is $\frac{B^2}{2}$. Suppose we have M machines, we need to distribute the tasks evenly to M machines so as to fully utilize every machine, i.e., we need to ensure $\frac{B^2}{2} = w \times M$ with w the number of tasks for each machine. Therefore, the number of blocks $B = \sqrt{2wM}$. In addition, as we need at least two blocks in memory at any given time, we need to make sure that ($2 \times \frac{|I|}{B} \times$ *Size of a Tuple*) $<$ *Memory Limit*.

5.3 DFS for Minimal Covers

Algorithm 4 presents the depth-first search (DFS) procedure for minimal covers for Evi_I. Ignore Lines (9-10) and Lines (11-12) for now, as they are described in Section 5.4 and in Section 6.3, respectively. We denote by Evi_{curr} the set of elements in Evi_I not covered so far. Initially $Evi_{curr} = Evi_I$. Whenever a predicate P is added to the cover, we remove from Evi_{curr} the elements that contain P, i.e., $Evi_{next} = \{E | E \in E_{curr} \wedge P \notin E\}$ (Line 23). There are two base cases to terminate the search:

(i) there are no more candidate predicates to include in the cover, but $Evi_{curr} \neq \emptyset$ (Lines 14-15); and

(ii) $Evi_{curr} = \emptyset$ and the current path is a cover (Line 16). If the cover is minimal, we add it to the result ***MC*** (Lines 17-19).

We speed up the search procedure by two optimizations: dynamic ordering of predicates as we descend down the search tree and branching pruning based on the axioms in Section 4.

Opt1: Dynamic Ordering. Instead of fixing the order of predicates when descending down the tree, we dynamically order the remaining candidate predicates, denoted as $>_{next}$, based on the number of remaining evidence set they cover (Lines 23

Algorithm 4 SEARCH MINIMAL COVERS

```
Input: 1. Input Evidence set, $Evi_I$
    2. Evidence set not covered so far, $Evi_{curr}$
    3. The current path in the search tree, $\mathbf{X} \subseteq \mathbf{P}$
    4. The current partial ordering of the predicates, $>_{curr}$
    5. The DCs discovered so far, $\Sigma$
Output: A set of minimal covers for $Evi$, denoted as MC
 1: Branch Pruning
 2: $P \leftarrow \mathbf{X}.last$ // Last Predicate added into the path
 3: if $\exists Q \in \overline{\mathbf{X} - P}$, s.t. $P \in Imp(Q)$ then
 4:     return //Triviality pruning
 5: if $\exists \mathbf{Y} \in$ MC, s.t. $\mathbf{X} \supseteq \mathbf{Y}$ then
 6:     return //Subset pruning based on MC
 7: if $\exists \mathbf{Y} = \{Y_1, \ldots, Y_n\} \in$ MC, and $\exists i \in [1, n]$,
    and $\exists Q \in Imp(Y_i)$, s.t. $\mathbf{Z} = \mathbf{Y}_{-i} \cup \overline{Q}$ and $\mathbf{X} \supseteq \mathbf{Z}$ then
 8:     return //Transitive pruning based on MC
 9: if $\exists \varphi \in \Sigma$, s.t. $\overline{\mathbf{X}} \supseteq \varphi.Pres$ then
10:     return //Subset pruning based on previous discovered DCs
11: if $Inter(\varphi) < t, \forall \varphi$ of the form $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$ then
12:     return //Pruning based on $Inter$ score
13: Base cases
14: if $>_{curr} = \emptyset$ and $Evi_{curr} \neq \emptyset$ then
15:     return //No DCs in this branch
16: if $Evi_{curr} = \emptyset$ then
17:     if no subset of size $|\mathbf{X}| - 1$ covers $Evi_{curr}$ then
18:         MC $\leftarrow$ MC $+ \mathbf{X}$
19:     return //Got a cover
20: Recursive cases
21: for all Predicate $P \in >_{curr}$ do
22:     $\mathbf{X} \leftarrow \mathbf{X} + P$
23:     $Evi_{next} \leftarrow$ evidence sets in $Evi_{curr}$ not yet covered by $P$
24:     $>_{next} \leftarrow$ total ordering of $\{P' | P >_{curr} P'\}$ wrt $Evi_{next}$
25:     SEARCH MINIMAL COVERS($Evi_I$, $Evi_{next}$, $\mathbf{X}$, $>_{next}$, $\Sigma$)
26:     $\mathbf{X} \leftarrow \mathbf{X} - P$
```

-24). Formally, we define the cover of P w.r.t. Evi_{next} as $Cov(P, Evi_{next}) = |\{P \in E | E \in Evi_{next}\}|$. And we say that $P >_{next} Q$ if $Cov(P, Evi_{next}) > Cov(Q, Evi_{next})$, or $Cov(P, Evi_{next}) = Cov(Q, Evi_{next})$ and P appears before Q in the preassigned order in the predicate space. The initial evidence set Evi_I is computed as discussed in Section 5.2. To computer Evi_{next} (Line 21), we scan every element in Evi_{curr}, and we add in Evi_{next} those elements that do not contain P.

EXAMPLE 9. *Consider Evi_{Emp} for the table in Example 7. We compute the cover for each predicate, such as $Cov(P_2, Evi_{Emp})$ = 3, $Cov(P_8, Evi_{Emp})$ = 2, $Cov(P_9, Evi_{Emp})$ = 1, etc. The initial ordering for the predicates according to Evi_{Emp} is $>_{init} = P_2 > P_3 > P_6 > P_8 > P_{10} > P_{12} > P_{14} > P_5 > P_7 > P_9 > P_{11} > P_{13}$.*

Opt2: Branch Pruning. The purpose of performing dynamic ordering of candidate predicates is to get covers as early as possible so that those covers can be used to prune unnecessary branches of the search tree. We list three pruning strategies.

(i) Lines(2-4) describe the first pruning strategy. This branch would eventually result in a DC of the form $\varphi : \neg(\overline{\mathbf{X} - P} \wedge \overline{P} \wedge \mathbf{W})$, where P is the most recent predicate added to this branch and $\mathbf{W}$ other predicates if we traverse this branch. If $\exists Q \in \overline{\mathbf{X} - P}$, s.t. $P \in Imp(Q)$, then φ is trivial according to Axiom Triviality.

(ii) Lines(5-6) describe the second branch pruning strategy, which is based on ***MC***. If $\mathbf{Y}$ is in the cover, then $\neg(\overline{\mathbf{Y}})$ is a valid DC. Any branch containing $\mathbf{X}$ would result in a DC of the form $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$, which is implied by $\neg(\overline{\mathbf{Y}})$ based on Axiom Augmentation, since $\overline{\mathbf{Y}} \subseteq \overline{\mathbf{X}}$.

(iii) Lines(7-8) describe the third branching pruning strategy, which is also based on ***MC***. If $\mathbf{Y}$ is in the cover, then $\neg(\overline{\mathbf{Y}_{-i}} \wedge \overline{Y_i})$ is a valid DC. Any branch containing $\mathbf{X} \supseteq \mathbf{Y}_{-i} \cup \overline{Q}$ would result in a DC of the form $\urcorner(\overline{\mathbf{Y}}_{-i} \wedge Q \wedge \mathbf{W})$. Since $Q \in Imp(Y_i)$, by applying Axiom Transitive on these two DCs, we would get that $\urcorner(\overline{\mathbf{Y}}_{-i} \wedge \mathbf{W})$ is also a valid DC, which would imply $\urcorner(\overline{\mathbf{Y}}_{-i} \wedge Q \wedge \mathbf{W})$ based on Axiom Augmentation. Thus this branch can be pruned.

5.4 Dividing the Space of DCs

Instead of searching for all minimal DCs at once, we divide the space into subspaces, based on whether a DC contains a specific predicate P_1, which can be further divided according to whether a DC contains another specific predicate P_2. We start by defining *evidence set modulo a predicate* P, i.e., Evi_I^P, and we give a theorem that reduces the problem of discovering all minimal DCs to the one of finding all minimal set covers of Evi_I^P for each $P \in \mathbf{P}$.

DEFINITION 3. *Given a* $P \in \boldsymbol{P}$, *the* evidence set *of* I modulo P *is,* $Evi_I^P = \{E - \{P\} | E \in Evi_I, P \in E\}$.

THEOREM 3. $\urcorner(\overline{X_1} \wedge \ldots \wedge \overline{X_n} \wedge P)$ *is a valid minimal DC, that contains predicate P, if and only if* $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ *is a* minimal set cover *for* Evi_I^P.

EXAMPLE 10. *Consider* Evi_{Emp} *for the table in Example 7,* $Evi_{Emp}^{P_1} = \emptyset$, $Evi_{Emp}^{P_{13}} = \{\{P_2, P_3, P_6, P_7, P_{10}, P_{11}\}\}$. *Thus* $\urcorner(P_1)$ *is a valid DC because there is nothing in the cover for* $Evi_{Emp}^{P_1}$, *and* $\urcorner(P_{13} \wedge \overline{P_{10}})$ *is a valid DC as* $\{P_{10}\}$ *is a cover for* $Evi_{Emp}^{P_{13}}$. *It is evident that* Evi_{Emp}^{P} *is much smaller than* Evi_{Emp}.

However, care must be taken before we start to search for minimal covers for Evi_I^P due to the following two problems.

First, a minimal DC containing a certain predicate P is not necessarily a global minimal DC. For instance, assume that $\urcorner(P, Q)$ is a minimal DC containing P because $\{\overline{Q}\}$ is a minimal cover for Evi_I^P. However, it might not be a minimal DC because it is possible that $\urcorner(Q)$, which is actually smaller than $\urcorner(P, Q)$, is also a valid DC. We call such $\urcorner(P, Q)$ a *local minimal DC w.r.t.* P, and $\urcorner(Q)$ a *global minimal DC*, or a minimal DC. It is obvious that a global minimal DC is always a local minimal DC w.r.t. each predicate in the DC. Our goal is to generate all globally minimal DCs.

Second, assume that $\urcorner(P, Q)$ is a global minimal DC. It is an local minimal DC w.r.t. P and Q, thus would appear in subspaces Evi_I^P and Evi_I^Q. In fact, a minimal DC φ would then appear in $|\varphi.Pres|$ subspaces, causing a large amount of repeated work.

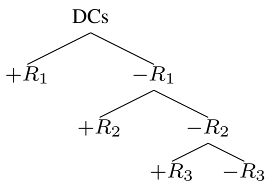


Figure 2: Taxonomy Tree.

We solve the second problem first, then the solution for the first problem comes naturally. We divide the DCs space and order all searches in a way, such that we ensure the output of a locally minimal DC is indeed global minimal, and a previously generated minimal DC will never appear again in latter searches. Consider a predicate space $\mathbf{P}$ that has only 3 predicates R_1 to R_3 as in Figure 2, which presents a taxonomy of all DCs. In the first level, all DCs can be divided into DCs containing R_1, denoted as $+R_1$, and DCs not containing R_1, denoted as $-R_1$. Since we know how to search for local minimal DCs containing R_1, we only need to further process DCs not containing R_1, which can be divided based on containing R_2 or not, i.e., $+R_2$ and $-R_2$. We will divide $-R_2$ as in Figure 2. We can enforce searching for DCs not containing R_i by disallowing $\overline{R_i}$ in the initial ordering of candidate predicates for minimal cover. Since this is a taxonomy of all DCs, no minimal DCs can be generated more than once.

We solve the first problem by performing DFS according to the taxonomy tree in a bottom-up fashion. We start by search for DCs containing R_3, not containing R_1, R_2. Then we search for DCs, containing R_2, not containing R_1, and we verify the resulting DC is global minimal by checking if the reverse of the minimal cover is a super set of DCs discovered from $Evi_I^{R_3}$. The process goes on until we reach the root of the taxonomy, thus ensuring that the results are both globally minimal and complete.

Dividing the space enables more optimization opportunities:

1. Reduction of Number of Searches. If $\exists P \in \mathbf{P}$, such that $Evi_I^P = \emptyset$, we identify two scenarios for Q, where DFS for Evi_I^Q can be eliminated.

(i) $\forall Q \in Imp(\overline{P})$, if $Evi_I^P = \emptyset$, then $\urcorner(P)$ is a valid DC. The search for Evi_I^Q would result in a DC of the form $\urcorner(Q \wedge \mathbf{W})$, where $\mathbf{W}$ represents any other set of predicates. Since $Q \in Imp(\overline{P})$, applying Axiom Transitivity, we would have that $\urcorner(\mathbf{W})$ is a valid DC, which implies $\urcorner(Q \wedge \mathbf{W})$ based on Axiom Augmentation.

(ii) $\forall \overline{Q} \in Imp(\overline{P})$, since $\overline{Q} \in Imp(\overline{P})$, then $Q \models P$. It follows that $Q \wedge \mathbf{W} \models P$ and therefore $\urcorner(P) \models \urcorner(Q \wedge \mathbf{W})$ holds.

EXAMPLE 11. *Consider* Evi_{Emp} *for the table in Example 7, since* $Evi_{Emp}^{P_1} = \emptyset$ *and* $Evi_{Emp}^{P_4} = \emptyset$, *then* $\boldsymbol{Q} = \{P_1, P_2, P_3, P_4\}$. *Thus we perform* $|\boldsymbol{P}| - |\boldsymbol{Q}| = 10$ *searches instead of* $|\boldsymbol{P}| = 14$.

2. Additional Branch Pruning. Since we perform DFS according to the taxonomy tree in a bottom-up fashion, DCs discovered from previous searches are used to prune branches in current DFS described by Lines(9-10) of Algorithm 4.

Since Algorithm 4 is an exhaustive search for all minimal covers for Evi_I, Algorithm 3 produces all minimal DCs.

THEOREM 4. *Algorithm 3 produces all non-trivial minimal DCs holding on input database I.*

Complexity Analysis of FASTDC. The initialization of evidence sets takes $O(|\mathbf{P}| * n^2)$. The time for each DFS search to find all minimal covers for Evi_I^P is $O((1 + w_P) * K_P)$, with w_P being the extra effort due to imperfect search of Evi_I^P, and K_P being the number of minimal DCs containing predicate P. Altogether, our FASTDC algorithm has worst time complexity of $O(|\mathbf{P}| * n^2 + |\mathbf{P}| * (1 + w_P) * K_P)$.

5.5 Approximate DCs: A-FASTDC

Algorithm FASTDC consumes the whole input data set and requires no violations for a DC to be declared valid. In real scenarios, there are multiple reasons why this request may need to be relaxed: (1) overfitting: data is dynamic and as more data becomes available, overfitting constraints on current data set can be problematic; (2) data errors: while in general learning from unclean data is a challenge, the common belief is that errors constitute small percentage of data, thus discovering constraints that hold for most of the dataset is a common workaround [8, 15, 17].

We therefore modify the discovery statement as follows: given a relational schema R and instance I, the *approximate DCs discovery problem* for DCs is to find all valid DCs, where a DC φ is valid if the percentage of violations of φ on I, i.e., number of violations of φ on I divided by total number of tuple pairs $|I|(|I| - 1)$, is within threshold ϵ. For this new problem, we introduce A-FASTDC.

Different tuple pairs might have the same satisfied predicate set. For every element E in Evi_I, we denote by $count(E)$ the number of tuple pairs $\langle t_x, t_y \rangle$ such that $E = SAT(\langle t_x, t_y \rangle)$. For example, $count(\{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}) = 2$ for the table in Example 7 since $SAT(\langle t_{10}, t_9 \rangle) = SAT(\langle t_{11}, t_9 \rangle) = \{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}$.

DEFINITION 4. *A set of predicates* $\mathbf{X} \subseteq \mathbf{P}$ *is an* ϵ-minimal cover *for* Evi_I *if* $Sum(count(E)) \leq \epsilon|I|(|I|-1)$, *where* $E \in Evi_I, \mathbf{X} \cap E = \emptyset$, *and no subset of* $\mathbf{X}$ *has such property.*

Theorem 5 transforms approximate DCs discovery problem into the problem of searching for ϵ-minimal covers for Evi_I.

THEOREM 5. $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ *is a valid approximate minimal DC if and only if* $\mathbf{X}=\{X_1, \ldots, X_n\}$ *is a* ϵ-minimal cover *for* Evi_I.

There are two modifications for Algorithm 4 to search for ϵ-minimal covers for Evi_I: (1) the dynamic ordering of predicates is based on $Cov(P, Evi) = \sum_{E \in \{E \in Evi, P \in E\}} count(E)$; and (2) the base cases (Lines 12-17) are either when the number of violations of the corresponding DC drops below $\epsilon|I|(|I|-1)$, or the number of violation is still above $\epsilon|I|(|I|-1)$ but there are no more candidate predicates to include. Due to space limitations, we present in [9] the detailed modifications for Algorithm 4 to search for ϵ-minimal covers for Evi_I.

5.6 Constant DCs: C-FASTDC

FASTDC discovers DCs without constant predicates. However, just like FDs may not hold on the entire dataset, thus CFDs are more useful, we are also interested in discovering constant DCs (CDCs). Algorithm 5 describes the procedure for CDCs discovery. The first step is to build a constant predicate space **Q** (Lines 1-6)[3]. After that, one direct way to discover CDCs is to include **Q** in the predicate space **P**, and follow the same procedure in Section 5.3. However, the number of constant predicates is linear w.r.t. the number of constants in the active domain, which is usually very large. Therefore, we follow the approach of [15] and focus on discovering τ-frequent CDCs. The support for a set of constant predicates **X** on I, denoted by $sup(\mathbf{X}, I)$, is defined to be the set of tuples that satisfy all constant predicates in **X**. A set of predicates is said to be τ-frequent if $\frac{|sup(\mathbf{X},I)|}{|I|} \geq \tau$. A CDC φ consisting of only constant predicates is said to be τ-frequent if all strict subsets of $\varphi.Pres$ are τ-frequent. A CDC φ consisting of constant and variable predicates is said to be k-frequent if all subsets of φ's constant predicates are τ-frequent.

EXAMPLE 12. *Consider* c_3 *in Example 1,* $sup(\{t_\alpha.CT = \text{'Denver'}\}, I) = \{t_2, t_6\}$, $sup(\{t_\alpha.ST \neq \text{'CO'}\}, I) = \{t_1, t_3, t_4, t_5, t_7, t_8\}$, *and* $sup(\{c3.Pres\}, I) = \emptyset$. *Therefore,* c_3 *is a* τ*-frequent CDC, with* $\frac{2}{8} \geq \tau$.

We follow an "Apriori" approach to discover τ-frequent constant predicate sets. We first identify frequent constant predicate sets of length L_1 from **Q** (Lines 7-15). We then generate candidate frequent constant predicate sets of length m from length $m-1$ (Lines 22-28), and we scan the database I to get their support (Line 24). If the support of the candidate c is 0, we have a valid CDC with only constant predicates (Lines 12-13 and 25-26); if the support of the candidate c is greater than τ, we call FASTDC to get the variable DCs (VDCs) that hold on $sup(c, I)$, and we construct CDCs by combining the τ-frequent constant predicate sets and the variable predicates of VDCs (Lines 18-21).

[3]We focus on two tuple CDCs with the same constant predicates on each tuple, i.e., if $t_\alpha.A\theta c$ is present in a two tuple CDC, $t_\beta.A\theta c$ is enforced by the algorithm. Therefore, we only add $t_\alpha.A\theta c$ to **Q**.

Algorithm 5 C-FASTDC

```
Input: Instance $I$, schema $R$, minimal frequency requirement $\tau$
Output: Constant DCs $\Gamma$
 1: Let $\mathbf{Q} \leftarrow \emptyset$ be the constant predicate space
 2: for all $A \in R$ do
 3:     for all $c \in ADom(A)$ do
 4:         $\mathbf{Q} \leftarrow \mathbf{Q} + t_\alpha.A\theta c$, where $\theta \in \{=, \neq\}$
 5:         if $A$ is numerical type then
 6:             $\mathbf{Q} \leftarrow \mathbf{Q} + t_\alpha.A\theta c$, where $\theta \in \{>, \leq, <, \geq\}$
 7: for all $t \in I$ do
 8:     if $t$ satisfies $Q$ then
 9:         $sup(Q, I) \leftarrow sup(Q, I) + t$
10: Let $L_1$ be the set of frequent predicates
11: for all $Q \in \mathbf{Q}$ do
12:     if $|sup(Q, I)| = 0$ then
13:         $\Gamma \leftarrow \Gamma + \neg(Q)$
14:     else if $\frac{|sup(\mathbf{Q},I)|}{|I|} \geq \tau$ then
15:         $L_1 \leftarrow L_1 + \{Q\}$
16: $m \leftarrow 2$
17: while $L_{m-1} \neq \emptyset$ do
18:     for all $c \in L_{m-1}$ do
19:         $\Sigma \leftarrow$ FASTDC($sup(c, I)$,R)
20:         for all $\varphi \in \Sigma$ do
21:             $\Gamma \leftarrow \Gamma + \phi$, $\phi$'s predicates comes from $c$ and $\varphi$
22:     $C_m = \{c | c = a \cup b \wedge a \in L_{m-1} \wedge b \in \bigcup L_{k-1} \wedge b \notin a\}$
23:     for all $c \in C_m$ do
24:         scan the database to get the support of $c$, $sup(c, I)$
25:         if $|sup(c, I)| = 0$ then
26:             $\Gamma \leftarrow \Gamma + \phi$, $\phi$'s predicates consist of predicates in $c$
27:         else if $\frac{|sup(c,I)|}{|I|} \geq \tau$ then
28:             $L_m \leftarrow L_m + c$
29:     $m \leftarrow m + 1$
```

6. RANKING DCS

Though our FASTDC (C-FASTDC) is able to prune trivial, non-minimal, and implied DCs, the number of DCs returned can still be too large. To tackle this problem, we propose a scoring function to rank DCs based on their size and their support from the data. Given a DC φ, we denote by $Inter(\varphi)$ its *interestingness* score.

We recognize two different dimensions that influence $Inter(\varphi)$: *succinctness* and *coverage* of φ, which are both defined on a scale between 0 and 1. Each of the two scores represents a different yet important intuitive dimension to rank discovered DCs.

Succinctness is motivated by the Occam's razor principle. This principle suggests that among competing hypotheses, the one that makes fewer assumptions is preferred. It is also recognized that overfitting occurs when a model is excessively complex [6].

Coverage is also a general principle in data mining to rank results [2]. They design scoring functions that measure the statistical significance of the mining targets in the input data.

Given a DC φ, we define the interestingness score as a linear weighted combination of the two dimensions: $Inter(\varphi) = a \times Coverage(\varphi) + (1-a) \times Succ(\varphi)$.

6.1 Succinctness

Minimum description length (MDL), which measures the code length needed to compress the data [6], is a formalism to realize the Occam's razor principle. Inspired by MDL, we measure the length of a DC $Len(\varphi)$, and we define the *succinctness* of a DC φ, i.e., $Succ(\varphi)$, as the minimal possible length of a DC divided by $Len(\varphi)$ thus ensuring the scale of $Succ(\varphi)$ is between 0 and 1.

$$Succ(\varphi) = \frac{\text{Min}(\{Len(\phi) | \forall \phi\})}{Len(\varphi)}$$

One simple heuristic for $Len(\varphi)$ is to use the number of predicates in φ, i.e., $|\varphi.Pres|$. Our proposed function computes the length of a DC with a finer granularity than a simple counting of the predicates. To compute it, we identify the alphabet from which DCs are formed as $\mathbb{A} = \{t_\alpha, t_\beta, \mathbb{U}, \mathbb{B}, Cons\}$, where $\mathbb{U}$ is the set of all attributes, $\mathbb{B}$ is the set of all operators, and $Cons$ are constants. The length of φ is the number of symbols in $\mathbb{A}$ that appear in φ: $Len(\varphi) = |\{a | a \in \mathbb{A}, a \in \varphi\}|$. The shortest possible DC is of length 4, such as c_5, c_9, and $\urcorner(t_\alpha.SAL \leq 5000)$.

EXAMPLE 13. *Consider a database schema R with two columns A, B, with 3 DCs as follows:*
$c_{14} : \urcorner(t_\alpha.A = t_\beta.A)$, $c_{15} : \urcorner(t_\alpha.A = t_\beta.B)$,
$c_{16} : \urcorner(t_\alpha.A = t_\beta.A \wedge t_\alpha.B \neq t_\beta.B)$
Len(c_{14}) = 4 < Len(c_{15}) = 5 < Len(c_{16}) = 6. Succ(c_{14}) = 1, Succ(c_{15}) = 0.8, and Succ(c_{16})=0.67. However, if we use $|\varphi.Pres|$ as Len(φ), Len(c_{14}) = 1 < Len(c_{15}) = 1 < Len(c_{16}) = 2, and Succ(c_{14})=1, Succ(c_{15})=1, and Succ(c_{16})=0.5.

6.2 Coverage

Frequent itemset mining recognizes the importance of measuring statistical significance of the mining targets [2]. In this case, the support of an itemset is defined as the proportion of transactions in the data that contain the itemset. Only if the support of an itemset is above a threshold, it is considered to be frequent. CFDs discovery also adopts such principle. A CFD is considered to be interesting only if their support in the data is above a certain threshold, where support is in general defined as the percentage of single tuples that match the constants in the patten tableaux of the CFDs [8, 15].

However, the above statistical significance measures requires the presence of constants in the mining targets. For example, the frequent itemsets are a set of items, which are constants. In CFDs discovery, a tuple is considered to support a CFD if that tuple matches the constants in the CFD. Our target DCs may lack constants, and so do FDs. Therefore, we need a novel measure for statistical significance of discovered DCs on I that extends previous approaches.

EXAMPLE 14. *Consider c_2, which is a FD, in Example 1. If we look at single tuples, just as the statistical measure for CFDs, every tuple matches c_2 since it does not have constants. However, it is obvious that the tuple pair $\langle t_4, t_7\rangle$ gives more support than the tuple pair $\langle t_2, t_6\rangle$ because $\langle t_4, t_7\rangle$ matches the left hand side of c_2.*

Being a more general form than CFDs, DCs have more kinds of evidence that we exploit in order to give an accurate measure of the statistical significance of a DC on I. An *evidence* of a DC φ is a pair of tuples that does not violate φ: there exists at least one predicate in φ that is not satisfied by the tuple pair. Depending on the number of satisfied predicates, different evidences give different support to the statistical significance score of a DC. The larger the number of satisfied predicates is in a piece of evidence, the more support it gives to the interestingness score of φ. A pair of tuples satisfying k predicates is a *k-evidence* (kE). As we want to give higher score to high values of k, we need a weight to reflect this intuition in the scoring function. We introduce $w(k)$ for kE, which is from 0 to 1, and increases with k. In the best case, the maximum k for a DC φ is equal to $|\varphi.Pres| - 1$, otherwise the tuple pair violates φ.

DEFINITION 5. *Given a DC φ:*
A k-evidence (kE) for φ w.r.t. a relational instance I is a tuple pair $\langle t_x, t_y\rangle$, where k is the number of predicates in φ that are satisfied by $\langle t_x, t_y\rangle$ and $k \leq |\varphi.Pres| - 1$.
The weight for a kE ($w(k)$) for φ is $w(k) = \frac{(k+1)}{|\varphi.Pres|}$.

EXAMPLE 15. *Consider c_7 in Example 3, which has 2 predicates. There are two types of evidences, i.e., 0E and 1E.*
$\langle t_1, t_2\rangle$ is a 0E since $t_1.FN \neq t_2.FN$ and $t_1.GD = t_2.GD$.
$\langle t_1, t_3\rangle$ is a 1E since $t_1.FN \neq t_3.FN$ and $t_1.GD \neq t_3.GD$.
$\langle t_1, t_6\rangle$ is a 1E since $t_1.FN = t_6.FN$ and $t_1.GD = t_6.GD$.
Clearly, $\langle t_1, t_3\rangle$ and $\langle t_1, t_6\rangle$ have higher weight than $\langle t_1, t_2\rangle$.

Given such evidence, we define $Coverage(\varphi)$ as follows:

$$Coverage(\varphi) = \frac{\sum_{k=0}^{|\varphi.Pres|-1} |kE| * w(k)}{\sum_{k=0}^{|\varphi.Pres|-1} |kE|}$$

The enumerator of $Coverage(\varphi)$ counts the number of different evidences weighted by their respective weights, which is divided by the total number of evidences. $Coverage(\varphi)$ gives a score between 0 and 1, with higher score indicating higher statistical significance.

EXAMPLE 16. *Given 8 tuples in Table 1, we have 8*7=56 evidences. $Coverage(c_7) = 0.80357$, $Coverage(c_2) = 0.9821$. It can be seen that coverage score is more confident about c_2, thus reflecting our intuitive comparison between c_2 and c_7 in Section 1.*
Coverage for CDC is calculated using the same formula, such as $Coverage(c_3) = 1.0$.

6.3 Rank-aware Pruning in DFS Tree

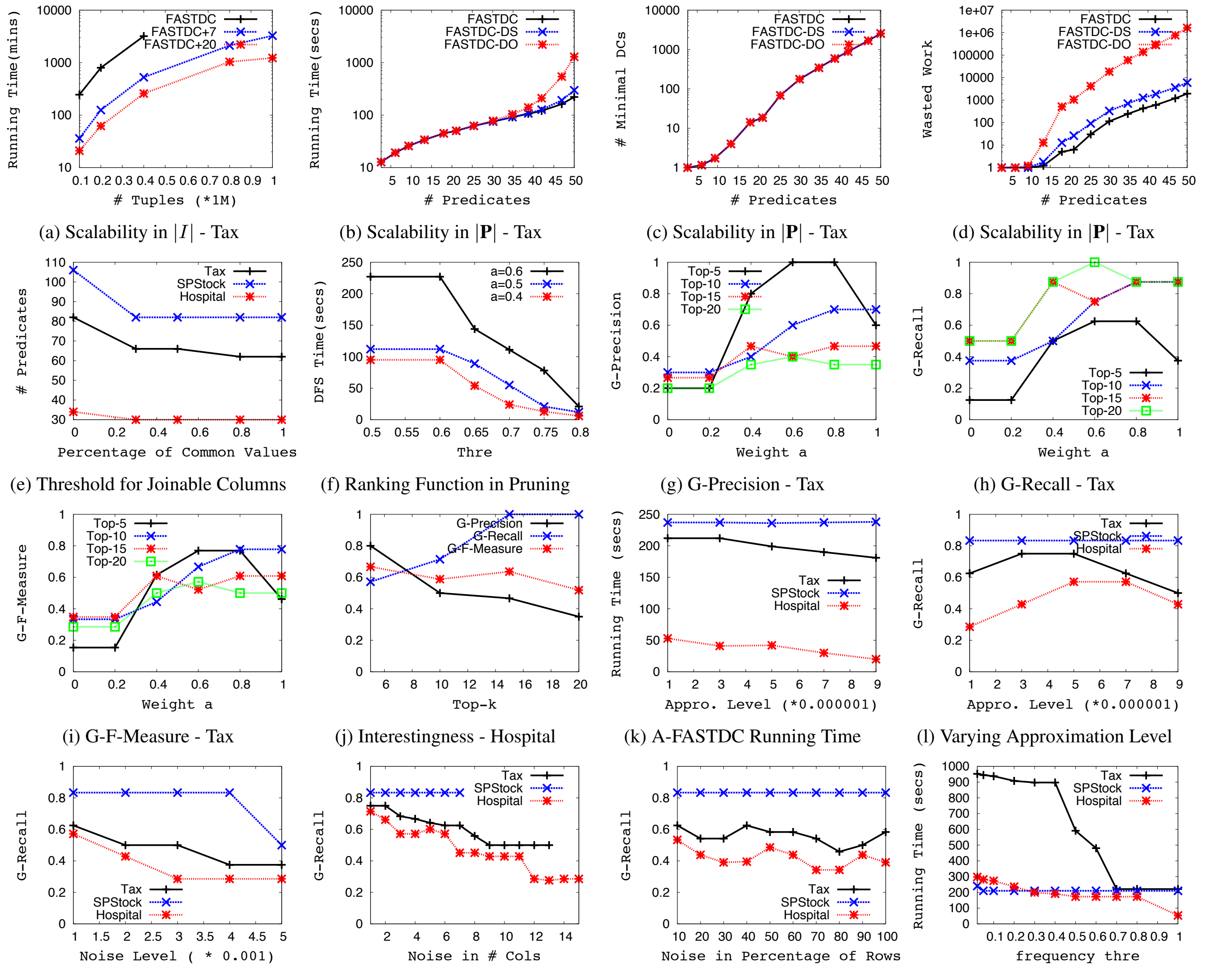
Having defined $Inter$, we can use it to prune branches in the DFS tree when searching for minimal covers in Algorithm 4. We can prune any branch in the DFS tree, if we can upper bound the $Inter$ score of any possible DC resulting from that branch, and the upper bound is either (i) less than a minimal $Inter$ threshold, or (ii) less than the minimal $Inter$ score of the Top-k DCs we have already discovered. We use this pruning in Algorithm 4 (Lines 11-12), a branch with the current path $\mathbf{X}$ will result in a DC φ: $\urcorner(\overline{\mathbf{X}} \wedge \mathbf{W})$, with $\mathbf{X}$ known and $\mathbf{W}$ unknown.

$Succ$ score is an anti-monotonic function: adding more predicates increases the length of a DC, thus decreases the $Succ$ of a DC. Therefore we bound $Succ(\varphi)$ by $Succ(\varphi) \leq Succ(\urcorner(\overline{\mathbf{X}}))$. However, as $Coverage(\varphi)$ is not anti-monotonic, we cannot use $\urcorner(\overline{\mathbf{X}})$ to get an upper bound for it. A direct upper bound, but not useful bound is 1.0, so we improve it as follows. Each evidence E or tuple pair is contributing $w(k) = \frac{(k+1)}{|\varphi.Pres|}$ to $Coverage(\varphi)$ with k being the number of predicates in φ that E satisfies. $w(k)$ can be rewritten as $w(k) = 1 - \frac{l}{|\varphi.Pres|}$ with l being the number of predicates in φ that E does not satisfy. In addition, we know l is greater than or equal to the number of predicates in $\overline{\mathbf{X}}$ that E does not satisfy; and we know that $|\varphi.Pres|$ must be less than the $\frac{|\mathbf{P}|}{2}$. Therefore, we get an upper bound for $w(k)$ for each evidence. The average of the upper bounds for all evidences is a valid upper bound for $Coverage(\varphi)$. However, to calculate this bound, we need to iterate over all the evidences, which can be expensive because we need to do that for every branch in the DFS tree. Therefore, to get a tighter bound than 1.0, we only upper bound the $w(k)$ for a small number of evidences[4], and for the rest we set $w(k) \leq 1$. We show in the experiments how different combinations of the upper bounds of $Succ(\varphi)$ and of $Coverage(\varphi)$ affect the results.

7. EXPERIMENTAL STUDY

We experimentally evaluate FASTDC, $Inter$ function, A-FASTDC, and C-FASTDC. Experiments are performed on a Win7 machine with QuadCore 3.4GHz cpu and 4GB RAM. The scalability experiment runs on a cluster consisting of machines with the same configuration. We use one synthetic and two real datasets.

[4]We experimentally identified that 1000 samples improve the upper bound without affecting execution times.



(a) Scalability in $|I|$ - Tax (b) Scalability in $|\mathbf{P}|$ - Tax (c) Scalability in $|\mathbf{P}|$ - Tax (d) Scalability in $|\mathbf{P}|$ - Tax

(e) Threshold for Joinable Columns (f) Ranking Function in Pruning (g) G-Precision - Tax (h) G-Recall - Tax

(i) G-F-Measure - Tax (j) Interestingness - Hospital (k) A-FASTDC Running Time (l) Varying Approximation Level

(m) Varying Noise Level (n) Skewed Noise in Columns (o) Skewed Noise in Rows (p) C-FASTDC Running Time

Figure 3: FASTDC scalability (a-f), ranking functions w.r.t. Σ_g (g-j), A-FASTDC scalability (k) and quality (l-o), C-FASTDC scalability (p).

Synthetic. We use the *Tax* data generator from [7]. Each record represents an individual's address and tax information, as in Table 1. The address information is populated using real semantic relationship. Furthermore, salary is synthetic, while tax rates and tax exemptions (based on salary, state, marital status and number of children) correspond to real life scenarios.

Real-world. We use two datasets from different Web sources[5].

Hospital data is from the US government. There are 17 string attributes, including Provider # (PN), measure code (MC) and name (MN), phone (PHO), emergency service (ES) and has 115k tuples.

SP Stock data is extracted from historical S&P 500 Stocks. Each record is arranged into fields representing Date, Ticker, Open Price, High, Low, Close, and Volume of the day. There are 123k tuples.

7.1 Scalability Evaluation

We mainly use the Tax dataset to evaluate the running time of FASTDC by varying the number of tuples $|I|$, and the number of predicates $|\mathbf{P}|$. We also report running time for the Hospital and the SP Stock datasets. We show that our implication testing algorithm, though incomplete, is able to prune a huge number of implied DCs.

[5]http://data.medicare.gov, http://pages.swcp.com/stocks

Algorithms. We implemented FASTDC in Java, and we test various optimizations techniques. We use FASTDC+M to represent running FASTDC on a cluster consisting of M machines. We use FASTDC-DS to denote running FASTDC without dividing the space of DCs as in Section 5.4. We use FASTDC-DO to denote running FASTDC without dynamic ordering of predicates in the search tree as in Section 5.3.

Exp-1: Scalability in $|I|$. We measure the running time in minutes on all 13 attributes, by varying the number of tuples (up to 1 million tuples), as reported in Figure 3a. The size of the predicate space $|\mathbf{P}|$ is 50. The Y axis of Figure 3a is in log scale. We compare the running time of FASTDC and FASTDC+M with number of blocks B=2M to achieve load balancing. Figure 3a shows a quadratic trend as the computation is dominated by the tuple pairwise comparison for building the evidence set. In addition, Figure 3a shows that we achieve almost linear improvement w.r.t the number of machines on a cluster; for example, for 1M tuples, it took 3257 minutes on 7 machines, but 1228 minutes on 20 machines. Running FASTDC on a cluster is a viable approach if the number of tuples is too large to run on a single machine.

Exp-2: Scalability in $|\mathbf{P}|$. We measure the running time in seconds using 10k tuples, by varying the number of predicates through including different number of attributes in the Tax dataset, as in Figure 3b. We compare the running time of FASTDC, FASTDC-DS, and FASTDC-DO. The ordering of adding more attributes is randomly chosen, and we report the average running time over 20 executions. The Y axes of Figures 3b, 3c and 3d are in log scale. Figure 3b shows that the running time increases exponentially w.r.t. the number of predicates. This is not surprising because the number of minimal DCs, as well as the amount of wasted work, increases exponentially w.r.t. the number of predicates, as shown in Figures 3c and 3d. The amount of wasted work is measured by the number of times Line 15 of Algorithm 4 is hit. We estimate the wasted DFS time as a percentage of the running time by *wasted work / (wasted work + number of minimal DCs)*, and it is less than 50% for all points of FASTDC in Figure3d. The number of minimal DCs discovered is the same for FASTDC, FASTDC-DS, and FASTDC-DO as optimizations do not alter the discovered DCs.

Hospital has 34 predicates and it took 118 minutes to run on a single machine using all tuples. Stock has 82 predicates and it took 593 minutes to run on a single machine using all tuples.

Exp-3: Joinable Column Analysis. Figure 3e shows the number of predicates by varying the % of common values required to declare joinable two columns. Smaller values lead to a larger predicate space and higher execution times. Larger values lead to faster execution but some DCs involving joinable columns may be missed. The number of predicates gets stable with low percentage of common values, and with our datasets the quality of the output is not affected when at least 30% common values are required.

Exp-4: Ranking Function in Pruning. Figure 3f shows the DFS time taken for the Tax dataset varying the minimum $Inter$ score required for a DC to be in the output. The threshold has to exceed 0.6 to have pruning power. The higher the threshold, the more aggressive the pruning. In addition, a bigger weight for $Succ$ score (indicated by smaller a in Figure 3f) has more pruning power. Although in our experiment golden DCs are not dropped by this pruning, in general it is possible that the upper bound of $Inter$ for interesting DCs falls under the threshold, thus this pruning may lead to losing interesting DCs. The other use of ranking function for pruning is omitted since it has little gain.

Dataset	# DCs Before	# DCs After	% Reduction
Tax	1964	741	61%
Hospital	157	42	73%
SP Stock	829	621	25%

Table 3: # DCs before and after reduction through implication.

Exp-5: Implication Reduction. The number of DCs returned by FASTDC can be large, and many of them are implied by others. Table 3 reports the number of DCs we have before and after implication testing for datasets with 10k tuples. To prevent interesting DCs from being discarded, we rank them according to their $Inter$ function. A DC is discarded if it is implied by DCs with higher $Inter$ scores. It can be seen that our implication testing algorithm, though incomplete, is able to prune a large amount of implied DCs.

7.2 Qualitative Analysis

Table 4 reports some discovered DCs, with their semantics explained in English[6]. We denote by Σ_g the golden VDCs that have been designed by domain experts on the datasets. Specifically, Σ_g for Tax dataset has 8 DCs; Σ_g for Hospital is retrieved from [10] and has 7 DCs; and Σ_g for SP Stock has 6 DCs. DCs that are implied by Σ_g are also golden DCs. We denote by Σ_s the DCs returned by FASTDC. We define *G-Precision* as the percentage of DCs in Σ_s that are implied by Σ_g, *G-Recall* as the number of DCs in Σ_s that are implied by Σ_g over the total number of golden DCs, and *G-F-Measure* as the harmonic mean of *G-Precision* and *G-Recall*. In order to show the effectiveness of our ranking function, we use the golden VDCs to evaluate the two dimensions of $Inter$ function in Exp-6, the performance of A-FASTDC in Exp-7. We evaluate C-FASTDC in Exp-8. However, domain experts might not be exhaustive in designing all interesting DCs. In particular, humans have difficulties designing DCs involving constants. We show with $U\text{-}Precision(\Sigma_s)$ the percentage of DCs in Σ_s that are verified by experts to be interesting, and we report the result in Exp-9. All experiments in this section are done on 10k tuples.

Exp-6: Evaluation of $Inter$ score. We report in Figures 3g– 3i G-Precision, G-Recall, and G-F-Measure for Tax, with Σ_s being the Top-k DCs according to $Inter$ by varying the weight a from 0 to 1. Every line is at its peak value when a is between 0.5 and 0.8. Moreover, Figure 3h shows that $Inter$ score with $a = 0.6$ for Top-20 DCs has perfect recall; while it is not the case for using $Succ$ alone ($a = 0$), or using $Coverage$ alone ($a = 1$). This is due to two reasons. First, $Succ$ might promote shorter DCs that are not true in general, such as c_7 in Example 3. Second, $Coverage$ might promote longer DCs that have higher coverage than shorter ones, however, those shorter DCs might be in Σ_g; for example, the first entry in Table 4 has higher coverage than $\urcorner(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$, which is actually in Σ_g. For Hospital, $Inter$ and $Coverage$ give the same results as in Figures 3j, which are better than $Succ$ because golden DCs for Hospital are all FDs with two predicates, therefore $Succ$ has no effect on the interestingness. For Stock, all scoring functions give the same results because its golden DCs are simple DCs, such as $\urcorner(t_\alpha.Low > t_\alpha.High)$.

This experiment shows that both succinctness and coverage are useful in identifying interesting DCs. We combine both dimensions into $Inter$ with $a = 0.5$ in our experiments. Interesting DCs usually have $Coverage$ and $Succ$ greater than 0.5.

Exp-7: A-FASTDC. In this experiment, we test A-FASTDC on noisy datasets. A noise level of α means that each cell has α probability of being changed, with 50% chance of being changed to a new value from the active domain and the other 50% of being changed to a typo. For a fixed noise level $\alpha = 0.001$, which will introduce hundreds of violating tuple pairs for golden DCs, Figure 3l plots the G-Recall for Top-60 DCs varying the approximation level ϵ. A-FASTDC discovers an increasing number of correct DCs as we increase ϵ, but, as it further increases, G-Recall drops because when ϵ is too high, a DC whose predicates are a subset of a correct DC might get discovered, thus the correct DC will not appear. For example, the fifth entry in Table 4 is a correct DC; however, if ϵ is set too high, $\urcorner(t_\alpha.PN = t_\beta.PN)$ would be in the output. G-Recall for SPStock data is stable and higher than the other two datasets because most golden DCs for SPStock data are one tuple DCs, which are easier to discover. Finally, we examine Top-60 DCs to discover golden DCs, which is larger than Top-20 DCs in clean datasets. However, since there are thousands of DCs in the output, our ranking function is still saving a lot of manual verification.

Figure 3m shows that for a fixed approximate level $\epsilon = 4 \times 10^{-6}$, as we increase the amount of noise in the data, the G-Recall for Top-60 DCs shows a small drop. This is expected because the nosier gets the data, the harder it is to get correct DCs. However, A-FASTDC is still able to discover golden DCs.

Figure 3n and 3o show how A-FASTDC performs when the noise is skewed. We fix 0.002 noise level, and instead of randomly distributing them over the entire dataset, we distribute them over a certain region. Figure 3n shows that, as we distribute the noise over

[6]All datasets, as well as their golden and discovered DCs are available at "`http://da.qcri.org/dc/`".

	Dataset	DC Discovered	Semantics
1	Tax	$\neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL$ $\wedge t_\alpha.TR > t_\beta.TR)$	There cannot exist two persons who live in the same state, but one person earns less salary and has higher tax rate at the same time.
2	Tax	$\neg(t_\alpha.CH \neq t_\beta.CH \wedge t_\alpha.STX < t_\alpha.CTX$ $\wedge t_\beta.STX < t_\beta.CTX)$	There cannot exist two persons with both having CTX higher than STX, but different CH. *If a person has CTX, she must have children.*
3	Tax	$\neg(t_\alpha.MS \neq t_\beta.MS \wedge t_\alpha.STX = t_\beta.STX)$ $\wedge t_\alpha.STX > t_\alpha.CTX)$	There cannot exist two persons with same STX, one person has higher STX than CTX and they have different MS. *If a person has STX, she must be single.*
4	Hospital	$\neg(t_\alpha.MC = t_\beta.MC \wedge t_\alpha.MN \neq t_\beta.MN)$	Measure code determines Measure name.
5	Hospital	$\neg(t_\alpha.PN = t_\beta.PN \wedge t_\alpha.PHO \neq t_\beta.PHO)$	Provider number determines Phone number.
6	SP Stock	$\neg(t_\alpha.Open > t_\alpha.High)$	The open price of any stock should not be greater than its high of the day.
7	SP Stock	$\neg(t_\alpha.Date = t_\beta.Date \wedge t_\alpha.Ticker = t_\beta.Ticker)$	Ticker and Date is a composite key.
8	Tax	$\neg(t_\alpha.ST =$ 'FL' $\wedge t_\alpha.ZIP < 30397)$	State Florida's ZIP code cannot be lower than 30397.
9	Tax	$\neg(t_\alpha.ST =$ 'FL' $\wedge t_\alpha.ZIP \geq 35363)$	State Florida's ZIP code cannot be higher than 35363.
10	Tax	$\neg(t_\alpha.MS \neq$ 'S' $\wedge t_\alpha.STX \neq 0)$	One has to be single to have any single tax exemption.
11	Hospital	$\neg(t_\alpha.ES \neq$ 'Yes' $\wedge t_\alpha.ES \neq$ 'No')	The domain value of emergency service is yes or no.

Table 4: Sample DCs discovered in the datasets.

a larger number of columns, the G-Recall drops because noise in more columns affect the discovery of more golden DCs. Figure 3o shows G-Recall as we distribute the noise over a certain percentage of rows; G-Recall is quite stable in this case.

Exp-8: C-FASTDC. Figure 3p reports the running time of C-FASTDC varying minimal frequent threshold τ from 0.02 to 1.0. When $\tau = 1.0$, C-FASTDC falls back to FASTDC. The smaller the τ, the more the frequent constant predicate sets, the bigger the running time. For the SP Stock dataset, there is no constant predicate set, so it is a straight line. For the Tax data, $\tau = 0.02$ results in many frequent constant predicate sets. Since it is not reasonable for experts to design a set of golden CDCs, we only report U-Precision.

	FASTDC			C-FASTDC		
Dataset	k=10	k=15	k=20	k=50	k=100	k=150
Tax	1.0	0.93	0.75	1.0	1.0	1.0
Hospital	1.0	0.93	0.7	1.0	1.0	1.0
SP Stock	1.0	1.0	1.0	0	0	0
Tax-Noise	0.5	0.53	0.5	1.0	1.0	1.0
Hosp.-Noise	0.9	0.8	0.7	1.0	1.0	1.0
Stock-Noise	0.9	0.93	0.95	0	0	0

Table 5: U-Precision.

Exp-9: U-Precision. We report in Table 5 the U-Precision for all datasets using 10k tuples, and the Top-k DCs as Σ_s. We run FASTDC and C-FASTDC on clean data, as well as noisy data. For noisy data, we insert 0.001 noise level, and we report the best result of A-FASTDC using different approximate levels. For FASTDC on clean data, Top-10 DCs have U-precision 1.0. In fact in Figure 3g, Top-10 DCs never achieve perfect G-precision because FASTDC discovers VDCs that are correct, but not easily designed by humans, such as the second and third entry in Table 4. For FASTDC on noisy data, though the results degrade w.r.t. clean data, at least half of the DCs in Top-20 are correct. For C-FASTDC on either clean or noisy data, we achieve perfect U-Precision for the Tax and the Hospital datasets up to hundreds of DCs. SP Stock data has no CDCs. This is because C-FASTDC is able to discover many business rules such as entries 8-10 in Table 4, domain constraints such as entry 11 in Table 4, and CFDs such as c_3 in Example 1.

8. CONCLUSION AND FUTURE WORK

Denial Constraints are a useful language to detect violations and enforce the correct application semantics. We have presented static analysis for DCs, including three sound axioms, and a linear implication testing algorithm. We also developed a DCs discovery algorithm (FASTDC), as well as A-FASTDC and C-FASTDC. In addition, experiments shown that our interestingness score is effective in identifying meaningful DCs. In the future, we want to investigate sampling techniques to alleviate the quadratic complexity of computing the evidence set.

9. ACKNOWLEDGMENTS

The authors thank the reviewers for their useful comments.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[3] M. Baudinet, J. Chomicki, and P. Wolper. Constraint-generating dependencies. *J. Comput. Syst. Sci.*, 59(1):94–115, 1999.

[4] O. Benjelloun, H. Garcia-Molina, H. Gong, H. Kawai, T. E. Larson, D. Menestrina, and S. Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *ICDCS*, 2007.

[5] L. E. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

[7] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. *ICDE*, 2007.

[8] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1):1166–1177, 2008.

[9] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. Tech. Report QCRI2013-1 at `http://da.qcri.org/dc/`.

[10] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[11] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD*, pages 240–251, 2002.

[12] D. Deroos, C. Eaton, G. Lapis, P. Zikopoulos, and T. Deutsch. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill, 2011.

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

[14] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), 2008.

[15] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE TKDE*, 23(5):683–698, 2011.

[16] L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.

[17] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[18] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[19] C. M. Wyss, C. Giannella, and E. L. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK*, 2001.

Temporal Rules Discovery for Web Data Cleaning

Ziawasch Abedjan[§] Cuneyt G. Akcora[♯] Mourad Ouzzani[♯]
Paolo Papotti[♯] Michael Stonebraker[§]
[♯] Qatar Computing Research Institute, HBKU [§] MIT CSAIL
{cakcora,mouzzani,ppapotti@qf.org.qa} {abedjan,stonebraker@csail.mit.edu}

ABSTRACT

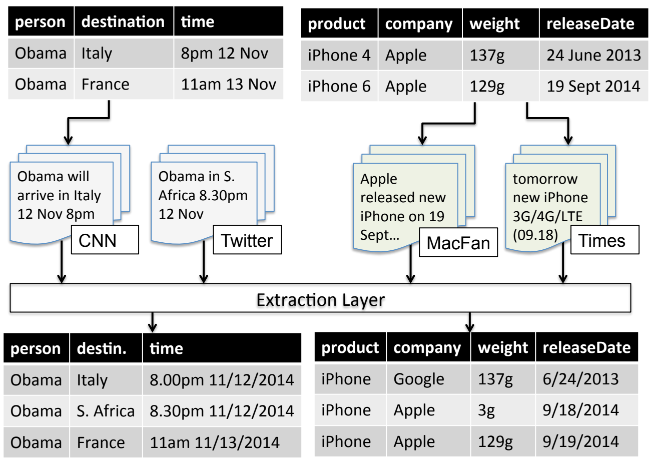
Declarative rules, such as functional dependencies, are widely used for cleaning data. Several systems take them as input for detecting errors and computing a "clean" version of the data. To support domain experts,in specifying these rules, several tools have been proposed to profile the data and mine rules. However, existing discovery techniques have traditionally ignored the time dimension. Recurrent events, such as persons reported in locations, have a duration in which they are valid, and this duration should be part of the rules or the cleaning process would simply fail.

In this work, we study the rule discovery problem for temporal web data. Such a discovery process is challenging because of the nature of web data; extracted facts are (i) sparse over time, (ii) reported with delays, and (iii) often reported with errors over the values because of inaccurate sources or non robust extractors. We handle these challenges with a new discovery approach that is more robust to noise. Our solution uses machine learning methods, such as association measures and outlier detection, for the discovery of the rules, together with an aggressive repair of the data in the mining step itself. Our experimental evaluation over real-world data from Recorded Future, an intelligence company that monitors over 700K Web sources, shows that temporal rules improve the quality of the data with an increase of the average precision in the cleaning process from 0.37 to 0.84, and a 40% relative increase in the average F-measure.

1. INTRODUCTION

With the increasing availability of web data, we are witnessing the proliferation of businesses engaged in automatic data extraction from thousands of web sources with the goal of gleaning useful information and intelligence about people, companies, countries, products, and organizations [30]. It is well recognized that the data cannot be used *as-is* because of errors that are in the sources themselves [15, 28, 29, 33] or that arise with automatic extractors [7, 13].

Proceedings of the VLDB Endowment, Vol. 9, No. 4
Copyright 2015 VLDB Endowment 2150-8097/15/12.



person	destination	time
Obama	Italy	8pm 12 Nov
Obama	France	11am 13 Nov

product	company	weight	releaseDate
iPhone 4	Apple	137g	24 June 2013
iPhone 6	Apple	129g	19 Sept 2014

Obama will arrive in Italy 12 Nov 8pm
CNN
Obama in S. Africa 8.30pm 12 Nov
Twitter
Apple released new iPhone on 19 Sept…
MacFan
tomorrow new iPhone 3G/4G/LTE (09.18)
Times
Extraction Layer

person	destin.	time
Obama	Italy	8.00pm 11/12/2014
Obama	S. Africa	8.30pm 11/12/2014
Obama	France	11am 11/13/2014

product	company	weight	releaseDate
iPhone	Google	137g	6/24/2013
iPhone	Apple	3g	9/18/2014
iPhone	Apple	129g	9/19/2014

Figure 1: From top to bottom: real facts, their representation on the Web, and the extracted data.

Consider the example depicted on the left hand side of Figure 1. Obama attended a dinner in Italy, on Nov 12th 2014 at 8 pm; this is a *real event* and is represented as a tuple in the relation at the top of the Figure. The information is correctly reported on a web page from *CNN* and a web data extractor identifies that a person ("Obama") was in a location ("Italy") at a certain time; this is an *extracted event* (relation at the bottom). However real events can be reported by multiple sources that may or may not agree on the details. In fact, another source reports Obama in South Africa on the same day. As it is unlikely that a person is reported in two different countries within 30 minutes, such a contradiction highlights a problem in the data. In this case, the event was extracted from a social media outlet that did not have a faithful knowledge about the real event. This happens in practice and is studied as the problems of *truth discovery* [33, 15, 28] or *fact-checking* [29]. By enforcing a rule over information from multiple sources, it is possible to gain understanding about the trustability of the sources and, ultimately, about the correct value of interest.

Consider another example for releases of products in the right hand side of Figure 1. The information reported by the source *Times* is a real event, but an error in the extractor led to an incorrect weight, namely *3g*, which in fact is a type of network supported by the phone. Detecting such problems can help identify faulty extractors [7, 13].

The two above examples highlight that identifying quality problems enables analysis over the sources, the data values, and the extractors. These analytics tasks usually rely on declarative rules (such as key constraints) for detecting problems in the data. For example, the fact that a product is always released by a company can be expressed with a functional dependency (FD), i.e., product $\rightarrow$ company. In the above example, the company releasing the phone cannot be both Google and Apple. Heuristics exploiting the redundancy are usually used to determine the correct value (the truth) [28]. However, there are other errors that can be identified only through *temporal functional dependencies*, which are FDs that restrict the rule on the temporal dimension [21]. For example, a domain expert may come up with a rule stating that a person cannot be reported arriving in two countries at the same time (person $\rightarrow$ destination in a 1-hour window), or that the same product cannot have different weights reported at the release date (product $\rightarrow$ weight in a 6-month window).

Coming up with these rules with the correct duration value for "same time" is not trivial. A conservative choice for this duration in a rule, such as "within a minute", leads to undetected errors in the Obama example. On the contrary, a high value for duration, such as "two days", does capture the problem, but would mark as errors all the tuples in the example, including the correct ones with Italy and France. Similar challenges arise for product release, a time window of one day for the weight would not capture the problem with *3g*. Moreover, durations depend on the entity at hand. For instance, Obama travels more frequently and faster than most people, so he should have a different temporal rule with a smaller time window.

Discovering constraints has been well studied in the literature [32, 1, 23, 8, 19]. However, a recurring assumption in these existing techniques is that data is either clean or, at worst, has a small amount of noise. Obviously, such assumptions do not hold for data extracted from the web due to the compounded effects of noise coming from the sources and errors made by the extractors. Moreover, even when it is possible to mine approximate dependencies over such dirty data, there is no algorithm to discover useful time-windows, or *durations*, to identify errors for the different events, e.g., a person is not reported traveling to two countries in a *1-hour* window. Without such a time dimension, rules are not usable, as discussed above.

In this paper, we present AETAS[1], our solution to overcome the above challenges by relying on two basic concepts: (i) the notion of approximation for the discovery of functional dependencies that hold for most of the data, and (ii) outlier detection techniques for the discovery of the durations. In a nutshell, we first create a set of approximate FDs that are valid in the smallest meaningful time interval. The dependencies are then ranked with an association measure, and validated by human experts. For each validated rule, we create a distribution of durations for all the objects in the data, e.g., how much time is observed within two consecutive destinations for every person, and mine it to compute the duration that identifies the lowest extreme values. This duration is then used as the time window for the rule to identify temporal outliers.

Our contributions are as follows:

1) We formulate the problem of discovering temporal functional dependencies for data cleaning (Section 2), and present techniques to discover approximate FDs based on statistical properties of the data (Section 3).

2) Given a rule, we mine the duration that lead to identifying temporal outliers. We tackle the problem of the sparseness of the data with value imputation, and reduce the noise by enforcing the rule in the smallest meaningful time bucket (Section 4). We also mine rules with constants (akin to conditional functional dependencies) such that specific durations can be used for specific entities.

3) We show over real and synthetic datasets that our techniques for approximate dependencies and duration discovery outperform alternative approaches in terms of quality. In particular, our durations lead to improvement in the data cleaning process compared to FDs, with an increase of the average precision in the repair of the temporal data from 0.37 to 0.84, and a 40% relative increase in the average F-measure (Section 5). Moreover, our technique discovers durations that lead to higher F-measure than the baselines, including the durations collected from a group of users.

We discuss related work in Section 6, and in Section 7 we draw some conclusions and list directions for future work.

2. RULE DISCOVERY FOR WEB DATA CLEANING

We first describe the kind of web data we are dealing with. We then define the syntax and semantics of temporal dependencies, give a definition of data cleaning, and define the problem of the rule discovery for web data cleaning.

From web pages to structured data. We are interested in event data collected from the Web by monitoring news media. Examples of such data include GDELT (gdeltproject.org) and Recorded Future (www.recordedfuture.com). Given a web page, the organization in charge of the event database runs extractors to produce structured data for different events. Examples of events include people traveling to destinations, company acquisitions, and occurrences of armed attacks. Figure 1 exemplifies data extracted from text in six web pages: three occurrences for event *PersonTravel* with person *Obama* as the only entity, and three occurrences for *ProductRelease* with product being the entity *iPhone*. In general, an entity may be an instance of a person, a location, a company, and so on. In the following, we assume that entity recognition from the text has been already performed. In addition to the event type specific attributes, *e.g.,* company, destination, all events have a timestamp attribute, such as time and releaseDate. We assume that all of these attributes may contain errors.

Temporal Functional Dependencies. We focus on a specific form of temporal functional dependencies similar to those described in [21]. We assume a total ordering on the time attribute t, and that there is a mapping $f()$ that linearizes the different time values into integers. For example, the value r[t] = (h,m,s) could be mapped to seconds via f(h,m,s) = 3600h + 60m + s. A *time interval* Δ is a pair with a minimum and a maximum value (for examples in hours), m and M, respectively, with $m \leq M$.

Given the pair $< U, t >$ with a fixed set $U = \{A_1, \ldots, A_n\}$ of event type attributes and the time attribute t, a tuple over $< U, t >$ is a set of $< r = \{A_1 : c_1, \ldots, A_n : c_n\}, t : c_t >$, where c_i is a constant. A relation I is a finite set of tuples over $< U, t >$.

[1]From "Omnia fert aetas", *Time cancels everything.*

Definition 1. *Let X, Y be two subsets of attributes from U, Δ a time interval, and π the permutation of rows of I increasing on the time value. A* temporal functional dependency *(*TFD*) over U is an expression $X \wedge \Delta \rightarrow Y$ that is satisfied if for all pairs of tuples $r_\pi, r_{\pi+1} \in I$, s.t. $r_{\pi+1}[t] - r_\pi[t] \in \Delta$, when $r_\pi[X] = r_{\pi+1}\ [X]$, it is the case that $r_\pi[Y] = r_{\pi+1}\ [Y]$.*

The subsets of attributes X and Y are referred to as left-hand side (LHS) and right-hand side attributes (RHS), respectively. When referring to values of X and Y attributes, we shall use the terms *reference value* and *attribute value*, respectively.

Example 1: The rule "a product cannot be released with two different weights in a time window of a year" defined over event *ProductRelease* can be stated as follows: $\mathsf{product} \wedge (0, 1\ \mathsf{year}) \rightarrow \mathsf{weight}$, where Δ is the pair $m = 0$ and $M = 1$ *year*. In Figure 1, *ProductRelease* events show conflicting weight values *3g* and *129g* on release dates *09/18/2014* and *09/19/2014* for product *iPhone*. □

While most of the entities for a given event abide by the same duration in a rule, some entities may require specific duration values. For example, in the case of *ProductRelease* events, new *iPhone* models are sometimes released with an interval of time shorter than a year, while for cars the interval is much longer (*e.g.,* BMW X5 car model is renewed every 6 years). Thus, in the same spirit of conditional function dependencies (CFDs) [5], we are also interested in TFDs that apply on subsets of tuples. We therefore extend the language to consider constant selections in the left-hand side, such as $\mathsf{product}[\text{"}iPhone\text{"}] \wedge (0, 8\ \mathsf{months}) \rightarrow \mathsf{weight}$. This is equivalent to having views for specific entities and applying the TFD on the view induced by the selection.

Data Repairing. While TFDs can be used in multiple applications, such as database design, our focus is on data quality scenarios. Data repairing is the application we will use in the following to evaluate the quality of the discovered dependencies. Given a database instance I of schema $\mathbb{R}$ and a dependency φ, if I satisfies φ, we write $I \models \varphi$. If we have a set of dependencies Σ, $I \models \Sigma$ if and only if $\forall \varphi \in \Sigma, I \models \varphi$. A *repair* I' of an inconsistent instance I is an instance that satisfies Σ. A repair solution is not unique, as discussed in the following example.

Example 2: Consider a different instance D for *ProductRelease* and the FD d_1 : $\mathsf{product} \rightarrow \mathsf{company}$. Value errors are reported in bold.

D	product	company	weight	releaseDate
t_1 :	iPhone	Apple	137g	**10am 6/24/2014**
t_2 :	iPhone	**Google**	129g	3pm 9/18/2014
t_3 :	iPhone	Apple	129g	4pm 9/19/2014

If we check the dependency over the data, we get the following pairs of violating tuples: (t_1,t_2),(t_2,t_3).
Two possible, alternative repairs are $R_1 - R_2$, as follows:

R_1	product	company	weight	releaseDate
t_1 :	iPhone	Apple	137g	**10am 6/24/2014**
t_2 :	iPhone	*Apple*	129g	3pm 9/18/2014
t_3 :	iPhone	Apple	129g	4pm 9/19/2014

R_2	product	company	weight	releaseDate
t_1 :	iPhone	*Google*	137g	**10am 6/24/2014**
t_2 :	iPhone	**Google**	129g	3pm 9/18/2014
t_3 :	iPhone	*Google*	129g	4pm 9/19/2014

Updates (in italic) in R_1 and R_2 make the new instance valid for d_1. An alternative repair strategy deletes tuple t_2 in R_1 or tuple t_1, t_3 in R_2.

Consider also a TFD d_2: $\mathsf{product}[\text{"}iPhone\text{"}] \wedge (0, 8\mathsf{months}) \rightarrow \mathsf{weight}$. A possible repair for violating tuples (t_1,t_2), (t_1,t_3) is by updating the value of weight for t_1 to *129g*, or to delete t_1. □

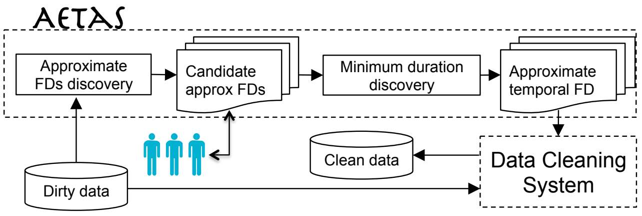


Figure 2: Architecture of the AETAS **system.**

Since the number of possible repairs is usually large and possibly infinite, a minimality principle is oftentimes used to identify desirable repairs for the *data cleaning problem* [22]: given a database I and a set of dependencies Σ, compute a *repair* I_r of I such that $I_r \models \Sigma$ (consistency) and their distance *cost*$(I_r,\ I)$ is minimal (accuracy). Depending on the distance function, the desired repair is the one with the minimal number of cell updates, or the one with minimal number of tuple deletions. Computing minimal repairs is NP-hard to be solved exactly for FDs [4, 24] which led to several heuristics-based methods [10, 12, 16, 24].

Discovering temporal dependencies. Given a relational schema $\mathbb{R}$ and an instance I, the discovery problem for TFDs is to find all valid TFDs that hold on I. Since web data is noisy in nature, we are interested in the approximate version of the problem, i.e., find all valid TFDs, where a rule r is valid if its support has a value higher than a given threshold δ. To solve this problem, we developed AETAS, a system to discover TFDs from web data.

Figure 2 shows the architecture of the system and the main steps in our solution. Given a noisy dataset, we first discover approximate functional dependencies, i.e., traditional FDs that hold on most of the relation within a given atomic duration t_α. The use of the atomic duration removes the temporal aspect of the relation so that dependencies can be discovered purely in terms of record attributes.

Given a set of approximate FDs, we rank them according to their support to assist the user in their validation. A user can either reject a suggested approximate FD, or validate it as being a simple FD or a TFD. For a validated TFD, we then discover its corresponding time interval, including values that only hold for specific entities as we discussed previously. Since the data is dirty, we cannot just examine consecutive occurrences for each entity and collect the minimum duration. Therefore, we compute the distribution of the durations and mine it to identify the minimum duration that would eventually cut-off the outlying values, *i.e.,* data that is invalid. This minimum duration is then assigned to M and, together with default $m = 0$, define Δ for the approximate FD at hand.

Finally, FDs and TFDs are fed to a constraint based data cleaning system, which takes the rules and the noisy data as input and outputs a consistent updated dataset.

3. FD DISCOVERY OVER DIRTY DATA

Two main characteristics of web data prevent us from using traditional dependency discovery algorithms. First, most of these algorithms assume that the data is clean. As we work with dirty web data from multiple independent sources, this assumption does not hold. There have been some work to tolerate some dirtiness up to a certain threshold on the percentage of not conforming tuples [8, 19]. However, dirtiness in real (web) data is so high that the corresponding threshold leads to the discovery of very general rules that are not valid in practice. For example, our test dataset has noise up to 26% wrt the number of tuples (Table 1 in Section 5). A threshold of 26% leads to the discovery of several key constraints and multiple functional dependencies that do not hold semantically.

Second, temporal data contains reference values that change over time, such as Obama with correct values "Italy" and "France" at two different timestamps. Because of this temporal nature, traditional FDs do not apply over the relations with extracted data for many events.

We introduce next how we model the data and then how we tackle the above problems with our approach for discovering approximate FDs.

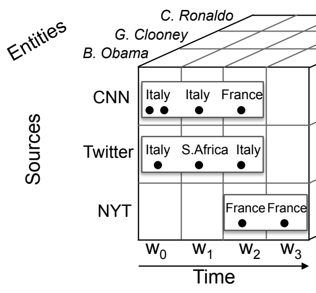


Figure 3: A dependency cube: w_0 is a 1-day time bucket, s_0 is source *CNN*, a_0 is entity *B. Obama*, {*Italy, S. Africa, France*} are attribute values.

Dependency cube. We start by considering all the possible dependencies with one attribute in the LHS and one attribute in the RHS. For each potential dependency $\mathsf{X} \rightarrow \mathsf{Y}$, we make the time dimension (attribute t) discrete by creating time buckets with the size of an atomic time duration. Given these time buckets, we define a *dependency cube* over four data dimensions: data sources S, time buckets W, reference values X, and attribute values Y. Figure 3 shows a dependency cube for the Obama example:

- The x axis is divided into homogeneous time buckets $w_i \in W$ (e.g., 1 hour).

- The z axis reports different reference values $x \in X$ (e.g., B. Obama, iPhone).

- The y axis reports different sources $s_i \in S$ (e.g., CNN).

- The reported cell values for a certain time bucket, source, and entity are attributes values $y \in Y$ (e.g., Italy, Apple).

The size of each dimension in this cube can be large. For example, Recorded Future continuously collects data from more than 0.7 Million web sources. However, due to how events are reported on the web, the data is very sparse and the size of the dependency cube is manageable in practice.

For a reference value $x \in X$, the sequence of values of Y reported by one source over time constitutes a stripe. For example, in Figure 3, source NYT reports two Y values for reference value $B.Obama$ in time buckets w_2 and w_3. For a reference value $x \in X$, the union of stripes from all sources constitutes a plate. For a time bucket w_i of a given dependency over R, we define the time slice R_i as the list of Y values for all X values reported within the time bucket w_i.

A potential dependency holds for the cube if, for each plate and for each stripe, it is true that $\mathsf{X} \rightarrow \mathsf{Y}$. If the dependency holds only for a specific plate for reference value x_i (i.e., a specific entity), then it is a constant dependency $\mathsf{X}[x_i] \rightarrow \mathsf{Y}$.

Implication discovery. Considering the aforementioned noise and temporal problems, we devise an algorithm that works with dirty, temporal data for the discovery of TFDs. We discover implications by first fixing an atomic time duration t_α such that we can mine the dependencies that hold within a time bucket. We observe that if a TFD holds for a certain duration Δ, it also holds for durations $\Delta' \leq \Delta$. The best bucket size is the smallest one that contains enough evidence to do the mining. Moreover, the value of the atomic duration cannot be more fine grained than the time granularity of the timestamps in the data. In our datasets, the granularity is up to milliseconds, but the data is too sparse to mine in such a small granularity, we therefore use $t_\alpha = 1$ *hour*. The atomic duration t_α is application-dependent and is an input parameter for the algorithm.

Given a t_α value, we partition the data and create time slices $\mathcal{R} = \{R_1, R_2, ..., R_N\}$. Given a time slice R_i, we employ an association rule based method to detect 1-to-1 and many-to-1 implications. More specifically, we use *normalized pointwise mutual information* (NPMI) [6], a standard association measure in collocation extraction, for implication discovery. In a time slice, NPMI of a pair of reference-attribute values $x \in X$ and $y \in Y$ is defined as:

$$i(x,y) = ln(\frac{P(x,y)}{P(x) \times P(y)})/-lnP(x,y)$$

where $P(x,y)$ is the joint probability of reference value x and attribute value y, and $P(x)$ is the marginal probability of reference value x. Intuitively, given a pair of outcomes x and y that belong to discrete random variables X and Y (assumed independent), the PMI quantifies the discrepancy between the probability of their coincidence given their joint distribution and their individual distributions. Its normalized version, NPMI, can have the following values: if x and y only occur together $i(x,y) = 1.0$, if x and y occur independently $i(x,y) = 0$, and -1 if they never co-occur. We use this score as an indicator of their correlation in the following.

We learn the implication $x \rightarrow y$ for a pair of values in the given time slice. In order to find the $X \rightarrow Y$ implication, we need to generalize the NPMI value over all value pairs of the two attributes. If the implication holds, we expect the NPMI value of each pair to be positive (i.e., the sign of the 1-to-1 implication) and, overall, $i(X,Y)$ close to 1.0. In fact, three factors can decrease NPMI values even in presence of real implications. First, multiple reference values can have the same attribute value in a given time slice (i.e., many-to-1 implication). For example, two persons can be in the same city at the same time. Second, because of a small bucket size, time slices may contain few instances about the same reference values. For example, in a bucket all events might be about Obama traveling to France. We employ a decision rule to overcome single reference value by assigning

$i(X,Y) = 1.0$ when $|X| = 1$. Third, dirty data can introduce different attribute values for the same reference value (i.e., 1-to-many occurrences), as in the example with Italy and S. Africa for Obama. As we assume dirtiness in the data, we need to tolerate this noise. Given these three possible causes, some value pairs have low NPMI values, thereby reducing the NPMI value of attributes, i.e., $i(X,Y) < 1.0$. This is a strong signal for the discovery of correlations and we exploit it in our algorithm.

From implications to dependencies. Given a list of NPMI values of multiple time slices, our next task is to decide whether the implication $X \rightarrow Y$ holds on enough time slices to be considered a dependency. Hence, we aggregate the expected value of NPMI values over time slices and compute a score. The score is then used to rank the output for user validation. Aggregation of NPMI values by expected value is weighted wrt the number of event instances (i.e., tuples) in time slices, such that an implication is penalized if it does not hold over time slices with large numbers of event instances.

To prune the number of results in the output, we also allow as input an optional user-defined significance threshold δ. In this case, we declare an implication to be a dependency if its aggregated score is higher than the threshold.

Example 3: Consider the case where an event R has instances distributed over a 36 hour period. Given $t_\alpha = 12\ hours$, we create three time slices R_1, R_2 and R_3, and compute their NPMI values to be 0.95, 0.3 and 0.5. Probabilities of event instances belonging to these time slices are $P(R_1) = 0.8$, $P(R_2) = 0.15$, and $P(R_3) = 0.05$. The expected NPMI value $E = 0.95 \times 0.8 + 0.3 \times 0.15 + 0.5 \times 0.05 = 0.83$. For $\delta = 0.7$, we assert that the implication $X \rightarrow Y$ holds. A value of 0.7 is usually used in practice [17]. □

Early Termination. If a threshold δ is defined, an implication can be declared to hold or to be pruned by considering a smaller number of slices. We thus stop NPMI computations if the remaining slices will not carry the expected score below or above the significance threshold δ. Consider the case when NPMI values of x out of n slices have been computed. In the best and worst cases, all the remaining slices can have NPMI values 1 or 0, respectively. If an implication does not hold even in the best case, or holds even in the worst case, we do not need to compute the NPMI values of the remaining slices. Otherwise we continue our computation. Formally, given a total of n slices for the implication $A \rightarrow B$, we terminate computations at the x^{th} slice with

$$\begin{cases} A \not\rightarrow B & \text{if } \delta > \\ & \sum_{j=1:x} i_j(A,B) \times P(j) + \\ & \sum_{k=x+1:n} i_k(A,B) \times P(k) \\ A \rightarrow B & \text{if } \delta \leq \sum_{j=1:x} i_j(A,B) \times P(j) \end{cases}$$

where δ is the significance threshold, $P(k)$ is the probability of an event instance being in the time slice k, and $i_k(A,B)$ is the NPMI value of the k^{th} slice.

Example 4: Consider the scenario of three time slices in Example 3. After computing the NPMI value of R_1 as 0.95, we can terminate computations for a significance threshold of $\delta = 0.7$ because the expected value $E = 0.95 \times 0.80 = 0.76$ is already above δ. □

```
Data: A relation $R$ of attributes $A$, $B$; atomic time
      length $t_\alpha$; (threshold $\delta$)
Result: A score for the dependency
1  $npmi = 0$, $current := 0$;
2  Create time buckets $\mathcal{R} = \{R_1, ..., R_n\}$ of $R$ with $t_\alpha$;
   // Iterate on each slice;
3  foreach $R' \in \mathcal{R}$ do
4      $current \leftarrow current + |R'|$;
5      $v = 0$;
       // Decision rule;
6      if $|R'.B| = 1$ then
           // Add an NPMI value of 1.0;
7          $v = 1.0$;
8      else
9          foreach $a \in R'.A$ do
10             foreach $b \in R'.B$ do
11                 $i(a,b) = \frac{ln(P(a,b)/(P(a)*P(b)))}{-ln(P(a,b))}$;
12                 $v = v + P(a,b) \times i(a,b)$;
       // Add expected value of the slice
13     $npmi = npmi + v \times \frac{|R'|}{|\mathcal{R}|}$;
       // Termination;
       // 1. Dependency will not hold;
14     if $\delta \neq null \wedge \delta > npmi + \frac{|\mathcal{R}|-current}{|\mathcal{R}|}$ then
15         return 0;
       // 2. Dependency will hold;
16     if $\delta \neq null \wedge \delta \leq npmi$ then
17         return 1;
18 return $npmi$;
```

Algorithm 1: Implication detection with NPMI.

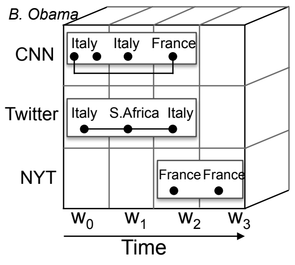
Algorithm. We now give a description of our approximate dependency discovery algorithm. Given two attributes A and B from an event R, we use Algorithm 1 to compute their NPMI score, or to find whether a dependency holds over the two attributes, if a threshold is given. The algorithm takes as input an atomic duration t_α, two attributes A, B from a relation R, and an optional significance threshold δ. The algorithm is called twice for each direction, namely $A \rightarrow B$ and $B \rightarrow A$. If an implication is found for the attributes, only one, or both of these dependencies may hold.

The algorithm starts by time bucketing the relation into smaller relations (Line 2). From Line 3, we find the strength of the implication within the slice. If the sub-relation contains a single attribute value, the decision rule in Line 2 assigns a NPMI value of 1.0 to the slice. Otherwise, we compute NPMI values of each (a,b) pair in Line 11. Line 12 adds the NPMI value to the expected value of the slice.

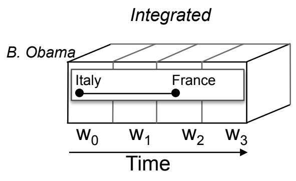
Once NPMI values of all pairs have been computed, the NPMI value of the slice is added to the expected value of the whole dependency in Line 13. If δ is defined, we check the termination conditions in Lines 14 and 16. In Line 14, we compute the NPMI value for the ideal case where all the remaining slices will be 1.0. Similarly, Line 16 checks whether the expected value is already above the threshold. In both cases, we stop the computations early if the condition holds and return 0 or 1 accordingly. If δ is not defined, we return the NPMI value for the dependency to be used for ranking and user's consumption. Once the user has selected the dependencies that are TFDs, we process then for duration discovery, as described in the next Section.

4. TIME DURATION DISCOVERY

Given an approximate FD $X \wedge \Delta \rightarrow Y$ with $\Delta_t=(0,t_\alpha)$ for an event, the goal of duration discovery is to expand the atomic duration t_α to the correct minimum duration M in Δ. In the ideal case, there exists one and only one Δ such that no reference value $x \in X$ can change its attribute value $y \in Y$ within a time interval $(t_y, t_y + m)$, where t_y is the reported time of value y.



(a) Baseline durations



(b) Projected durations

Figure 4: Stripes integration for duration discovery.

A naive approach for duration discovery is to take each stripe of a dependency cube, and find the time it takes for an attribute value to change from $y_1 \in Y$ to $y_2 \in Y$, i.e., $t_{y_2} - t_{y_1}$. This results in a list of time difference values. Then, the minimum value among all the time differences can be chosen as the M in Δ. This approach is shown in Figure 4(a) for a single plate, where value changes in stripes are highlighted.

A major assumption in the naive approach is that web sources correctly report attribute values. When the data comes from non-authoritative web sources, this assumption can easily be broken. A more robust approach is to exploit the evidence coming from multiple sources such that the accuracy of an attribute value can be "verified". The idea is to first repair the data within a time bucket with the evidence coming from multiple sources. We then compute durations on a "clean" integrated stripe, as in Figure 4(b). Below, we first describe how to repair data in the buckets and then give the full discovery algorithm.

Repair step. Given an approximate FD $A \rightarrow B$, we create a plate p for each reference value $a \in A$, and the plate is partitioned into time buckets of size t_α. Each bucket $w_n \in p$ has a time slice of attribute values R_n reported by sources where $|R_n| \geq |distinct(R_n)| \geq 1$. A new stripe I is created, where the results of the integration will be reported. In a bucket, if there exists a b' such that $mode(R_n) = b'$, then the corresponding w_n bucket for I is updated with b'. If there is no majority, the value in bucket w_{n-1} is assigned to w_n for I.

Figure 5 shows a window repair for three sources. In the figure, the value *Italy* from source *Twitter* is less frequent than *France*, so it is not in the result. Although our repair approach uses a simple majority voting scheme, any repair algorithm can be plugged into the system, for example by using truth discovery algorithms [15, 28, 29, 33] or by involving domain experts.

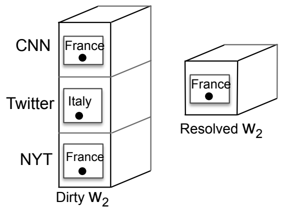


Figure 5: Repair step.

Time durations. Given the integrated plate, we compute a distribution of time durations between consecutive, distinct attribute values for every reference value. Figure 4(b) shows time durations on I, the stripe with the outcome of the repair step. Even with repaired values in time buckets, reference values have varying durations for the same temporal dependency. Two factors impact the observed durations:

- **Dirty data.** Sources can report conflicting values in two consecutive buckets that cannot be detected by local repairs. This problem raises many short durations. For example, the *Twitter* stripe in Figure 4(a).

- **Reporting frequency.** Although a reference value changes its value in the real world, sources may not report it. In our dataset, only a small set of entities, such as political leaders, have their changes reported frequently. This leads to some durations that are longer than the real time windows between two occurrences of an event.

As a result of these factors, time durations constitute a non-uniform distribution $D(x, y)$, with a range of $[t_\alpha, |W|]$. Our goal is to mine a duration that would remove the outlying values from this distribution.

```
 Data: A dependency cube $C$ for $A \rightarrow B$, a cut-off
       value $c$ with $1 \leq c \leq 100$
 Result: A time duration $M$
1  Define D(A,B) to be an empty duration list;
2  foreach plate $p \in C$ do
3      Define $I$ to be an empty stripe with $|I|$ equals to
       the # of buckets in $p$;
4      foreach non-empty bucket $w_i \in p$ with time slice $R_i$
       do
5          $b' \leftarrow Mode(R_i)$;
6          if $b'$ is not null then
7              update bucket $w_i \in I$ with $b'$;
8          else
9              update bucket $w_i \in I$ with value in $w_{i-1}$;
10     $l = 0$;
11     for i=0:length(I) do
12         if $value(w_i) \neq value(w_l)$ then
13             add $i - l$ to $D(A, B)$;
14             $l = i$;
15 return $percentile(D(A, B), c)$;
```

Algorithm 2: Duration discovery for a temporal rule.

Algorithm. Taking into account the above factors, we propose an approach for time duration discovery in Algorithm 2. A dependency cube C for an approximate functional dependency $A \rightarrow B$ is given as input as well as a cut-point $1 \leq c \leq 100$ for the identification of the duration that removes outliers. We use as default value of 10 for the cut-point, as this is a common value used for trimming of outliers (e.g., interdecile range). We also show in the experimental study how this parameter affects the results.

In a nutshell, the algorithm first corrects the erroneous attribute values reported by the sources for each plate in an integrated stripe I (Lines 3-10), and then adds the durations over I to a duration list (Lines 11-15). The output is the minimum time duration value M that removes outlying durations for the time dependency $A \rightarrow B$.

The algorithm iterates over each plate (entity) in the cube (Line 2). For each plate, we create a new, empty integrated stripe (Line 3). In the time slice for each bucket in the plate, depending on the source quality, sources can agree or disagree on the attribute value. To alleviate the problem of sources with poor quality, we employ a repair step (Line 5). In a simple analysis, if there is a single most frequent value, this is assigned to the integrated stripe (Line 7). If a majority cannot be determined, the values are ignored and the value imputation is done with the previous values in the strip (from the Occam's razor principle) (Line 9).

After the repair process, the algorithm works on the integrated stripe I and extracts time durations between different consecutive attribute values (Lines 11-16). Parameter l in Line 11 records the first point in time when the stripe reports an attribute value. In the following windows, the source may report the same value, or change it. If the value changes, the l parameter is used to compute the time difference between the two different attribute values.

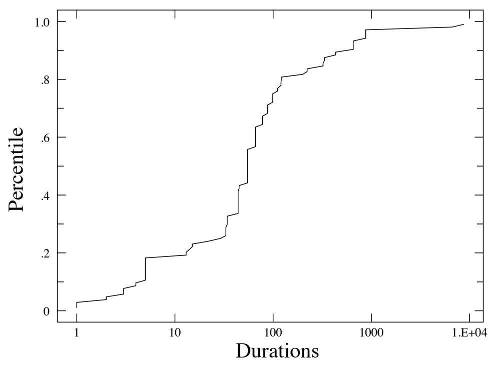


Figure 6: Duration discovery with percentile plot.

With multiple time durations from multiple integrated stripes, we use a trimming (truncation) function, namely the c^{th} percentile, to compute the duration M. The intuition is that trimming identifies outlying values (*trimmed minima*), and we are after the duration that identifies such outliers. For example, in the probability distribution of time durations, the 10^{th} percentile specifies the time duration value at which the probability of the time durations is less than or equal to 0.1. We report an example of a minimum duration of six hours (x axis) discovered with the 10^{th} percentile (y axis) in Figure 6.

Timestamps or Values? It is worth observing that the algorithm above aligns the timestamps and then compares values to perform the analysis. An alternative approach is to align the values, after they have been ordered, and then perform the counting of the durations. We will show in the experimental section that an algorithm that relies on values for alignment performs worse than the one we propose based on time alignment. In particular, we implemented a variant of sequence alignment from the bioinformatics domain [26]. Taking all stripes from a plate, we align attribute values of each pair of stripes. The alignment process creates two temporary stripes that are the aligned versions of the input pair; the temporary stripes both report the same value at a given time, or one of them reports nothing (i.e., reports a *null* value) whereas the other reports an attribute value. The alignment approach mines durations between value changes only when the change is reported by both stripes.

Conditional Durations. As we mentioned earlier, some TFDs may only apply to a subset of entities because some, usually popular, entities have more frequent attribute changes at smaller time frames. To discover the corresponding durations, we track the duration sequences of a specific entity and compute its duration by mining M only with values from their plate. As the minimum duration is computed based on only the sequences that refer to a specific entity, this entity has to be popular, i.e., there must be at least some observations to compute a distribution. As it is common in statistics, we require 30 observations for the computation of the percentile. Therefore, we compute constant rules only for entities with at least 30 durations in their plate. For instance, while 24 hours is the minimum duration that removes outliers for the majority of persons in our person travel dataset, persons such as *Vladimir Putin* or *Ban Ki Moon* should have smaller minimum durations, and this is reflected with their constant rules.

5. EXPERIMENTS

In the following, we first study the performance of our solutions and compare them to baseline alternatives using a real dataset provided by Recorded Future. We then study our algorithms in depth with synthetic data.[2]

We measure the effectiveness of both implication and duration discoveries. We also measure the execution time needed by the algorithms. Experiments were conducted on a Linux machine with 24 1.5GHz Intel CPUs and 48GB of RAM. All algorithms have been implemented in Java with Heap size set to 12GB.

Algorithms. For the implication discovery, we compare our proposal (Section 3) to CORDS [20], a state of the art algorithm for the discovery of approximate dependencies. We test both methods for time slices, therefore they do not have to deal with the time dimension. For the duration discovery, we test the following algorithms:

- REPAIR-OUTLIERS (RO), our method reported in Section 4 where the durations collection is performed over a unified view of every plate. These "clean" durations are then used for mining the minimum duration M that isolates outliers.

- NO REPAIR-OUTLIERS (NR), a variant of our algorithm where we do not perform repair; we collect all durations over the stripes to mine M. This method shows the role of the repair.

- ALIGNMENT-OUTLIERS (AL), a variant of sequence alignment in genomics [26] (Section 4). This is an alternative method that trusts values more than time, as the former are used for alignment.

- NO REPAIR-PROBABILITY (NP), an adaptation of the disagreement decay from the duration discovery algorithm in [27]. Disagreement decay is the probability that an entity changes its value within time Δt. For an increasing Δt, the probability of decay $0 \leq p \leq 1$ also increases. The authors use a probability distribution D for various Δt values [27]. We use a probability cut-point δ_c, such that we select the smallest $\Delta t'$ that satisfies the condition $D(t') \geq \delta_c$ as the duration for our temporal dependency.

[2]The annotated real-world data and the program to generate synthetic data can be downloaded at `https://github.com/Qatar-Computing-Research-Institute/AETAS_Dataset`

Event	# Atts	# Ground Rules	Rules Coverage	Rule Annotated Over Data	# Annotated Tuples	% Errors
Acquisition	3	4	1.00	acquired company → acquirer company	217	26
Company Employees #	2	2	0.50	company → employees number	198	26
Company Meeting	5	6	0.80	company → meeting type	179	17
Company Ticker	3	2	0.67	ticker → company	1,906	4
Credit Rating	4	6	1.00	company → new rank	150	8
Employment Change	7	11	0.86	person → company	186	14
Insider Transaction	22	210	0.95	insider → company	150	0
Natural Disaster	2	1	0.50	location → natural disaster	250	10
Person Travel	6	2	0.67	person → destination	372	21
Political Endorsement	2	1	0.50	endorser → endorsee	199	11
Product Recall	5	12	0.67	product → company	216	5
Voting Result	2	2	1.00	location → winner	215	10

Table 1: Events, correct rules, and annotated rule used in the real data evaluation.

5.1 Real Data

Dataset. We obtained a 3-month snapshot of data extracted by Recorded Future, a leading web data analytics company. The dataset has about 188M JSON documents with a total size of about 3.9 TB. Each JSON document contains *extracted events* defined over entities and their attributes. An entity can be an instance of a person, a location, a company, and so on. Events have also attributes. In total, there are 150M unique event instances excluding meta-events such as co-occurrence.

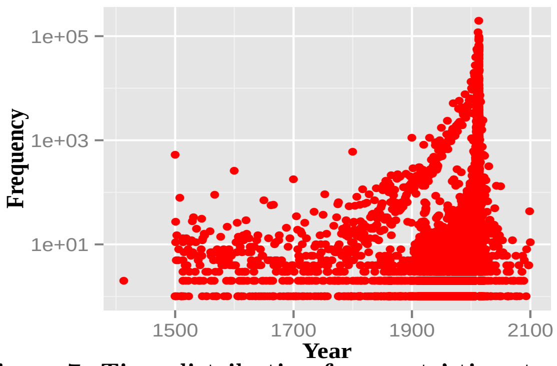


Figure 7: Time distribution for events' timestamp.

The data contains events from circa *15th* to *21st* century. Figure 7 shows events across years, where each point corresponds to the number of events in a day. Most of the events occur around Fall 2014, because the data is mostly around the 3 months covered by the snapshot. Starting from January 2015 onward, the reported events are future forecasts.

Metrics. We crafted ground rules for all events to evaluate the discovery algorithms. These rules are dependencies that are semantically correct. We report their number for every event in Table 1, where we also report as "Coverage" the ratio of the distinct number of attributes in the set of rules to the total number of attributes for that event. We then use this ground truth to evaluate the precision and recall over the top-k dependencies returned by the algorithms.

In the case of TFDs, crafting also their ground durations is more challenging since there are rarely clear duration values that can be set. We argue that it is not correct to compare the discovered duration value versus an arbitrary manually set ground truth; different persons may come up with different durations. We also show that a combination of these arbitrary values does not lead to the best duration value. Distance between the discovered duration and the real one can be better measured in a controlled environment with synthetic data, as we discuss in Section 5.2. We therefore evaluate the quality of the duration discovery on real data in a different way. Instead of measuring the distance between durations, we evaluate the quality of a duration by measuring its effect in a target application. In particular, for a TFD, we run it multiple times in the same data cleaning tool with different durations, and measure the quality of the obtained repairs. We use BigDansing [22] , a data cleaning system that can handle TFDs with the repair semantics discussed in Section 2.

For this validation, we manually created ground truth for a large sample of the data. We randomly picked 12 event types and discovered rules over their corresponding instance datasets. For each selected rule, we sampled 1% of the tuples, making sure that (i) at least 150 tuples comprising 5 different entities were selected, (ii) both popular and rare entities were selected, and (iii) at most 150 tuples were annotated for one single entity. Each tuple has been manually validated with sources such as Twitter accounts for persons, LinkedIn official web pages for companies, and stock brokers. Table 1 gives the details about the selected events, representative rules, and size of the samples.

Given the ground data and the results of the cleaning, we follow common practices from the data cleaning literature [3, 10] to evaluate the quality of the obtained repair. We count as *correct detections* the updates in the repair that correctly identify dirty values. This corresponds to measuring the effectiveness of repairs based on delete-semantics, where tuples with errors are removed. We count as *correct changes* the updates in the repair that are equal to the original values in the ground truth. Based on these two metrics, we can then measure precision $P = \frac{|changes \cap errors|}{|changes|}$, which corresponds to correct changes in the repair, recall $R = \frac{|changes \cap errors|}{|errors|}$, which corresponds to coverage of the errors, and F-measure $F = \frac{2\times(P\times R)}{(P+R)}$.

Results. We start with evaluating the discovery of approximate FDs. Table 2 shows the obtained precision and recall values. We rank the approximate FDs based on their scores and compute the precision and recall @k= {1,3,5}, where k is the number of dependencies evaluated after they are ordered with decreasing scores. On average, the precision and recall @k of the NPMI-sorted results is significantly higher than the CORDS-sorted results. The significance is most obvious @k = 1. On average, the NPMI scoring approach clearly yields better results than the baseline. This is not surprising, since CORDS was designed to discover approximate FDs on relational databases and it is reported to re-

Event	NMPI						CORDS					
	k=1		k=3		k=5		k=1		k=3		k=5	
	Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
Acquisition	1	0.25	0.66	0.5	0.8	1	1	0.25	0.66	0.5	0.6	0.75
Company Employees #	1	1	0.5	1	0.5	1	0	0	0.5	1	0.5	1
Company Meeting	1	0.14	1	0.42	1	0.7	1	0.14	0.66	0.28	0.6	0.42
Company Ticker	1	0.5	0.33	0.5	0.4	1	0	0	0	0	0.2	0.5
Credit Rating	1	0.16	1	0.5	1	0.83	1	0.16	1	0.5	1	0.83
Employment Change	0	0	0.66	0.18	0.6	0.27	0	0	0.33	0.09	0.4	0.18
Insider Transaction	1	0	1.0	0.01	1.0	0.02	0	0	0	0.66	0.01	0.02
Natural Disaster	1	1	0.5	1	0.5	1	1	1	0.5	1	0.5	1
Person Travel	0	0	0.33	0.5	0.4	1	0	0	0.33	0.5	0.4	1
Political Endorsement	1	1	0.5	1	0.5	1	0	0	0.5	1	0.5	1
Product Recall	0	0	0.66	0.17	0.8	0.33	0	0	0.66	0.17	0.8	0.33
Voting Result	1	0.5	1	1	1	1	1	0.5	1	1	1	1
Avg	0.75	0.38	0.68	0.57	0.71	0.76	0.42	0.17	0.57	0.50	0.62	0.67

Table 2: Precision/Recall of approximate FD discovery for sample events over 3 months of data.

quire a sample of size between 1k and 2k pairs [20]. Such amounts of data are not always available when the data is chunked into time buckets.

We analyze next how different duration values for the same rule impact the quality of the repairs. In Figure 8(a), the event is Acquisition. Since a company is usually acquired only once, the considered rule is a FD as it is demonstrated by the improvement in the quality of the repair for both precision and recall when the duration exceeds 3600 days (10 years). The explanation is that for smaller values, the rules cannot detect errors. As expected, the results in terms of detection (delete semantics) are much better than the ones that consider the modification of problematic values (update semantics). In particular, there is not enough redundancy in the data to find the correct update for the repair. In Figure 8(b), the event reports the number of employees for a company. As this information changes over time, different durations lead to different quality results in the repair. Intuitively, if the time is small, precision is favored over recall, and the other way around with large values. This is the behavior we observed for all events with temporal characterization. Also in this case, the detection has much better performance than the metric considering also the values of the updates. Finally, Figure 8(c) reports a case where there is a clear point in which precision falls to low values when the duration increases. In this case, the duration is too large and covers several changes of employment for a person, thus several correct values are detected as problematic.

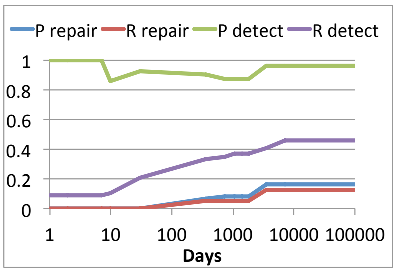
In Table 3, we report the discovered minimum duration M and the cleaning quality results with the Repair-Outliers (RO) and No Repair-Probability (NP) approaches. We compare them against (i) the results obtained using a Fd, (ii) the average of the durations suggested by three domain experts, (iii) the best duration value for the rule, selected with the previous study (as in Figure 8). The first three rules do not depend on time since they have only one correct reference value in our dataset. Hence, the Fds perform best for this case. The duration discovery algorithm was not able to find a duration that is large enough to make the Tfd perform better. This is because our dataset mainly contains events that happened within three months and the discovery approach subsequently suffers from the limited timespan when identifying these larger durations. Interestingly, values for the remaining events change over time. In these cases, the durations discovered with our RO approach always lead

Event	Entity	Conditional		Global	
		M	F	M	F
Company Emp #	Wal-Mart	24	**0.82**	24	**0.82**
Company Emp #	Tesco	27	**1.0**	24	**1.0**
Company Meet.	Val. Pharm. Int.	217	**0.68**	336	**0.68**
Company Meet.	Wal-Mart	45	**0.57**	336	**0.57**
Credit Rating	Tysons Foods	53	**1.0**	48	**1.0**
Credit Rating	NY Method. Hosp.	72	**0.66**	48	0.0
Emp. Change	Sean Moriarty	3168	**1.0**	24	**1.0**
Emp. Change	Rodney Reid	12	**1.0**	24	**1.0**
Natural Disaster	Argentina	45	**0.57**	24	**0.57**
Natural Disaster	England	7	**0.67**	24	0.6
Person Travel	C. Ronaldo	24	**0.71**	48	0.69
Person Travel	Lady Gaga	26	**0.73**	48	0.69
Pol. Endorsement	Ron Paul	96	0.52	48	**0.54**
Pol. Endorsement	Sarah Palin	20.5	**0.75**	48	**0.75**
Product Recall	vehicle	108	**0.76**	177	**0.76**
Product Recall	cars	32	**1.0**	177	0.89
Voting Result	Afghanistan	24	**0.76**	24	**0.76**
Voting Result	United States	6	0.33	24	**0.86**

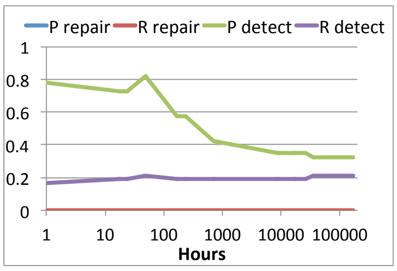
Table 4: Comparison of F-measure results for conditional and global TFDs.

to better a F-measure value than the Fds and the alternative approach NP. Moreover, in several cases we are able to achieve the same precision and recall of the best duration from the previous study. The other Tfd approaches NR and AL performed similarly to RO on these datasets with failures in some cases. For example AL discovers a duration of 0 for Voting Result, while NR fails with the events with higher noise rate, such as Company Employees #. We shall elaborate on the differences of the Tfd approaches in more detail in the next subsection. Finally, the average of the durations collected from the three domain experts show poor results in terms of F-measure, with the exception of the Person Travel case. This confirms that manually crafting the correct durations for data cleaning is a hard problem to be tackled top-down; a bottom-up approach that mines the data leads to more useful results.

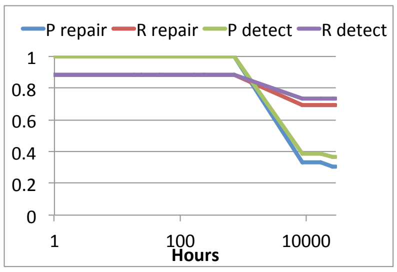
While the minimum durations from Table 3 can be applied for all the entities, we report in Table 4 minimum duration values for popular entities over all events. In our approach, the specific minimum duration for each entity can be computed before the aggregation of the stripes. For popular entities, these values can lead to better cleaning results. For example, while the discovered minimum duration for Person Travel is 48 hours (Table 3), conditional rules for pop-



(a) Acquisition (FD)



(b) Company Employee # (TFD)



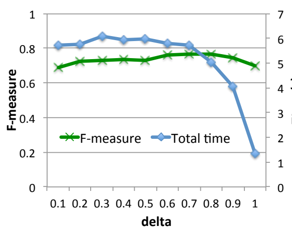
(c) Employment Change (TFD)

Figure 8: Cleaning results with different durations.

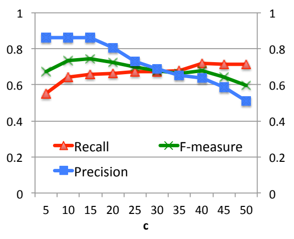
Event	TFD Semantic (RO)				TFD Semantic (NP)				FD semantic			Humans		Best possible F			
	M	*P*	*R*	*F*	*M*	*P*	*R*	*F*	*P*	*R*	*F*	*M*	*F*	*M*	*P*	*R*	*F*
Acquisition	48	1.0	0.08	0.16	840	0.92	0.21	0.34	0.96	0.46	**0.62**	-	-	-	0.96	0.46	0.62
Company ticker	24	0.43	0.25	0.31	1	0.69	0.14	0.23	0.96	1.0	**0.98**	-	-	-	0.96	1.0	0.98
Insider transaction	24	1.0	1.0	**1.0**	264	1.0	1.0	**1.0**	1.0	1.0	**1.0**	-	-	-	1.0	1.0	1.0
Company Employees #	24	0.74	0.17	**0.27**	1344	0.37	0.17	0.23	0.24	0.19	0.22	1016	0.23	48	0.73	0.20	0.31
Company Meet.	336	0.94	0.5	**0.65**	5k	0.4	0.54	0.46	0.38	0.53	0.44	4560	0.46	720	0.88	0.53	0.67
Credit Rating	48	0.6	0.75	**0.67**	72	0.56	0.75	0.64	0.18	0.66	0.29	4680	0.55	24	0.69	0.75	0.72
Employment Change	24	1.0	0.88	**0.94**	14k	0.39	0.73	0.51	0.37	0.8	0.51	12k	0.5	$\leq$720	1	0.88	0.94
Natural Disaster	24	0.8	0.5	0.62	29	0.8	0.5	0.62	0.51	0.91	0.65	255	**0.86**	[168:500]	0.93	0.78	0.86
Person Travel	48	0.61	0.82	0.7	72	0.59	0.84	0.69	0.42	0.93	0.58	36	**0.73**	24	0.92	0.85	0.88
Political Endorsement	48	1.0	0.59	**0.74**	216	0.85	0.65	0.73	0.52	0.88	0.65	1200	0.68	[24:70]	1.0	0.59	0.74
Product Recall	177	0.9	0.9	**0.9**	7,033	0.41	0.9	0.56	0.38	0.9	0.53	352	**0.9**	[100:400]	0.9	0.9	0.9
Voting Result	24	1.0	0.6	**0.75**	816	0.79	0.71	**0.75**	0.31	0.9	0.59	4440	0.57	720	0.83	0.75	0.79

Table 3: Precision and recall of the error detection based on duration discovery approaches (M in hours).

ular entities yield higher F-measure than the unconditional TFDs. Interestingly, there is a case where the conditional TFD performs worse that the non-conditional one. Since in the US there can be multiple elections in different states in the same day, the algorithm mines a very low duration of 6 hours. This suggests that Voting Result extractors can be revised to consider American states, instead of one country.



(a) Early termination δ



(b) Cut off point c with RO

Figure 9: Study of the input parameters averaged over 9 temporal events: (a) Execution time and F-measure for the top-5 rules, (b) Precision, Recall, and F-measure of the repair for RO**.**

Parameters. In the above experiments the early termination threshold δ and the cut-off point c for trimming and duration discovery were set to 0.7 and 10%, respectively. We report in Figure 9(a) how different values for δ affect the F-measure of the top-5 dependencies discovered with our method. We observe that aggressive pruning leads to faster execution, but to a loss in the quality of the results. However, an early termination with δ=0.7 reduces the execution from 52 to 6 seconds and preserves the quality.

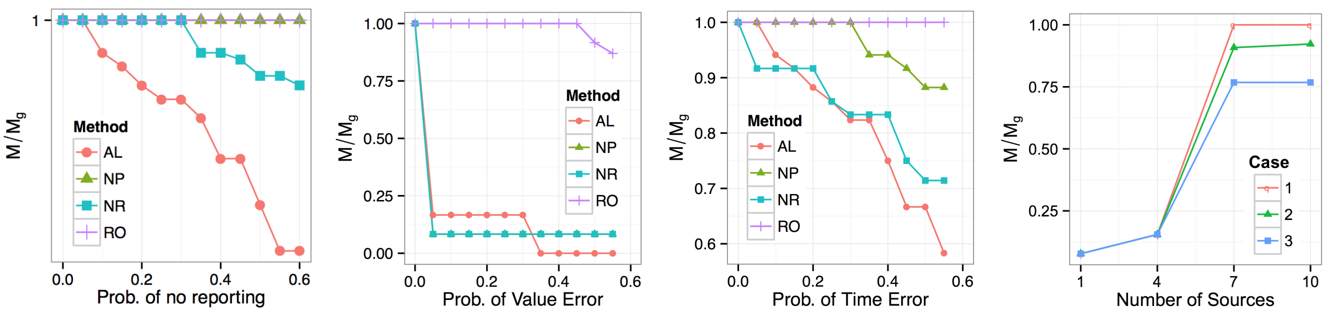
Figure 9(b) shows that the discovery algorithm behaves as expected with respect to the cut-off parameter c: low cut-off points lead to high precision and higher values lead to higher recall. This property allows the user to tune the discovery for their target application requirements. Interestingly, increasing values for c show similar behavior for both RO and NP, and the default value of $c = 10$ is close to the max F-measure value for both methods.

Execution times. AETAS's runtime is dominated by the time needed for reading the data from a database. For the largest dataset, CompanyTicker with more than one million tuples, and without early termination, AETAS took a total time of 53 seconds from which 52 seconds were spent to identify the implications and 1 second to discover the minimum duration for a chosen implication. With early termination, the process takes less than 2 seconds. The dependency cube is also easy to maintain in memory as we handle one cube per FD at a time and its size is bound by the number of tuples. Also, when imputing missing timestamps, we do not materialize their values in the stripe, and we implicitly maintain the time sequence for a non changing value.

5.2 Synthetic Data

The goal of the experiments with synthetic data is to analyze how the discovery algorithms perform wrt different properties of the data.

Dataset. In each scenario, we generate S sources with information over events for O objects for T timestamps. The generation of the values follows a TFD with a given M_g. Each tuple for a source has an attribute for the entity (reference value), an attribute value for the TFD, and a timestamp. For example, for TFD *name* $\wedge\, \Delta \rightarrow$ *position* with Δ=[0, 2] would generate tuples such as *(Jay, worker, 1)*, *(Jay, worker, 2)*, *(Jay, manager, 3)*, *(Jay, manager, 4)*,



(a) Reporting error P_r (b) Value error P_e (c) Time error P_t (d) Multiple sources

Figure 10: Ratio of the mined duration M wrt the golden duration M_g with different kinds of errors.

(Jay, manager, 5), (Jay, manager, 7), and (Jay, clerk, 8). For each timestamp and entity, a source has a probability P_r of not reporting the current value in a tuple, a probability P_h of changing the value (with the current duration up to a given maximum value), a probability P_e of reporting a wrong value, and a probability P_t of reporting a wrong timestamp. We run the different discovery algorithms on the union of the data from all sources. All experiments have been carried out for 1000 reference values and the probability of changing the value P_h was set to 0.2.

Metrics. In this controlled environment, we know the properties of our generative model, such as the golden values for M_g. We can thus measure the quality of the mining as the ratio of the discovered duration M to the input duration M_g. A ratio of 1 shows that the method has correctly mined M.

Results. To evaluate the influence of errors and sparseness in the data, we created different scenarios by varying different parameters. We first tested each type of error, namely missing values, wrong values, and wrong timestamps in isolation, i.e., we varied the probability of the error at hand from 0 to 0.6 and fixed the others probabilities to zero. We had 12 experiments for each error type by varying (i) the number of reporting sources from 2 to 5, and (ii) M_g to 7, 12, 17. The maximum duration (the time an event holds) was fixed at $M_g + 5$. From these experiments, we collected 12 M/M_g ratios, and took their median. We then tested the role of sources and skewed error rates for our method. In this experiment, we considered 1, 4, 7, and 10 sources, all of them with P_r=0.1 and P_t=0.2, and three cases. In case 1, the first source has P_e=0.1, and at each step we add three new sources with P_e values 0.1, 0.3, and 0.5. Similarly, in case 2 (resp. 3), the first source has P_e=0.2 (resp. 0.3), and at each step we add three sources with P_e values 0.2, 0.4, and 0.6 (0.3, 0.5, and 0.7 resp.).

Figure 10 shows the overall results. We see that RO performs better than the baselines with all error types. In Figure 10(a), it is easy to see that both RO and NP are robust to missing values, AL performs poorly because it cannot align stripes when values do not match, and the absence of integration leads to several missing values for NR. With errors in the values (Figure 10(b)) RO is the only one able to perform well with high percentages of noise, while the other methods experience a big drop in performance. In Figure 10(c)), we see that RO is robust to errors in the timestamps and computes better durations that the others (NP drops in performance at 0.3). Finally, Figure 10(d) shows that increasing the number of sources leads to improvement in the mining. The combination of missing values, errors in the timestamps, and very erroneous sources make the problem more challenging. In particular, a useful duration is discovered starting with seven sources in all cases.

6. RELATED WORK

Our work is related to two main areas, namely, dependency discovery and temporal data management.

In the context of constraints discovery, FDs attracted the most attention. TANE is a representative for the schema-driven approach to discovery [19], while FASTFD is an instance-driven approach [32]. Recently, DFD has also been proposed with improvements in performance [1]. While all these methods are proven to be valid over clean data, few solutions have been proposed for discovery over noisy data. An extension in this direction is the discovery of approximate FDs that hold for a subset of a relation with respect to a given threshold [23]. A similar extension has been proposed to mine approximate TFDs [11]. The major drawback of approximate FDs on noisy data is that from a certain threshold of noise on, such as 26% in our real world data scenario, the results of the discovery approach will mix up useful approximate FDs with actual non-dependent columns.

Another aspect of discovering constraints is to measure their importance according to a scoring function. CORDS [20], which we use as baseline for approximate FDs discovery, uses statistical correlations for each column pair to score possible FDs. In conditional functional dependencies (CFDs) discovery, other measures have been proposed, including support, which is defined as the percentage of the tuples in the data that match the pattern tableaux (the constants) and χ^2 test [8, 14]. Song et al. introduced the concept of differential dependencies [31] by extending FDs with differential functions, which are dependencies that change over time. They also mine dependencies, but they have focused on identifying dependencies on clean data only.

Integration and cleaning with temporal data [2, 9, 25, 27] is also of interest. The related approaches can benefit from our algorithms. The PRAWN integration system [2] can use our TFDs to detect errors, while the record-linkage systems for temporal data can exploit our repair-based duration discovery for their mining of temporal behavior. In fact, their goal is to identify records that describe the same entity over time and understanding how long a value should hold is critical for their algorithms. In particular, we adapted the disagreement decay discovery algorithm from [27] to our setting and indeed it can be applied for minimum duration discovery. From the experimental study, it is clear that our algorithm does better because of the improved robustness wrt the noise in the data. Notice that, differently from [27], noisy data cannot be clustered with good results. We therefore decided to go directly to the tuple-pair comparisons in the cleaning step, and this aggressive cleaning is supported by the experimental results with low execution times and good results in terms of quality. Another application for our rules is truth discovery [15, 28, 29, 33].

7. CONCLUSION

We presented AETAS, a system for the discovery of approximate temporal functional dependencies. At the core of the system are two modules that exploit machine learning techniques to identify approximate dependencies and their durations from noisy web data. As we have shown in the experimental study, traditional FDs lead to poor results when used on a temporal dataset in a data cleaning system. On the contrary, temporal dependencies can improve the quality of the data; our system is able to discover TFDs with minimal interactions with the users and with better results than alternative methods.

As a future direction, we plan to mine TFDs that identify *large* extreme values over the duration distributions, i.e., outlying durations that are too long for a certain event. For example, in many countries politicians have a maximum number of mandates for a certain position. We also plan to extend our duration discovery algorithm with more sophisticated methods for temporal outlier detection [18].

8. ACKNOWLEDGMENTS

This research was supported by Qatar Computing Research Institute (QCRI). The research in this paper used data kindly provided by Recorded Future.

9. REFERENCES

[1] Z. Abedjan, P. Schulze, and F. Naumann. DFD: efficient functional dependency discovery. In *CIKM*, pages 949–958, 2014.

[2] B. Alexe, M. Roth, and W.-C. Tan. Preference-aware integration of temporal data. *PVLDB*, 8(4):365–376, 2014.

[3] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1):197–207, 2010.

[4] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.

[5] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

[6] G. Bouma. Normalized (pointwise) mutual information in collocation extraction. In *GSCL*, pages 31–40, 2009.

[7] M. Bronzi, V. Crescenzi, P. Merialdo, and P. Papotti. Extraction and integration of partially overlapping web sources. *PVLDB*, 6(10):805–816, 2013.

[8] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1):1166–1177, 2008.

[9] Y.-H. Chiang, A. Doan, and J. F. Naughton. Modeling entity evolution for temporal record matching. In *SIGMOD*, pages 1175–1186, 2014.

[10] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[11] C. Combi, P. Parise, P. Sala, and G. Pozzi. Mining approximate temporal functional dependencies based on pure temporal grouping. In *ICDMW*, pages 258–265, 2013.

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

[13] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *PVLDB*, 7(10):881–892, 2014.

[14] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE TKDE*, 23(5):683–698, 2011.

[15] A. Galland, S. Abiteboul, A. Marian, and P. Senellart. Corroborating information from disagreeing views. In *WSDM*, pages 131–140, 2010.

[16] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.

[17] A. Gelman and J. Hill. *Data analysis using regression and multilevel/hierarchical models.* Cambridge U. Press, 2006.

[18] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. *Outlier Detection for Temporal Data.* Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan & Claypool Publishers, 2014.

[19] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[20] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[21] C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Extending existing dependency theory to temporal databases. *IEEE Trans. Knowl. Data Eng.*, 8(4):563–582, 1996.

[22] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. Bigdansing: A system for big data cleansing. In *SIGMOD*, pages 1215–1230, 2015.

[23] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. In *ICDT*, pages 129–149, 1995.

[24] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, pages 53–62, 2009.

[25] F. Li, M. Lee, W. Hsu, and W. Tan. Linking temporal records for profiling entities. In *SIGMOD*, pages 593–605, 2015.

[26] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–483, 2010.

[27] P. Li, X. L. Dong, A. Maurino, and D. Srivastava. Linking temporal records. *PVLDB*, 4(11):956–967, 2011.

[28] X. Li, X. L. Dong, K. Lyons, W. Meng, and D. Srivastava. Truth finding on the deep web: Is the problem solved? *PVLDB*, 6(2):97–108, 2012.

[29] J. Pasternack and D. Roth. Making better informed trust decisions with generalized fact-finding. In *IJCAI*, pages 2324–2329, 2011.

[30] S. Truvé. A white paper on temporal analytics. www.recordedfuture.com/assets/RF-White-Paper.pdf.

[31] S. Song, L. Chen, and H. Cheng. Efficient determination of distance thresholds for differential dependencies. *IEEE Trans. Knowl. Data Eng.*, 26(9):2179–2192, 2014.

[32] C. M. Wyss, C. Giannella, and E. L. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK*, pages 101–110, 2001.

[33] B. Zhao, B. I. Rubinstein, J. Gemmell, and J. Han. A bayesian approach to discovering truth from conflicting sources for data integration. *PVLDB*, 5(6):550–561, 2012.

Robust Discovery of Positive and Negative Rules in Knowledge-Bases

Stefano Ortona* Venkata Vamsikrishna Meduri† Paolo Papotti‡
* Meltwater – stefano.ortona@meltwater.com
† Arizona State University – vmeduri@asu.edu
‡ Eurecom – papotti@eurecom.fr

Abstract—**We present RUDIK, a system for the discovery of declarative rules over knowledge-bases (KBs). RUDIK discovers rules that express *positive* relationships between entities, such as "if two persons have the same parent, they are siblings", and *negative rules*, i.e., patterns that identify contradictions in the data, such as "if two persons are married, one cannot be the child of the other". While the former class infers new facts in the KB, the latter class is crucial for other tasks, such as detecting erroneous triples in data cleaning, or the creation of negative examples to bootstrap learning algorithms. The system is designed to: *(i)* enlarge the *expressive power* of the rule language to obtain complex rules and wide coverage of the facts in the KB, *(ii)* discover *approximate* rules (soft constraints) to be robust to errors and incompleteness in the KB, *(iii)* use disk-based algorithms, effectively enabling rule mining in commodity machines. In contrast with traditional ranking of all rules based on a measure of support, we propose an approach to identify the subset of useful rules to be exposed to the user. We model the mining process as an incremental graph exploration problem and prove that our search strategy has guarantees on the optimality of the results. We have conducted extensive experiments using real-world KBs to show that RUDIK outperforms previous proposals in terms of efficiency and that it discovers more effective rules for the application at hand.**

I. INTRODUCTION

Building large RDF knowledge-bases (KBs) is a popular trend in information extraction. KBs store information in the form of triples, where a *predicate*, expresses a binary relation between a *subject* and an *object*. KB triples, called facts, store information about real-world entities and their relationships, such as "Michelle Obama is married to Barack Obama". Significant effort has been put on KBs creation in the last 10 years in the research community (DBPedia [3], FreeBase [4], Wikidata [24], DeepDive [19], Yago [20]) as well as in the industry (e.g., Google [10], Wal-Mart [9]).

Unfortunately, due to their creation process, KBs are usually erroneous and incomplete. KBs are bootstrapped by extracting information from sources with minimal or no human intervention. This leads to two main problems. First, false facts are propagated from the sources to the KBs, or introduced by the extractors [10]. Second, usually KBs do not limit the information of interest with a schema and let users add facts defined on new predicates by simply inserting new triples. Since *closed world assumption* (CWA) does no longer hold in KBs [10], [13], we cannot assume that a missing fact is false, but we rather label it as *unknown* (*open world assumption*).

As a consequence, the amount of errors and incompleteness in KBs can be significant, with up to 30% errors for facts derived from the Web [1], [21]. Since KBs are large, e.g., WIKIDATA has more than 1B facts and 300M entities, checking all triples to find errors or to add new facts cannot be done manually. A natural approach to assist curators is to discover *declarative rules* that can be executed over the KB to improve the quality of the data [2], [5], [13]. We target the discovery of two types of rules: *(i) positive rules* to enrich the KB with new facts and thus increase its coverage, *(ii) negative rules* to spot logical inconsistencies and identify erroneous triples.

Example 1: Consider a KB with information about parent and child relationships. A positive rule is the following:

$$r_1 : \texttt{parent}(b, a) \Rightarrow \texttt{child}(a, b)$$

stating that if a person a is parent of person b, then b is child of a. A negative rule has similar form, but different semantics. For example (*DOB* stands for Date Of Birth),

$$r_2 : \texttt{DOB}(a, v_0) \wedge \texttt{DOB}(b, v_i) \wedge v_0 > v_i \wedge \texttt{child}(a, b) \Rightarrow \perp$$

states that person b cannot be child of a if a was born after b. By executing the rule as a query over `child` facts, we identify erroneous triples.

In order to be executed over a KB, or plugged into an existing inference system [17], rules must be manually crafted, a task that can be difficult for domain experts without a CS background. Also, the rule creation process is usually very expensive, as large KB can have rules in the thousands [22]. A rule discovery system is therefore a crucial asset to help the users in data curation. However, three main challenges arise when discovering positive and negative rules from KBs.

Data Quality. While traditional rule mining techniques assume that data is either clean or has a negligible amount of errors [6], KBs can present errors and are incomplete.

Open World Assumption. Other approaches rely on the presence of positive and negative examples [8], [16], but KBs contain only positive statements, and, without CWA, there is no immediate solution to derive counter examples.

Volume. Existing approaches for rule discovery assume that data fit into main memory [2], [13], [5], [12]. Given the large and increasing size of KBs, these approaches focus on a simple rule language to minimize the size of the search space.

We present RUDIK (Rule Discovery in Knowledge Bases), a novel system for the discovery of rules over KBs that addresses these challenges. RUDIK is the first system designed to discover both *positive and negative rules* over noisy and incomplete KBs. By relying on disk based algorithms, RUDIK can handle a larger search space and discover rules with a richer language that allows value comparisons. This increase in the *expressive power* enables a larger number of patterns to be expressed in the rules, and therefore a larger number of new facts and errors can be identified with high accuracy. These results are achieved by exploiting the following contributions.

1. Problem Definition. We formally define the problem of robust rule discovery over erroneous and incomplete KBs. The input of the problem are two sets of positive and negative examples for every predicate. In contrast to the traditional ranking of a large set of rules based on a measure of support [8], [13], [18], our problem definition aims at the identification of a subset of *approximate* rules, i.e., rules that do not necessarily hold over all the examples, since data errors and incompleteness are in the nature of KBs. The solution is then the smallest set of rules that cover the majority of input positive examples, and as few input negative examples as possible (Section III).

2. Example Generation. Positive and negative examples for a target predicate are crucial to our approach as they determine the ultimate quality of the rules. However, crafting a large number of negative examples is a tedious exercise that requires manual work. We present an algorithm for example generation that is aware of missing data and inconsistencies in the KB. Our generated examples lead to better rules than examples obtained with alternative approaches (Section IV).

3. Rule Discovery Algorithm. We give a $\texttt{log}(k)$-approximation algorithm for the rule discovery problem, where k is the maximum number of input positive examples covered by a single rule. We discover rules by judiciously using the memory. The algorithm incrementally materializes the KB as a graph, and discovers rules by navigating only the paths that potentially lead to the best rules. By materializing only the portion of the KB that is needed for the promising rules, the disk-access is minimized and the low memory footprint enables the mining with a richer rule language (Section V).

We experimentally test the performance of RuDiK on three popular and widely used KBs. We show that our system delivers accurate rules, with a relative increase in average precision by 45% both in the positive and in the negative settings w.r.t. state-of-the-art systems. Also, differently from other proposals, RuDiK performs consistently well with KBs of all sizes on a regular laptop. Finally, we demonstrate how discovered negative rules provide Machine Learning algorithms with training examples of quality comparable to examples manually crafted by humans (Section VI).

II. Preliminaries

We focus on discovering rules from RDF KBs. An RDF KB is a database that represents information through RDF triples $\langle s, p, o\rangle$, where a *subject* (s) is connected to an *object* (o) via a *predicate* (p). Triples are often called *facts*. For example, the fact that Scott Eastwood is the child of Clint Eastwood could be represented through the triple $\langle Clint\_Eastwood, child, Scott\_Eastwood\rangle$. RDF KB triples respect three constraints: (i) triple subjects are always *entities*, i.e., concepts from the real world; (ii) triple objects can be either entities or *literals*, i.e., primitive types such as numbers, dates, and strings; (iii) triple predicates specify real-world relationships between subjects and objects.

Differently from relational databases, KBs usually do not have a schema that defines allowed instances, and new predicates can be added by inserting triples. This model allows great flexibility, but the likelihood of introducing errors is higher than traditional schema-guided databases. While KBs can include *T-Box* facts to define classes, domain/co-domain types for predicates, and relationships among classes to check integrity, in most KBs – including the ones used in our experiments – such information is missing. Hence our focus is on the *A-Box* facts that describe instance data.

A. Language

Our goal is to automatically discover first-order logical formulas in KBs. More specifically, we target the discovery of *Horn Rules* with universally quantified variables only. A Horn Rule is a disjunction of *atoms* with at most one unnegated atom. In the implication form, they have the following format:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow B$$

where $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ is the *body* of the rule (a conjunction of atoms) and B is the *head* of the rule (a single atom). However, it is logically equivalent to rewrite the atom in the head of the rule in its negated form in the body to emphasize contradictions:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge \neg B \Rightarrow \bot$$

We therefore distinguish between *positive rules*, which generate new facts (e.g., r_1 in Example 1), and *negative rules* (e.g., r_2 in Example 1), which identify incorrect facts, similarly to denial constraints for relational data [6]. An atom is a predicate connecting two variables, two entities, or an entity and a variable. For simplicity, we write an atom with the notation $\texttt{rel}(a, b)$, where $\texttt{rel}$ is a KB predicate and a, b are either variables or entities. Given a rule r, we define r_{body} and r_{head} as the body and the head of the rule, respectively, and refer to the variables in the head of the rule as the *target variables*.

We remark that we also discover rules with a body atom in its negated form in the head. The result is a formula that generates negative facts. For example, negative rule r_2 is obtained by rewriting in the body the atom $\texttt{notChild}$ in the following rule:

$$r_2' : \texttt{DOB}(a, v_0) \wedge \texttt{DOB}(b, v_i) \wedge v_0 > v_i \Rightarrow \texttt{notChild}(a, b)$$

As shown in the negative rule, we allow *literal comparisons* in our rules. A literal comparison is a special atom $\texttt{rel}(a, b)$, where $\texttt{rel} \in \{<, \leqslant, \neq, >, \geqslant\}$, and a and b can only be assigned to literal values except if $\texttt{rel}$ is equal to $\neq$, i.e., we allow inequality comparisons for entities.

Given a KB kb and an atom $A = \texttt{rel}(a, b)$ where a and b are two entities, we say that A *holds* over kb iff $\langle a, \texttt{rel}, b\rangle \in kb$. Given an atom $A = \texttt{rel}(a, b)$ with at least one variable, we say that A can be *instantiated* over kb if there exists at least one entity from kb for each variable in A s.t. if we substitute all variables in A with these entities, A holds over kb. Transitively, we say that r_{body} can be instantiated over kb if every atom (with entities) in r_{body} can be instantiated and every literal comparison is logically true.

As in other approaches [13], [5], we want to avoid Cartesian products in our rules and therefore define a rule *valid* iff every variable in it appears at least twice. Target variables already appear once in the head of the rule, but each non target variable must be involved in a join or in a comparison.

B. Rule Coverage

Given a pair of entities (x, y) from a KB kb and a Horn Rule r, we say that r_{body} *covers* (x, y) if $(x, y) \models r_{body}$. In other words, given a rule $r : r_{body} \Rightarrow \texttt{r}\,(a, b)$, r_{body} covers a pair of entities $(x, y) \in kb$ iff we can substitute a with x, b with y, and the rest of the body can be instantiated over kb. Given a set of pair of entities $E = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ and a rule r, we denote by $C_r(E)$ the *coverage* of r_{body} over E as the set of elements in E covered by r_{body}: $C_r(E) = \{(x, y) \in E | (x, y) \models r_{body}\}$.

Given the body r_{body} of a rule r, we denote by r^*_{body} the *unbounded body* of r. The unbounded body of a rule is obtained by keeping only atoms that contain a target variable and substituting such atoms with new atoms where the target variable is paired with a new unique variable. As an example, given $r_{body} = \texttt{rel}_1\,(a, v_0) \wedge \texttt{rel}_2\,(v_0, b)$ where a and b are the target variables, $r^*_{body} = \texttt{rel}_1\,(a, v_i) \wedge \texttt{rel}_2\,(v_{ii}, b)$. While in r_{body} the target variables are bounded to be connected by variable v_0, in r^*_{body} they are unbounded. Given a set of pair of entities $E = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ and a rule r, we denote by $U_r(E)$ the *unbounded coverage* of r^*_{body} over E as the set of elements in E covered by r^*_{body}: $U_r(E) = \{(x, y) \in E | (x, y) \models r^*_{body}\}$. Note that, given a set E, $C_r(E) \subseteq U_r(E)$.

Example 2: We denote with E the set of all possible pairs of entities in kb. The coverage of r_2 of Example 1 over E ($C_r(E)$) is the set of all pairs of entities $(x, y) \in kb$ s.t. both x and y have the DOB information and x is born after y. The unbounded coverage of r over E ($U_r(E)$) is the set of all pairs of entities (x, y) s.t. both x and y have the DOB information, no matter what the relation between the two birth dates is.

The unbounded coverage is essential to distinguish between missing and inconsistent information: if for a pair of entities (x, y) the DOB is missing for either x or y, we cannot say whether x was born before or after y. But if both x and y have the DOB and x is born before y, we can state that r_2 does not cover (x, y). As KBs are incomplete, we must discriminate between missing and conflicting information. We extend the definition of coverage and unbounded coverage to a set of rules $R = \{r_1, r_2, \cdots, r_n\}$ as the union of individual coverages:

$$C_R(E) = \bigcup_{r \in R} C_r(E) \qquad U_R(E) = \bigcup_{r \in R} U_r(E)$$

III. Rule Discovery for Noisy KBs

For the sake of simplicity, we define the discovery problem for a single *target predicate* given as input. To obtain all rules for a given KB, we compute rules for every predicate in it. We characterize a predicate with two sets of pairs of entities. The *generation set* G contains examples for the target predicate, while the *validation set* V contains counter examples for the same. Consider the discovery of positive rules for the child predicate; G contains true pairs of parents and children and V contains pairs of people who *are not* in a child relation. If we want to identify errors (negative rules), the sets of examples are the same, but they switch role. To discover negative rules for child, G contains pairs of people not in a child relation and V contains pairs of entities respecting the child relation.

We formalize next the *exact discovery problem.* In the following definitions, we assume for the sake of simplicity that all possible valid rules and the sets of examples have been already generated, we detail in the rest of the paper how they are efficiently obtained from the KB.

Definition 1: Given a KB kb, two sets of pairs of entities G and V from kb with $G \cap V = \emptyset$, and all the valid Horn Rules R for kb, a solution for the *exact discovery problem* is a subset R' of R s.t.:

$$\underset{R'}{\operatorname{argmin}}(size(R') | (C_{R'}(G) = G) \wedge (C_{R'}(V) \cap V = \emptyset))$$

The exact solution is the minimal set of rules that covers all pairs in G and none of the pairs in V. It minimizes the number of rules in the output ($size(R')$) to avoid overfitting rules covering only one pair, as such rules have no impact when applied on the KB. In fact, given a pair of entities (x, y), there is always an overfitting rule whose body covers only (x, y) by assigning target variables to x and y as shown next.

Example 3: Consider the discovery of positive rules for the predicate couple between two persons using as example the Obama family. A positive example is (Michelle, Barack) and a negative example is their daughters (Malia, Natasha). Given three rules:

$$r_3 : \texttt{livesIn}\,(a, v_0) \wedge \texttt{livesIn}\,(b, v_0) \Rightarrow \texttt{couple}\,(a, b)$$

$$r_4 : \texttt{hasChild}\,(a, v_i) \wedge \texttt{hasChild}\,(b, v_i) \Rightarrow \texttt{couple}\,(a, b)$$

$$r_5 : \texttt{hasChild}\,(Michelle, Malia) \wedge \texttt{hasChild}\,(Barack, Malia) \Rightarrow \texttt{couple}\,(Michelle, Barack)$$

Rule r_3 states that two persons are a couple if they live in the same place, while rule r_4 states that they are a couple if they have a child in common. Assuming the information livesIn(x,y) and hasChild(x,y) are in the KB, both rules r_3 and r_4 cover the positive example. Rule r_4 is an exact solution, as it does not cover the negative example, while this is not true for r_3, as also the daughters live in the same place. Rule r_5 explicitly mentions entity values (constants) in its head and body. It is also an exact solution, but it applies only for the given positive example.

If any of the hasChild relationships between the parents and the daughters is missing in G, the exact discovery would find only r_5 as a solution. This highlights that the exact discovery is not robust to data problems in KBs. Even if a valid rule exists semantically, missing triples or errors for the examples in G and V can lead to faulty coverage. In the worst case, every rule in the exact solution would cover only one example in G, i.e., a set of overfitting rules with no effect when applied on the KB.

A. Weight Function

Given errors and missing information in both G and V, we drop the requirement of exactly covering the sets with the rules. In other words, we mine rules that hold for most of the data (soft-constraints), as we want to be robust w.r.t. noise and incompleteness. However, coverage is a strong indicator of quality: good rules should cover several examples in G, while covering elements in V can be an indication of incorrect rules. We model this idea in a *weight* associated with every rule.

Definition 2: Given a KB kb, two sets of pair of entities G and V from kb with $G \cap V = \varnothing$, and a Horn Rule r, the *weight of* r is defined as follow:

$$w(r) = \alpha \cdot (1 - \frac{\mid C_r(G) \mid}{\mid G \mid}) + \beta \cdot (\frac{\mid C_r(V) \mid}{\mid U_r(V) \mid}) \qquad (1)$$

with $\alpha, \beta \in [0,1]$ and $\alpha + \beta = 1$, thus $w(r) \in [0,1]$.

The weight captures the quality of a rule w.r.t. G and V: the better the rule, the lower the weight – a perfect rule covering all generation elements of G and none of the validation elements in V has a weight of 0. The weight is made of two components normalized by parameters α and β. The first component captures the coverage over the generation set G – the ratio between the coverage of r over G and G itself. The second component quantifies the coverage of r over V. The coverage over V is divided by the unbounded coverage of r over V, instead of the total elements in V, because some elements in V might not have the predicates stated in r_{body}. Intuitively, we restrict V with unbounded coverage to validate on "qualifying" examples.

Parameters α and β give relevance to each component. A high β steers the discovery towards rules with high precision by penalizing the ones that cover negative examples, while a high α champions the recall by favoring rules covering more generation examples.

Example 4: Consider again rule r_2 of Example 1 and two sets of pairs of entities G and V from a KB kb. The first component of w_r is computed as 1 minus the number of pairs (x, y) in G where x is born after y divided by the number of elements in G. The second component is the ratio between number of pairs (x, y) in V where x is born after y and number of pairs (x, y) in V where the birth date for both x and y is known in kb, i.e., examples with missing birth dates are not in $U_{r_2}(V)$.

Definition 3: Given a set of rules R, the *weight for* R is:

$$w(R) = \alpha \cdot (1 - \frac{\mid C_R(G) \mid}{\mid G \mid}) + \beta \cdot (\frac{\mid C_R(V) \mid}{\mid U_R(V) \mid})$$

Weights enable the modeling of the presence of errors in KBs. Consider the case of negative rule discovery, where V contains positive examples from the KB. We report in the experimental evaluation several negative rules with significant coverage over V, which corresponds to errors in the KB. The weight is important also for plugging rules into existing inference systems for KBs. For example, weighted rules can be interpreted as soft constraints for probabilistic reasoning [17].

B. Problem Definition

We can now state the approximate version of the problem.

Definition 4: Given a KB kb, two sets of pair of entities G and V from kb with $G \cap V = \varnothing$, all the valid Horn Rules R for kb, and a w weight function for R, a solution for the *robust discovery problem* is a subset R' of R such that:

$$\underset{R'}{\operatorname{argmin}}(w(R') | C_{R'}(G) = G)$$

The *robust* version of the discovery problem aims to identify rules that cover all elements in G and as few as possible elements in V. Since we do not want overfitting rules, we do not generate in R rules having constants in both target variables, thus avoiding any rule that covers only one example.

We can map this problem to the *weighted set cover problem*, which is proven to be NP–complete [7]. The reduction follows immediately from the following mapping: the set of elements (universe) corresponds to the generation examples in G, the input sets are identified by the rules defined in R (where each rule covers a subset of G), the non-negative weight function $w : r \to \mathbb{R}$ is $w(r)$ in Definition 2, and the cost of R is defined to be its total weight, according to Definition 3.

IV. Rule and Example Generation

In this Section we describe how to generate the universe of all possible rules. We start by assuming that the positive and the negative examples are given, and then show how they can be computed. However, our approach is independent of how G and V are generated: they could be manually crafted by domain experts, with significant additional manual effort.

We detail the discovery of positive rules having true facts in G and false facts in V. In the dual problem of negative rule discovery, our approach remains unchanged, we just switch the roles of G and V. The generation set G is formed out of false facts, while the validation set V is built from true facts.

A. Rule Generation

In the universe of all possible rules R, each rule must cover one or more examples from the generation set G. Thus the universe of all possible rules is generated by inspecting the elements of G alone. We translate a KB kb into a directed graph: entities and literals are the nodes, and there is a directed edge from node a to node b for each triple $\langle a, rel, b \rangle \in kb$. Edges are labelled with the relation rel that connects subject to object. Figure 1 shows four triples.

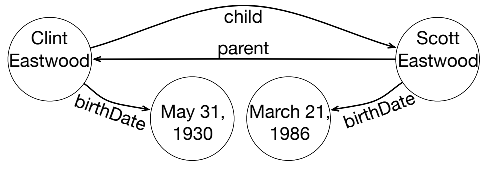


Fig. 1. Graph example for four triples from DBpedia.

The body of a rule can be seen as a path in the graph. In Figure 1, the body `child`(a, b) $\wedge$`parent`(b, a) corresponds to the path *Clint Eastwood* $\rightarrow$ *Scott Eastwood* $\rightarrow$ *Clint Eastwood*. As defined in Section II-A, a valid body contains target variables a and b at least once, every other variable at least twice, and atoms are transitively connected. If we allow navigation of edges independently of the edge direction, we can translate bodies of valid rules to valid paths on the graph. Given a pair of entities (x, y), a *valid body* corresponds to a valid path p on the graph such that: (i) p starts at the node x; (ii) p covers y at least once; (iii) p ends in x, in y, or in a different node that has been already visited. Given the body of a rule r_{body}, r_{body} covers a pair of entities (x, y) iff there exists a valid path on the graph that corresponds to r_{body}. This implies that for a pair of entities (x, y), we can generate bodies of all possible valid rules by computing all valid paths starting at x with a standard BFS. The key point is the ability to navigate each edge in any direction by turning the original directed graph into an undirected one. However, we need to keep track of the original direction of the edges. This is essential when translating paths to rule bodies. In fact, an edge directed from a to b produces the atom `rel`(a,b), while b to a produces `rel`(b,a).

Since every node can be traversed multiple times, for two entities x and y there might exist infinite valid paths starting from x. This is avoided with a $maxPathLen$ parameter that determines the maximum number of edges in the path, i.e., the maximum number of atoms allowed in the corresponding body of the rule. We show the impact of this parameter in Section VI.

We now describe the two main steps in our generation of the universe of all possible rules for G.

1. Create Paths. Given a pair of entities (x,y), we retrieve from the KB all nodes at a distance smaller than $maxPathLen$ from x or y, along with their edges. The retrieval is done recursively: we maintain a queue of entities, and for each entity in the queue we execute a SPARQL query against the KB to get all entities (and edges) at distance 1 from the current entity – we call these queries *single hop queries*. At the n-th step, we add the new found entities to the queue iff they are at a distance less than $(maxPathLen - n)$ from x or y and they have not been visited before. The queue is initialized with x and y. Given the graph for every (x,y), we then compute all valid paths starting from every x.

2. Evaluate Paths. Computing paths for every example in G implies also computing the coverage over G for each rule. The *coverage* of a rule r is the number of elements in G for which there exists a path corresponding to r_{body}. Once the universe of all possible rules has been generated (along with coverages over G), computing coverage and unbounded coverage over V requires only the execution of two SPARQL queries against the KB for each rule in the universe.

Since one of our goals is to increase the expressive power of discovered rules, we generate different atom types:

Literal comparison. We want predicate atoms with comparisons beyond equalities. To discover such atoms, the graph representation must contain edges that connect literals with one (or more) symbol from $\{<, \leqslant, \neq, >, \geqslant\}$. As an example, Figure 1 would contain an edge '$<$' from node "*March 31, 1930*" to node "*March 21, 1986*". Unfortunately, the original KB does not contain this kind of information explicitly, and materializing such edges among all literals is infeasible.

However, in our algorithm we discover paths for a pair of entities from G in isolation. The size of the graph resulting for a pair of entities is orders of magnitude smaller than the KB, thus we can afford to compare all literal pairwise comparisons within a single example graph. Besides equality comparisons, we add '$>$','$\geqslant$','$<$','$\leqslant$' relationships between numbers and dates, and $\neq$ between all literals. These new relationships are treated as normal atoms (edges): $x \geqslant y$ is equivalent to `rel`(x,y), where `rel` is equal to $\geqslant$.

Not equal variables. The "not equal" operator introduced for literals is useful for entities as well. Consider the rule:

$$\texttt{bornIn}(a,x) \wedge x \neq b \wedge \texttt{president}(a,b) \Rightarrow \bot$$

It states that if a person a is born in a country that is different from b, then a cannot be the president of b. One way to consider inequalities among entities is to add edges among all pairs of entities in the graph. However, this strategy is inefficient and would lead to many meaningless rules. To limit the search space while aiming at meaningful rules, we use the `rdf:type` triples associated to entities. We add an inequality edge in the input example graph only between pairs of entities of the same type (as in the example above).

Constants. Finally, we allow the discovery of rules with constant selections. Suppose that for the above president rule, all examples in G are people born in "*U.S.A.*", and there is at least one country for which this rule is not valid. According to our problem statement, the right rule is therefore:

$$\texttt{bornIn}(a,x) \wedge x \neq \textit{U.S.A.} \Rightarrow \neg\texttt{president}(a,\textit{U.S.A.})$$

To discover such atoms, we promote a variable v in a given rule r to an entity e iff for every $(x,y) \in G$ covered by r, v can always be instantiated with the same value e.

B. Input Example Generation

Given a KB kb and a predicate $rel \in kb$, we automatically build a generation set G and a validation set V as follows. G consists of positive examples for the target predicate rel, i.e., all pairs of entities (x,y) such that $\langle x, rel, y\rangle \in kb$. V consists of counter (negative) examples for the target predicate. These are more complicated to generate because of the open world assumption in KBs. Differently from classic databases, we cannot assume that what is not stated in a KB is false (closed world assumption), thus everything that is not stated is *unknown*. However, since the likelihood of two randomly selected entities being a positive example is extremely low, one simple way of creating false facts is to randomly select pairs from the Cartesian product of the entities [16]. While this process gives negative examples with a very high precision, only a very small fraction of these entity pairs are *semantically related*. This semantic aspect has effects in the applications that use the generated negative examples. In fact, unrelated entities have less meaningful paths than semantically related entities and this is reflected in lower quality in the experimental results.

A semantic connection is guaranteed for positive examples by definition, since pairs in G are always connected at least by the target predicate. To generate negative examples that are likely to be correct (true false facts) and that are semantically related, we mine the facts to identify the entities that are more likely to be complete, i.e., entities for which the KB contains full information. This process is done exploiting and extending the popular notion of *Local-Closed World Assumption* (*LCWA*) [13]. LCWA states that if a KB contains one or more object values for a given subject and predicate, then it contains all possible values. For example, if a KB contains one or more children of Clint Eastwood, then it contains all his children. This is always true for *functional* predicates (e.g., `capital`), while it might not hold for non-functional ones (e.g., `child`).

We generate negative examples taking the union of entities satisfying the LCWA. For a predicate rel, a negative example is a pair (x,y) where either x is the subject of one or more triples $\langle x, rel, y' \rangle$ with $y \neq y'$, or y is the object of one or more triples $\langle x', rel, y \rangle$ with $x \neq x'$. For example, if $rel = \texttt{child}$, a negative example is a pair (x, y) s.t. x has some children in the KB who are not y, or y is the child of someone who is not x. The LCWA guarantees that, since at least another child exists for x, (x, y) cannot be in such relation and we can safely use the pair as a counter-example. In addition, to obtain examples that are semantically related, it is enough to add the constraint that every example is made from a pair of entities that are connected via a predicate different from the target predicate. In other words, given a KB kb and a target predicate rel, (x, y) is a negative example if $\langle x, rel', y \rangle \in kb$, with $rel' \neq rel$.

Example 5: A negative example (x, y) for the target predicate `child` has the following characteristics: *(i)* x and y are not connected by a `child` predicate; *(ii)* either x has one or more children (different from y) or y has one or more parents (different from x); *(iii)* x and y are connected by a predicate that is different from `child` (e.g., `colleague`).

To enhance the quality of the input examples and avoid cases of mixed types, we require that for every example pair (x, y), either in G or V, all the x occurrences have the same *type*, same for the y values.

V. Discovery Algorithm

We introduce a greedy approach to solve the approximate discovery problem (Section III-B). Since the number of possible rules can be very large, we introduce an algorithm that generates only promising rules from the KB, while preserving the same quality guaranteed by the exhaustive generation.

A. Marginal Weight for a Greedy Algorithm

Our goal is to discover a set of rules to produce a weighted set cover for the given examples. We therefore follow the intuition behind the greedy algorithm for weighted set cover by defining a *marginal weight* for rules that are not yet included in the solution [7].

Definition 5: Given a set of rules R and a rule r such that $r \notin R$, the marginal weight of r w.r.t. R is defined as:

$$w_m(r) = w(R \cup \{r\}) - w(R)$$

The marginal weight quantifies the weight increase by adding r to an existing set of rules. Since the problem aims at minimizing the total weight, we never add a rule to the solution if its marginal weight is greater than or equal to 0.

If all rules have been generated, the algorithm for greedy rule selection is quite straightforward: given a generation set G, a validation set V, and the universe of all possible rules R, pick at each iteration the rule r with minimum marginal weight and add it to the solution R'. The algorithm stops when one of the following termination conditions is met: *1)* R is empty – all the rules have been included in the solution; *2)* R' covers all elements of G; *3)* the minimum marginal weight is greater than or equal to 0, i.e., among the remaining rules in R, none of them has a negative marginal weight.

The greedy solution guarantees a `log(`k`)` approximation to the optimal solution [7], where k is the largest number of elements covered in G by a rule in R. If the optimal solution is made of rules that cover disjoint sets over G, then the greedy solution coincides with the optimal one.

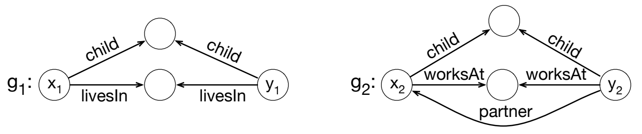


Fig. 2. Two positive examples.

B. A^* Graph Traversal

The greedy algorithm for weighted set cover assumes that the universe of rules R has been generated. To generate R, we need to traverse all valid paths from a node x to a node y, for every pair $(x, y) \in G$. But do we need all possible paths for every example?

Example 6: Consider the mining of positive rules for the target predicate `spouse`. The generation set G includes two examples g_1 and g_2 shown as graphs in Figure 2. Assume for simplicity that all rules in the universe have the same coverage and unbounded coverage over the validation set V. One candidate rule is $r : \texttt{child}(x, v_0) \wedge \texttt{child}(y, v_0) \Rightarrow \texttt{spouse}(x, y)$, stating that entities x and y with a common child are married. In the graph, r covers both g_1 and g_2. Since all rules have the same coverage and unbounded coverage over V, there is no need to generate any other rule. In fact, any other candidate rule will not cover new elements in G, therefore their marginal weights will be greater than or equal to 0. Thus the creation and navigation of edges `livesIn` in g_1, `worksAt` in g_2, and `partner` in g_2 is not needed.

Based on the above observation, we avoid the generation of the entire universe R, but rather consider at each iteration the most promising path on the graph as in the A^* graph traversal algorithm [14]. For each example $(x, y) \in G$, we start the navigation from x. We keep a queue of not valid rules, and at each iteration we consider the rule with the minimum marginal weight, which corresponds to paths in the example graphs. We expand the rule by following the edges, and we add the new founded rules to the queue of not valid rules. Unlike A^*, we do not stop when a rule (path) reaches the node y (i.e., becomes valid). Whenever a rule becomes valid, we add the rule to the solution and we do not expand it any further. The algorithm keeps looking for plausible paths until one of the termination conditions of the greedy cover algorithm is met.

A crucial point in A^* is the definition of the estimation cost. To guarantee the solution to be optimal, the estimation must be *admissible* [14], i.e., the estimated cost must be less than or equal to the actual cost. In our setting, given a rule that is not yet valid and needs to be expanded, we define an admissible estimation of the marginal weight.

Definition 6: Given a rule $r : A_1 \wedge A_2 \cdots A_n \Rightarrow B$, we say that a rule r' is an *expansion* of r iff r' has the form $A_1 \wedge A_2 \cdots A_n \wedge A_{n+1} \Rightarrow B$.

In the graph traversal, expanding r means traversing one further edge on the path defined by r_{body}. To guarantee the optimality condition, the estimated marginal weight for a rule r that is not valid must be less than or equal to the actual

Algorithm 1: RuDiK Rule Discovery.

```
input : $G$ – generation set
input : $V$ – validation set
input : $maxPathLen$ – maximum rule body length
output: $R_{opt}$ – union of rules in the solution
1  $R_{opt} \leftarrow \varnothing$;
2  $N_f \leftarrow \{x|(x,y) \in G\}$;
3  $Q_r \leftarrow$ expandFrontiers($N_f$);
4  $r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w^*_m(r))$;
5  repeat
6      $Q_r \leftarrow Q_r \backslash \{r\}$;
7      if isValid($r$) then
8          $R_{opt} \leftarrow R_{opt} \cup \{r\}$;
9      else
           // rules expansion
10         if length($r_{body}$) $< maxPathLen$ then
11             $N_f \leftarrow$ frontiers($r$);
12             $Q_r \leftarrow Q_r \cup$ expandFrontiers($N_f$);
13     $r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w^*_m(r))$;
14 until $Q_r = \varnothing \vee C_{R_{opt}}(G) = G \vee w^*_m(r) \geqslant 0$;
15 return $R_{opt}$
```

weight of any valid rule that is generated by expanding r. Given a rule and some expansions of it, we can derive the following.

Lemma 1: *Given a rule r and a set of pair of entities E, then for each r' expansion of r, $C_{r'}(E) \subseteq C_r(E)$ and $U_{r'}(E) \subseteq U_r(E)$.*

The above Lemma states that the coverage and unbounded coverage of an expansion r' of r are contained in the coverage and unbounded coverage of r, respectively, and directly derives from the augmentation inference rule for functional dependencies. The only positive contribution to marginal weights is given by $|C_{R\cup\{r\}}(V)|$. $|C_{R\cup\{r\}}(V)|$ is equivalent to $|C_R(V)| + |C_r(V) \backslash C_R(V)|$, thus if we set $|C_r(V) \backslash C_R(V)| = 0$ for any r that is not valid, we guarantee an admissible estimation of the marginal weight. We estimate the coverage over the validation set to be 0 for any rule that can be further expanded, since expanding it may bring the coverage to 0.

Definition 7: Given a *not valid* rule r and a set of rules R, we define the *estimated marginal weight* of r as:

$$w^*_m(r) = -\alpha \cdot \frac{|C_r(G) \backslash C_R(G)|}{|G|} + \beta \cdot \left(\frac{|C_R(V)|}{|U_{R\cup\{r\}}(V)|} - \frac{|C_R(V)|}{|U_R(V)|}\right)$$

The estimated marginal weight for a valid rule is equal to the actual marginal weight (Definition 5). Valid rules are not considered for expansion, therefore we do not need to estimate their weights since we know the actual ones. Given Lemma 1, we can see that $w^*_m(r) \leqslant w^*_m(r')$, for any r' expansion of r. Thus our marginal weight estimation is admissible.

We are ready to introduce Algorithm 1, which shows the modified set cover procedure, including the A^*-like rule generation. For a rule r, we call *frontier nodes*, $N_f(r)$, the last visited nodes in the paths that correspond to r_{body} from every example graph covered by r. Expanding a rule r implies navigating a single edge from any frontier node. In the algorithm, the set of frontier nodes is initialized with starting nodes x, for every $(x,y) \in G$ (Line 2). The algorithm maintains a queue of rules Q_r, from which it chooses at each iteration the rule with minimum estimated weight. The function expandFrontiers retrieves all nodes (along with edges) at distance 1 from frontier nodes and returns the set of all rules generated by this one hop expansion. Q_r is therefore initialized with all rules of length 1 starting at x (Line 3). In the main loop, the algorithm checks if the current best rule r is valid or not. If r is valid, it is added to the output and it is not expanded (Line 8). If r is not valid, it is expanded iff the length of its body is less than $maxPathLen$ (Line 10). The termination conditions and the last part of the algorithm are the same of the greedy set-cover algorithm, except that the output may not cover all input examples in G.

To analyze the complexity of Algorithm 1, we assume that each query has a constant cost (linear scan over an index). Each iteration in Algorithm 1 corresponds to the discovery of a rule (valid or invalid), and for each rule we count how many examples from G such a rule covers. The total number of iterations is at most the total number of rules. The worst case is a complete graph where for each predicate p in the KB and for each pair of nodes (x,y), there exists a labelled edge with p that connects x with y. In this case, the number of distinct paths of length $L \leqslant maxPathLen$ between any two nodes of G is $|P|^L$, where $|P|$ is the number of predicates in the KB. The asymptotic complexity of Algorithm 1 is therefore $O(|G| * |P|^L)$, where G is the generation set, and P is the set of predicates in the KB. In reality, most pairs in KBs are connected by very few predicates (1 to 2), thus $|P|$ is small. This is reflected by low execution times for the algorithm in the experiments.

The simultaneous rule generation and selection of Algorithm 1 brings multiple benefits. First, we do not generate the entire graph for every example in G. Nodes and edges are generated *on demand*, whenever the algorithm requires their navigation (Line 12). Rather than materializing the entire graph and then traversing it, our solution gradually materializes parts of the graph whenever they are needed for navigation (Lines 3 and 12). Second, the weight estimation prunes unpromising rules. If a rule does not cover new elements in G and does not unbounded cover new elements in V, then it is pruned.

VI. Experiments

We implemented the above techniques in RuDiK, our system for Rule Discovery in Knowledge Bases (https://github.com/stefano-ortona/rudik). We carried out an experimental evaluation of our approach and grouped the results in four categories: *(i)* demonstrating the quality of our output for positive and negative rules; *(ii)* comparing our method with the state-of-the-art systems; *(iii)* showing the applicability of rule discovery to create representative training data to learning algorithms; *(iv)* testing the role of the parameters in the system.

Settings. Experiments were run on a desktop with a quad-core i5 CPU at 2.80GHz and 16GB RAM. We used OpenLink Virtuoso, optimized for 8GB RAM, with its SPARQL query endpoint on the same machine. Weight parameters were set to $\alpha = 0.3$ and $\beta = 0.7$ for positive rules, and to $\alpha = 0.4$ and $\beta = 0.6$ for negative rules. We set the maximum number of atoms admissible in the body of a rule ($maxPathLen$) to 3. We discuss the role of these parameters in Section VI-D.

TABLE I. DATASET CHARACTERISTICS.

KB	*Version*	*Size*	*#Triples*	*#Predicates*
DBPEDIA	3.7	10.06GB	68,364,605	1,424
YAGO 3	3.0.2	7.82GB	88,360,244	74
WIKIDATA	20160229	12.32GB	272,129,814	4,108

Evaluation Metrics. We evaluated the effectiveness in discovering both positive and negative rules. For every KB, we first ordered predicates according to descending popularity (i.e., number of triples having that predicate). We then picked the top 3 predicates for which we knew there existed at least one meaningful rule, and other 2 top predicates for which we did not know whether meaningful rules existed or not.

The evaluation of the discovered rules has been done according to the best practice for rule evaluation [13]. If a rule was clearly semantically correct, we marked all its results over triples as true. If a rule correctness was unknown, we randomly sampled 30 triples either among the new facts (for positive rules) or among the errors (for negative rules), and manually checked them. The *precision* of a rule is then computed as the ratio of correct assertions out of all assertions. While we manually annotated only popular predicates, we executed RUDIK on all predicates in DBPEDIA and verified that results are consistent even with non popular predicates. Source code and test results, including annotated examples and discovered rules, are available online at https://github.com/stefano-ortona/rudik.

A. *Quality of Rule Discovery in* RUDIK

The first experiment evaluated the accuracy of discovered rules over three KBs: DBPEDIA, YAGO, and WIKIDATA. Table I shows their characteristics. Over the three KBs, the selected predicates cover 0.2% to 0.4% of the total triples, 0.2% to 8% of the total predicates, 3% to 7% of the total entities, with 8% to 14% entity overlap among the predicates.

Size is important, as loading a KB entirely in memory requires to either use large amount of memory [5], [12], or to shrink it by eliminating the literals [13]. Given the small memory footprint of our algorithm, we can mine rules with commodity HW resources and retain the literals, which are crucial for obtaining expressive rules. While RUDIK takes as input a target predicate at a time, it can discover rules over the entire KB by applying the same procedure on every predicate in it. We discuss next results for subsets of predicates because the manual annotation of the identified new facts and errors is a very expensive process. However, when RUDIK is executed on all the predicates of a KB, results are consistent in terms of number of discovered rules and execution times. For example, for 600 predicates in DBPEDIA we mined about 3000 positive rules, with at most 26 rules for a predicate, and 4000 negative rules, with at most 32 rules for a predicate.

Positive Rules RUDIK. We evaluate the precision for the positive discovered rules on the top 5 predicates for each KB. The number of new induced facts varies significantly from rule to rule. To avoid the overall precision to be dominated by such rules, we first compute the precision for each rule, and then average values over all induced rules. Table II reports precision values, along with predicates average running time, and the number of manually annotated triples. We distinguish predicates for which we knew there existed at least one correct rule (in bold), and all predicates (in brackets).

TABLE II. RUDIK POSITIVE RULES ACCURACY.

KB	*Avg. RunTime*	*Avg. Precision over Predicates with Rules (All)*	*# Labeled Triples*
DBPEDIA	35min	**87.86**% (63.99%)	139
YAGO 3	59min	**79.17**% (62.86%)	150
WIKIDATA	141min	**85.71**% (73.33%)	180

As precision varies across different KBs and facts, we report the value for every predicate. For DBPEDIA: academicAdvisor (100%), child (58%), spouse (97%), founder (no valid rules), successor (68%). YAGO: hasChild (50%), influences (35%), isLeaderOf (70%), isMarriedTo (100%), exports (83%). WIKIDATA: spouse (100%), child (76%), paintingCreator (60%), academicAdvisor (100%), subsidiary (67%). Average precision values are brought down by few predicates, such as `founder`, where meaningful positive rules probably do not exist at all. Our experience show that it suffices to read the rules to recognize that they are semantically wrong and should be discarded, e.g., a human immediately sees that it is not possible to derive a founder from the KB's predicates.

The running time is influenced by the size of the KB. The more edges we have on average for a node (entity), the more alternative paths we test while traversing the graph. Another relevant aspect is the target predicate involved. Some entities have a large number of outgoing and incoming edges, e.g., entity "*United States*" in WIKIDATA has more than 600K. When the generation set includes such entities, the navigation of the graph is slower. Parameter $maxPathLen$ also impacts the running time. The longer the rule, the bigger is the search space, as we discuss in Section VI-D.

TABLE III. RUDIK NEGATIVE RULES ACCURACY.

KB	*Avg. Run Time*	*# Pot. Errors*	*Precision*
DBPEDIA	19min	499 (84)	**92.38**%
YAGO 3	10min	2,237 (90)	**90.61**%
WIKIDATA	65min	1,776 (105)	**73.99**%

Negative Rules RUDIK. We evaluate discovered negative rules as the percentage of correct errors identified for the top 5 predicates in each KB. Table III shows, for each KB, the total number of potential erroneous triples found with the discovered rules, whereas the precision is computed as the percentage of actual errors among potential errors. Numbers in brackets show the number of triples manually annotated to obtain the precision. At the predicate level, the results are the following. DBPEDIA: academicAdvisor (29%), child (90%), spouse (87%), founder (95%), ceremonialCounty (100%). YAGO: hasChild (82%), isMarriedTo (97%), created (100%), hasAcademicAdvisor (100%), wroteMusicFor (43%). WIKIDATA: spouse (78%), child (82%), founder (100%), creator (48%), oathGiven (100%).

Negative rules have better accuracy than positive ones when considering all predicates. This is due to the fact that negative rules exist more often than positive rules. While quality of the rules is good, especially on the more noisy KBs, we also discover rules that are supported by the large majority of the data, but do not hold semantically. For example, we identify the rule that two people with same gender cannot be married both in YAGO and WIKIDATA. Such rule has a 94% precision in YAGO and 57% in WIKIDATA. Differently from positive rules, literals play a key role in negative rules. In fact, several correct negative rules rely on temporal aspects in which something cannot happen before/after something else.

TABLE IV. AMIE DATASET CHARACTERISTICS.

KB	*Size*	*#Triples*	*#Predicates*	*#rdf:type*
DBPEDIA	551M	7M	10,342	22.2M
YAGO 2	48M	948.3K	38	77.9M

Temporal information is usually expressed through dates and years, which are represented as literal values in KBs.

Discovering negative rules is faster than discovering positive rules because of the different nature of the examples covered by validation queries. Whenever we identify a candidate rule, we execute the body of the rule against the KB with a SPARQL query to compute its coverage over the validation set. These queries are faster for negative rules since the validation set only contains entities directly connected by the target predicate, whereas in the positive case the validation set corresponds to counter examples that do not have this property and are more expensive to evaluate.

For non popular predicates, the system found rules with quality comparable to the popular predicates. For example, it discovers the valid negative rule routeStart$(x, a) \wedge$ routeEnd$(x, b) \Rightarrow$ notMeetingRoad(a, b) for predicate meetingRoad with just 114 facts in DBPEDIA, and the valid positive rule highestState$(a, x) \wedge$ municipality$(b, x) \Rightarrow$ highestRegion(a, b) for predicate highestRegion with just 36 facts.

B. Comparative Evaluation

We compared our methods against AMIE [13], a state-of-the-art positive rule discovery system for KBs. AMIE assumes that the given KB fits into memory and discovers positive rules for every predicate. It then outputs all rules that exceed a given threshold and ranks them according to a coverage function.

Given its in-memory implementation, AMIE went out of memory for the KBs of Table I on our machine. Thus, we used the modified versions of YAGO and DBPEDIA from the AMIE paper [13], which are devoid of literals and rdf:type facts. Removing literals and rdf:type triples drastically reduce the size of the KB. Since our approach needs type information (for the generation of G and V and for the discovery of inequality atoms), we run AMIE on its original datasets, while for our algorithm we used the AMIE dataset plus rdf:type triples. Last column of Table IV reports the number of triples added to the original AMIE dataset.

Positive Rules Comparison. For this experiment we ran RUDIK as follows: we first list all the predicates in the KB that connect a *subject* to an *object*. We then computed for both subject and object the most popular rdf:type that is not super class of any other most popular type. We finally ran our approach sequentially on every predicate, with

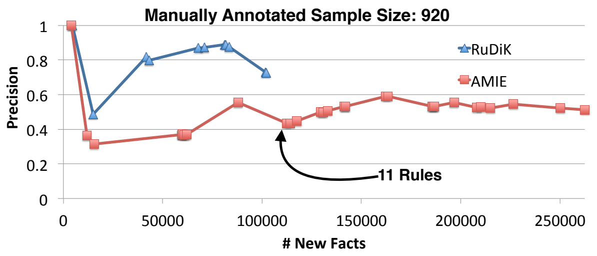


Fig. 3. Accuracy for new facts identified by executing rules in descending AMIE's score on YAGO 2 (no literals).

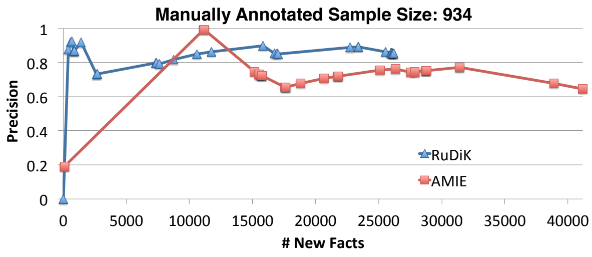


Fig. 4. Accuracy for new facts identified by executing rules in descending score on DBPEDIA (no literals).

$maxPathLen = 2$ (AMIE default setting). AMIE discovers 75 output rules in YAGO, and 6090 in DBPEDIA. We followed their experimental setting and picked the first 30 best rules according to their score. We then picked the rules produced by our approach on the same head predicate of the 30 best rules output of AMIE.

Figures 3 and 4 report the results on YAGO and DBPEDIA, respectively. We plot the total cumulative number of new unique facts (x-axis) versus the aggregated precision (y-axis) when incrementally including in the solution the rules according to their descending (AMIE's) score. Rules from AMIE produce more predictions, but with significant lower accuracy in both KBs. This is because many good rules are preceded by meaningless ones in the ranking, and it is not clear how to set a proper k to get the best ones. In RUDIK, instead of the conventional ranking mechanism, we use a scoring function that discovers only inherently meaningful rules with enough support. As a consequence, RUDIK outputs just 11 rules for 8 target predicates on the entire YAGO – for the remaining predicates RUDIK does not find any rule with enough support. If we limit the output of AMIE to the best 11 rules in YAGO (same output as our approach), its final accuracy is still 29% below our approach, with just 10K more predictions.

Negative Rules Comparison. While AMIE has not been designed to discover negative rules, we created a baseline solution on top of it. First, we created a set of negative examples (Section IV-B) for each predicate in the top-5. For each example, we added a new fact to the KB by connecting the two entities with the *negation* of the predicate. For example, we added a notSpouse predicate connecting each pair of people who are not married according to our generation technique. We then ran AMIE on these new predicates.

Table V shows that RUDIK outperforms AMIE in both cases with an absolute precision gain of almost 20% (41-49% relative). The drop in quality for RUDIK w.r.t. the results in Section VI-A is because of the KBs without literals. Numbers in brackets show the number of triples manually annotated.

TABLE V. NEGATIVE RULES VS AMIE.

	AMIE		RUDIK (no literals)	
KB	*# Errors*	*Precision*	*# Errors*	*Precision*
DBPEDIA	457 (157)	38.85%	148 (73)	**57.76%**
YAGO 2	633 (100)	48.81%	550 (35)	**68.73%**

Running Time. On our machine, AMIE could finish the computation on YAGO 2, while for other KBs it got stuck after some time. For these cases, we stopped the computation if there were no changes in the output for more than 2 hours. Running times for AMIE are different from [13], where it was run on a 48GB RAM server.

TABLE VI. TOTAL RUN TIME COMPARISON.

KB	*#Predicates*	*AMIE*	RUDIK	*Types*
YAGO 2	20	30s	18m,15s	12s
YAGO 2s	26 (38)	> 8h	47m,10s	11s
DBPEDIA 2.0	904 (10342)	> 10h	7h,12m	77s
DBPEDIA 3.8	237 (649)	> 15h	8h,10m	37s
WIKIDATA	118 (430)	> 25h	8h,2m	11s
YAGO 3	72	-	2h,35m	128s

Table VI reports the running time on different KBs. The first five KBs are AMIE modified versions, while YAGO 3 includes literals and `rdf:type`. The second column shows the total number of predicates for which AMIE produced at least one rule before getting stuck, while in brackets we report the total number of predicates in the KB. The third and fourth columns report the total running time of the two approaches. Despite being disk-based, RUDIK successfully completes the task faster than AMIE in all cases, except for YAGO 2. This is because of the very small size of this KB, which fits in memory. However, when we deal with complete KBs (YAGO 3), the KB could not even be loaded due to out of memory errors. The last column reports the running time to compute `rdf:type` information for all predicates.

Other Systems. In [2], the system mines rules that are less general than our approach; on YAGO 2, it discovers 2K new facts with a precision lower than 70%, while our rule on YAGO 2 already produces more than 4K facts with a 100% precision. Another system [5] implements AMIE algorithm with a focus on scalability and the output is the same as AMIE. We did not compare with Inductive Logic Programming systems [8], [23], as these are already significantly outperformed by AMIE both in accuracy and running time.

C. Machine Learning Application

The goal of this experiment is to test RUDIK's ability in providing valid training examples to ML models. We chose DeepDive [19], a framework for information extraction. DeepDive extracts entities and relations from text articles via distant supervision. The key idea in distant supervision is to use an external source of information (e.g., a KB) to provide training examples for a supervised algorithm. For example, DeepDive can extract mentions of married couples from text documents. In this scenario, it uses a KB to label pairs of married couples that can be found in DBPEDIA as *true* positive example. As KBs provide facts, in DeepDive the burden of creating negative examples is left to the user. We compare the output of DeepDive upon its spouse example trained with different sets of negative examples over two datasets.

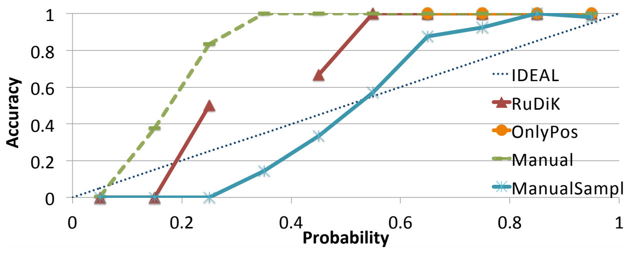


Fig. 5. DeepDive executions with different training examples – 1K articles.

Figure 5 shows DeepDive accuracy plot with 1K input documents. The plot shows the fraction of correct positive predictions over total predictions (y-axis), for each output probability value (x-axis). The ideal execution, marked by the dotted blue line, would predict all facts with a probability of 1 and zero facts with an output probability of 0. The best algorithm deflects the least from the blue dotted line, and this distance is our evaluation metric. RUDIK is the output of DeepDive using our discovered rules to generate negative examples on DBPEDIA. `OnlyPos` uses only positive examples from DBPEDIA, `Manual` uses positive examples from DBPEDIA and manually defined rules to generate negative examples, while `ManualSampl` uses a sample of the manually generated negative examples in size equal to positive examples. `OnlyPos` and `Manual` do not provide valid training, as the former has only positive examples and labels everything as true, while the latter has many more negative examples than positive ones and labels everything as false. `ManualSampl` is the winner, while our approach suffers from the absence of data to mine: over the input 1K articles, there are only 20 positive examples from DBPEDIA. The lack of evidence in the training data also explains the missing points for RUDIK, with no prediction in the probability range 25-45%.

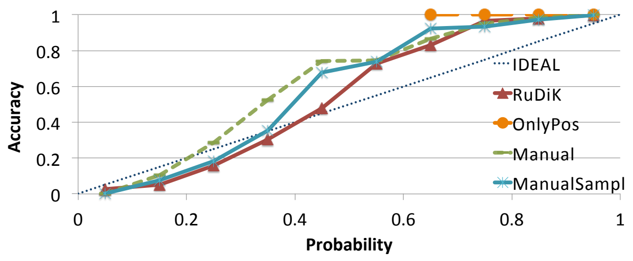


Fig. 6. DeepDive executions with different training examples – 1M articles.

When we extend the input to 1M articles, things change drastically (Figure 6). All approaches except `OnlyPos` successfully drive the training, with the examples provided with RUDIK leading to the best result. This is because of the quality of the negative examples: our rules generate representative examples that are correct (thanks to the LCWA), semantically related (thanks to the constraint on the predicate connecting them), and have the number of negative examples in the same order of magnitude of the positive ones. The correct and rich examples enable DeepDive to identify discriminatory features between positive and negative labels. The output of `ManualSampl` and RUDIK are very similar, meaning that we can use our approach to simulate user behavior and automatically produce negative examples.

D. Internal Evaluation

We outline the impact of individual components in RUDIK. Full results are reported in the technical report online at http://www.eurecom.fr/publication/5321.

KB Noise Impact. In terms of quality of the KBs, the percentages of erroneous triples identified by our rules are 0.23% for WIKIDATA, 0.26% for DBPEDIA, and 0.6% for YAGO. To study the impact of errors in the KB, we first manually removed errors from the top five predicates in DBPEDIA to obtain clean positive and negative examples. We collected such rules and consider them the best possible output. We then gradually introduced errors by switching positive and negative examples between their sets. Figure 7 shows the accuracy degradation

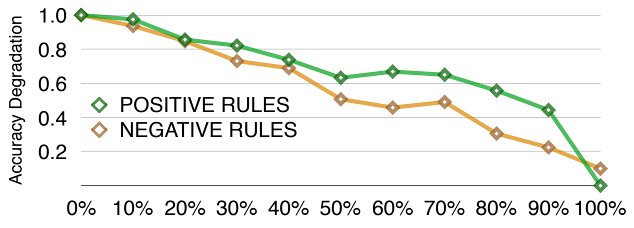


Fig. 7. KB Noise Impact on Rules Quality.

averaged over predicates (y-axis) from 0% errors to 100% (x-axis). As expected, the accuracy decreases with the amount of errors. RUDIK is robust enough to deliver mostly correct rules until 40% of errors, while after that accuracy starts to drop significantly. An interesting point is that even with 90% of errors, RUDIK is still able to isolate the 10% of good examples to mine at least one valid rule.

LCWA. We study the effect of the LCWA assumption for the generation of negative examples. Given a predicate p, we tested three generation strategies: RUDIK strategy (Section IV-B), Random (randomly select k pairs (x, y) from the Cartesian product s.t. triple $\langle x, p, y\rangle \notin kb$), and LCWA (RUDIK strategy but x and y do not have to be connected by a predicate different from p). Table VII reports quality results for the discovered rules. *Random* and *LCWA* show similar behavior, with a slightly better precision than RUDIK. This is because by randomly picking examples from the Cartesian product of subject and object, the likelihood of getting entities from different time periods is very high, and negative rules pivoting on time constraints are usually correct. Instead, by forcing x and y to be connected with different predicate, we generate semantically related examples that lead to more rules. Rules such as `parent`$(a, b) \Rightarrow$ `notSpouse`(a, b) are not generated with random strategies, since the likelihood of picking two people that are in a parent relation is very low. The RUDIK strategy enables the discovery of more types of rules, and not only rules involving time constraints.

TABLE VII. IMPACT OF EXAMPLES GENERATION ON DBPEDIA.

Strategy	*# Potential Errors*	*Precision*
Random	247	**95.95%**
LCWA	263	95.82%
RUDIK	**499**	92.38%

Effect of Literals. Table VIII reports the output precision obtained by enabling and disabling the use of literal comparisons in RUDIK. Including literal values has a considerable impact on accuracy, both for positive and negative rules. Negative rules without literals find less than half potential errors (numbers in brackets) with lower precision. For predicate `founder`, RUDIK discovers 79 potential errors with a 95% precision with literal rules, while none are detected by using rules without literals. Interestingly, including literals reduces also the running time. This is due to the pruning effect of the A^* search, literals enable the early discovery of good rules.

Rule Length Impact. The $maxPathLen$ parameter fixes the maximum number of atoms in the body of a rule. Low

TABLE VIII. IMPACT OF LITERALS ON DBPEDIA.

	With Literals		**Without Literals**	
Rules	*Run Time*	*Precision*	*Run Time*	*Precision*
Pos.	~35min	**63.99%**	~54min	60.49%
Neg.	~19min	**92.38%** (**499**)	~25min	84.85% (235)

values may exclude from the search space meaningful rules, while high values exponentially increase the search space and consequently the running time. With $maxPathLen = 2$, there is a significant improvement in running time, but meaningful rules are lost and precision drop to 49% for positive rules and 90% for negative ones. In particular, we lose rules with literals comparison, as these require at least three atoms in the body. At the other side of the spectrum, with $maxPathLen = 4$ the search space explodes and RUDIK could not finish the computation within 24 hours for any predicate. We measured the accuracy of rules discovered in 24 hours of computation and the results are comparable to those computed with $maxPathLen = 3$, with a small increase in precision for positive rules and a small drop for negative ones. Rules with length 4 are more complex to understand, and when executed over the KB they often return an empty result because of their higher selectivity. We therefore set $maxPathLen = 3$ as a compromise between efficiency and accuracy.

Weight Parameters. For positive rules, the best assignment is $\alpha = 0.3$ and $\beta = 0.7$, while for negative rules is $\alpha = 0.4$ and $\beta = 0.6$. Since discovering correct positive rules is more challenging than negative ones, favoring precision over recall gives the best accuracy, while for negative rules we can be more recall oriented. In both positive and negative settings, the variation in performance for $\alpha \in [0.1, 0.9]$ is limited ($\leqslant 12\%$), showing the robustness of the set cover problem formulation.

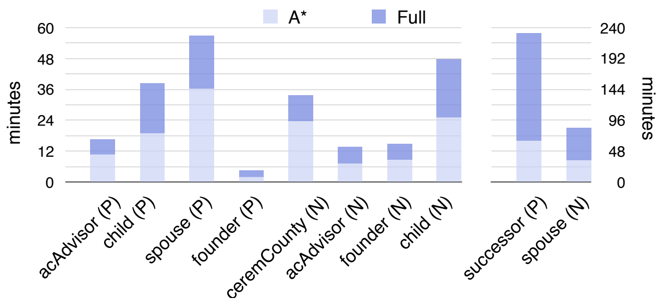


Fig. 8. A^* Pruning Runtime Improvement.

Search. We quantify the benefit of the A^* algorithm on the running time. Figure 8 shows the running time, for each predicate, of the A^* algorithm (light-colored bars) against a modified version that first generates the universe of all possible rules, and then applies the greedy set cover algorithm on such a universe (dark-colored bars). The last two predicates refer to the y-axis labels on the right hand side, as they have higher running times. In the figure, `(P)` indicates positive rules and `(N)` negative ones. The A^* strategy shows an average 50% improvement in running times as it avoids the generation of unpromising paths and the loading of the corresponding RDF instances from disk. When there exist rules that cover many examples from the generation set (e.g., `successor (P)`, `founder (P)`), the algorithm identifies such rules rather early, thus pruning several unpromising paths. In such cases the running time improvement is above 70%.

Set Cover. Our set cover problem formulation leads to a concise set of rules in the output, which is preferable to the large set of rules obtained with a ranking based solution. Oftentimes correct rules are not among the top-10 ranked, and we found cases where meaningful rules are below the 100^{th} position. For example, the only valid negative rule for the predicate `founder`, which states that a person born after the company was founded cannot be its founder, figures at a rank of 127 when emitted by the ranking-based version of RUDIK, whereas it is included in the compact set discovered by the standard variant of RUDIK.

VII. Related Work

A significant body of work has addressed the problem of discovering constraints over *relational data*, e.g., [6]. However, these techniques cannot be applied to KBs because of the schema-less nature of RDF data and the OWA. Traditional approaches rely on the assumption that data is either clean or has a negligible amount of errors, which is not the case with KBs, and, even when the algorithms are designed to tolerate errors [1], [15], a direct application of relational database techniques on RDF KBs requires the prohibitive materialization of all possible predicate combinations into relational tables. Recently, theoretical foundations of Functional Dependencies on Graphs have been laid [11]. However, their language covers only a portion of our negative rules and does not include general literal comparisons.

Rule mining approaches designed for positive rule discovery in RDF KBs load the entire KB into memory prior to the graph traversal step [13], [5]. This is a limitation for their applicability over large KBs, and neither of these two approaches consider value comparison. In contrast to them, RUDIK load in memory a small fraction of the KB. This makes it scalable and the low memory footprint enables a bigger search space with rules that have literal comparisons. Finally, association rules can be mined to recommend new facts [2], but such rules are made of constants only and are therefore less general than the rules generated by RUDIK.

ILP systems such as WARMR [8], Sherlock [18], and ALEPH[1] are designed to work under the CWA and require the definition of positive and negative error-free examples. It has been showed how this assumption does not hold in KBs and that AMIE outperforms this kind of systems [13]. Detection of semantic errors in KBs has also been tackled with approaches that are orthogonal to negative rules. For example, discovering domain and range restrictions [23], or identifying outliers after grouping subjects by type [25]. Finally, the output of our rules can be modeled as the result of a link prediction problem over the KB [10]. However, we focus on logical rules for their benefits as "white boxes", including the possibility of doing static analysis, execution optimization, and interpretability.

VIII. Conclusion

We presented RUDIK, a rule discovery system that mines both positive and negative rules on noisy and incomplete KBs. Positive rules identify new valid facts for the KB, while negative rules identify errors. We experimentally showed that our approach generates concise sets of meaningful rules with high precision, is scalable, and can work with exisisting KBs.

Open questions are related to the interactive discovery of the rules, if and how it is possible to drastically reduce the runtime of the discovery without compromising the quality of the rules. Another interesting direction is to discover more expressive rules that exploit temporal information through smarter analysis of literals [1], e.g., "if two person have age difference greater than 100 years, then they cannot be married".

[1]https://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph

Acknowledgement. This work was partly supported by Google with a Faculty Research Award.

References

[1] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *PVLDB*, 9(4):336–347, 2015.

[2] Z. Abedjan and F. Naumann. Amending RDF entities with new facts. In *ESWC*, pages 131–143, 2014.

[3] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia-A crystallization point for the web of data. *J. Web Semantics*, 7(3):154–165, 2009.

[4] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250, 2008.

[5] Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri. Ontological pathfinding. In *SIGMOD*, pages 835–846, 2016.

[6] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[7] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[8] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data mining and knowledge discovery*, 3(1):7–36, 1999.

[9] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: a report from the trenches. In *SIGMOD*, pages 1209–1220, 2013.

[10] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD*, 2014.

[11] W. Fan, Y. Wu, and J. Xu. Functional dependencies for graphs. In *SIGMOD*, pages 1843–1857, 2016.

[12] M. H. Farid, A. Roatis, I. F. Ilyas, H. Hoffmann, and X. Chu. CLAMS: bringing quality to data lakes. In *SIGMOD*, pages 2089–2092, 2016.

[13] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *The VLDB Journal*, 24(6):707–730, 2015.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *TCS*, 149(1):129–149, 1995.

[16] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

[17] J. Pujara, H. Miao, L. Getoor, and W. Cohen. Knowledge graph identification. In *ISWC*, pages 542–557, 2013.

[18] S. Schoenmackers, O. Etzioni, D. S. Weld, and J. Davis. Learning first-order horn clauses from web text. In *EMNLP*, 2010.

[19] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using DeepDive. *PVLDB*, 8(11):1310–1321, 2015.

[20] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge unifying Wordnet and Wikipedia. In *WWW*, 2007.

[21] F. M. Suchanek, M. Sozio, and G. Weikum. SOFIE: A self-organizing framework for information extraction. In *WWW*, pages 631–640, 2009.

[22] P. Suganthan GC, C. Sun, H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, et al. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, pages 265–276, 2015.

[23] G. Töpper, M. Knuth, and H. Sack. Dbpedia ontology enrichment for inconsistency detection. In *I-SEMANTICS*, pages 33–40, 2012.

[24] D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[25] D. Wienand and H. Paulheim. Detecting incorrect numerical data in dbpedia. In *ESWC*, 2014.

Chapter 3

Data repairing with declarative rules

In this part of the thesis, I present my work on the development of scalable and accurate algorithms for data repairing. The starting point of this line of work is based on the core idea of designing algorithms that can consume as many user specifications as possible. In fact, identifying errors and their possible repair updates is very hard in general and any external information is valuable. Based on this idea, I have explored four orthogonal directions: the use of expressive rule languages [28, 48], the combination of heterogeneous specifications [48] and external resources [30], and the involvement of the users in an interactive cleaning process where rule discovery and data repairing are interleaved [52, 30].

Resources and grants: These activities involved a number of students and collaborators. The data repairing work on denial constraints [28] involved my time as well as that of an intern at QCRI under my co-supervision. The general cleaning framework [48] involved my time in a collaboration with University of Basilicata and Antwerp University (one PhD student and two academics). The work exploiting knowledge bases and crowdsourcing [30] was led by Ph.D. students visiting QCRI under my co-supervision, and in collaboration with colleagues within the Data Analytics group at QCRI. For the interactive cleaning [52], the work was led by a Ph.D. student under my co-supervision and in collaboration with other institutions. Most of the funding for these activities came from the national Qatar funding body and a 1-year private ASURE Research Grant during the time I spent in Arizona at ASU.

- **Holistic data cleaning.** A rule-based data cleaning system takes as input a noisy dataset, a collection of rules expressed in the supported specification language for such dataset, and produces a new version of the dataset that satisfies the rules. Algorithms have been proposed to do repairing based on rules defined by different integrity constraints (ICs) [39, 38, 18, 15, 46, 40, 67]. These algorithms try to find a consistent database that satisfies the given ICs with a minimum cost, e.g., with minimal changes to the original datasets [36, 95, 65, 74, 107, 19, 16].

It has been shown that, in general, all languages are useful to capture different errors, and that coupling different methods, such as the combination of IC-based repair with lookup approaches [45], leads to better quality in the repair. However, previously proposed data repairing algorithms focus on fixing violations that belong to each class of constraints in isolation, e.g., FD violation repairs. These techniques miss the opportunity of considering the interaction among different classes of constraints violations. This problem motivated us to study methods to correct violations for different types of constrains with desirable repairs in a unified algorithm [28]. To this end, we adopted rules expressed as denial constraints (DCs), as they are able to cover existing heterogeneous formalisms. Given a set of DCs and a database to be cleaned, our approach starts by compiling the violations over the instance in a unified representation based on a hypergraph, so that, by analyzing their interaction, it is possible to identify the cells that are more likely to be wrong. Once we have identified what are the cells that are most likely to change, we process their violations to get information about how to repair them. In the last step, heterogeneous requirements from different constraints are holistically combined in order to fix the violations.

We verified experimentally the effectiveness and scalability of the algorithm, which outperforms state of the art solutions in all scenarios. What we also found in follow up work is that the hypergraph representation and the proposed repair algorithm can also handle rules expressed as procedural code, as long as their output violations as sets of cells and repair conditions over cell combinations [64]. This showed that the technical contributions of this work go beyond DCs as input specifications.

- **The LLUNATIC DataCleaning Framework.** While the ICs are mandatory specifications for data repair, it is known that the repair algorithm alone cannot always ensure the accuracy of data cleaning needed in several real world applications [46, 3]. In order to improve the accuracy, I have explored several directions. The first one is a framework that can model as input the declarative rules together with other specifications that express confidence and preferences among values and possible repairs [45, 48, 44, 43, 100]. The vision of the framework is to be as general as possible, thus allowing different rule languages (including denial constraints with equalities only), different strategies to use the constraints to modify the noisy data (master data, tuple-certainty, value-accuracy, freshness and currency), and even the set of restrictions that are imposed on the target repaired instance to limit the search space in the process, such as different definitions of (cost-based) repair minimality, the use of certain fixes, and the adoption of sampling techniques.

  To achieve such an ambitious goal, our work starts in the definition of a novel semantics for the data-repairing problem. We formalize the process of cleaning an instance as the process of upgrading its quality, regardless of the specific notions of value preference adopted in a given scenario. This enables users to plug-in their preference strategies for a given scenario into the semantics (e.g., currency). With a clear semantics defined, we then introduce the notion of a minimal solution and develop algorithms to compute it, based on a parallel-chase procedure. Finally, as a plug-in for the chase algorithm, we define the notion of a cost manager that selects which repairs should be kept and which ones should be discarded. The cost manager abstracts and generalizes popular solution selection strategies, including similarity-based cost, set-minimality, set-cardinality minimality, certain regions, and sampling.

In our experiments, we show that the chase engine at the core of the system enable us to scale the cleaning process to databases with millions of tuples, a considerable advancement in scalability wrt previous main memory implementations[1]. Our experience with this engine also led us into the design of a benchmark for chase algorithms [13], that has ultimately been released as an open source environment[2].

- **KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing.** A clear source of evidence to steer the cleaning process based on rules is reference master data [46, 106, 60], i.e., data that is considered curated and therefore trusted. In this context, we studied how to exploit emerging resources expressed as knowledge bases (KBs), both the ones that are general-purpose, such as Yago[3]) and DBpedia[4]), and the ones that are developed within enterprises or in domain-specific community (e.g., RxNorm[5]). Here the major question is how to increase the accuracy of data cleaning methods by exploiting external information that is incomplete and not available in tabular form. In fact, matching (noisy) tables to KBs is a hard problem. On one hand, tables may lack reliable, comprehensible labels, thus requiring the matching to be executed on the data values. This may lead to ambiguity; more than one mapping may be possible. Moreover, tables usually contain errors that may trigger problems such as erroneous matching. On the other hand, while in some domains it is reasonable to assume curated KBs, those are usually incomplete (OWA). This makes the matching to the noisy tables even harder and the cleaning process not obvious. In the case of failing to find a match, it is not clear whether the database values are erroneous or the KB does not cover these values.

  To solve these problems, we exploit a second resource: crowdsourcing marketplaces, such as Amazon Mechanical Turk[6] and Figure Eight[7]. While access to one domain expert may be limited and expensive, crowdsourcing to a population of users familiar with the topic has been proven to be a viable alternative solution. Human involvement is our

[1]`http://db.unibas.it/projects/llunatic/`

[2]`https://dbunibas.github.io/chasebench/`

[3]`https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`. While we found errors in Yago [80], it indeed has a “confirmed accuracy of 95%”, which we found high enough to help in cleaning several datasets in practice.

[4]`https://wiki.dbpedia.org/`

[5]`https://www.nlm.nih.gov/research/umls/rxnorm/`

[6]`https://www.mturk.com/`

[7]formerly CrowdFlower, `https://www.figure-eight.com/`

proposal to validate matchings and to verify data when the KBs do not have enough coverage. Effectively involving the crowd requires dealing with traditional crowdsourcing issues such as forming easy-to-answer questions for the new data cleaning tasks and optimizing the order of issuing questions to reduce monetary cost.

These ideas are deployed in a knowledge base and crowd powered data cleaning system that, given a table, a KB, and a crowd, interprets table semantics to align it with the KB, identifies correct and incorrect data, and generates top-k possible repairs for incorrect data [30]. At the core of the solution there is a new class of matching patterns between tables and graph KBs, which explain table semantics by using the (semantically rich) predicates in the graphs. Each matching pattern is a directed graph, where a node represents a type of a column and a directed edge represents a binary relationship between two columns. Our rank-join algorithm discovers matching patterns that are then validated via crowdsourcing. To minimize the number of questions, we use a scheduling algorithm to maximize the uncertainty reduction of candidate patterns. Finally, given a matching pattern, we annotate data as (i) correct data validated by the KB; (ii) correct data jointly validated by the KB and the crowd; and (iii) erroneous data jointly identified by the KB and the crowd. For the erroneous data we also generate top-k possible repairs from the evidence in the KB. Experiments show that our system can be applied to various datasets and KBs to effectively annotates data.

- **Interactive and Deterministic Data Cleaning.** While the mining approaches introduced in the first part of the thesis have proven to be useful in practice, they still have limitations that make them hard to use in settings with noisy datasets. In fact, despite the many results in this area, discovering specification from noisy data is challenging for multiple reasons. (1) In the bootstrapping step, several input parameters strongly impact the final output, but are very hard to set upfront. Examples of such parameters include the percentage of tolerance to noise to discover approximate rules, or the way to select constants to be considered for rule discovery. These parameters are rarely known apriori, but tuning them with a trial-and-error approach is infeasible, given the large number of possible value combinations and the long execution times for the mining, as discussed next. (2) Complexity comes from both the size of the schema and the size of the data, with an exponential dependency on the number of attributes, as all their combinations must be tested [27]. Moreover, if the language supports complex pairwise rules, such as denial constraints or de-duplication rules, the complexity is quadratic over the number of tuples. (3) Finally, the number of discovered specifications that hold over the data is usually large, especially when constants and approximate rules, often needed in practice, are allowed. When tolerance to noise in the data is required, semantically valid rules are mixed with incorrect rules because of noisy values in the data. This problem is alleviated by pruning mechanisms and ranking, but ultimately it leads to a large amount of time spent by the data engineer and the domain expert to identify the valid rules among the thousands that approximately hold on the data.

To alleviate these problems, several approaches have been proposed for the interactive definition of cleaning specifications. More precisely, methods reason about user-provided updated values to learn transformation specifications [87, 54, 108, 92, 8, 1]. These algorithms focus on string manipulation and reformatting at the text level for one attribute. In a similar interaction, several proposals have exploited data examples for specific tasks such as discovering queries [22, 115, 91, 114, 5, 20], schema matchings [112, 85], schema mappings [6, 51, 21], or parameters for entity resolution [104, 103]. Successful commercialization of related research projects also demonstrate that data engineers successfully engage with this kind of interfaces [96, 61].

Rule definition by example is therefore one of the most promising directions for cleaning, as it places the human at the center of the data preparation process. As most methods discovering transformations focus on specification involving one attribute only, we wanted to design a system to interactively discover with the user rules that can span over multiple attributes.

Our answer to this challenge is a cleaning system that does not rely on the existence of a set of predefined data quality rules. On the contrary, it takes as input a single user update and guesses a set of possible single tuple constant CFD (presented as deterministic SQL update queries), which that can be used to repair the data. The main technical challenge addressed in this work is the identification of the rule that fixes the largest number of errors in the data with a constraint on the number of interactions with the human. In other words, what are the right questions to ask to the users in order to minimize the manual cost to go from the single update to the most general (correct) rule? We formalize this problem as a search in a lattice-shaped space. We navigate the lattice by interacting with users to gradually validate the set of possible rules. To speed up the search, we go beyond the traditional one-hop based traverse algorithms (e.g., BFS or DFS) and design novel multi-hop search algorithms. Experiments show that the system effectively steer the users towards correct and general rules that are useful for data cleaning.

Holistic Data Cleaning: Putting Violations Into Context

Xu Chu[1]⋆ Ihab F. Ilyas[2] Paolo Papotti[2]

[1] *University of Waterloo, Canada* [2] *Qatar Computing Research Institute (QCRI), Qatar*
x4chu@uwaterloo.ca, {ikaldas,ppapotti}@qf.org.qa

***Abstract*—Data cleaning is an important problem and data quality rules are the most promising way to face it with a declarative approach. Previous work has focused on specific formalisms, such as functional dependencies (FDs), conditional functional dependencies (CFDs), and matching dependencies (MDs), and those have always been studied in isolation. Moreover, such techniques are usually applied in a pipeline or interleaved.**

In this work we tackle the problem in a novel, unified framework. First, we let users specify quality rules using denial constraints with ad-hoc predicates. This language subsumes existing formalisms and can express rules involving numerical values, with predicates such as "greater than" and "less than". More importantly, we exploit the interaction of the heterogeneous constraints by encoding them in a conflict hypergraph. Such holistic view of the conflicts is the starting point for a novel definition of *repair context* which allows us to compute automatically repairs of better quality w.r.t. previous approaches in the literature. Experimental results on real datasets show that the holistic approach outperforms previous algorithms in terms of quality and efficiency of the repair.

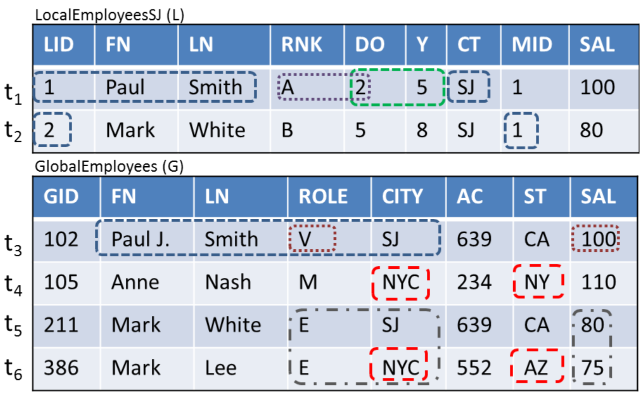
I. Introduction

It is well recognized that business and scientific data are growing exponentially and that they have become a first-class asset for any institution. However, the quality of such data is compromised by sources of noise that are hard to remove in the data life-cycle: imprecision of extractors in computer-assisted data acquisition may lead to missing values, heterogeneity in formats in data integration from multiple sources may introduce duplicate records, and human errors in data entry can violate declared integrity constraints. These issues compromise querying and analysis tasks, with possible damage in billions of dollars [9]. Given the value of clean data for any operation, the ability to improve their quality is a key requirement for effective data management.

Data cleaning refers to the process of detecting and correcting errors in data. Various types of data quality rules have been proposed for this goal and great efforts have been made to improve the effectiveness and efficiency of their cleaning algorithms (e.g., [4], [8], [21], [18], [13]). Currently existing techniques are used in isolation. One naive way to enforce all would be to cascade them in a pipeline where different algorithms are used as black boxes to be executed sequentially or in an interleaved way. This approach minimizes the complexity of the problem as it does not consider the interaction between different types of rules. However, this simplification can compromise the quality in the final repair due to the lack of end-to-end quality enforcement mechanism as we show in this paper.

Example 1.1: Consider the GlobalEmployees table (G for short) in Figure 1. Every tuple specifies an employee in a company with her id (GID), name (FN, LN), role, city, area code (AC), state (ST), and salary (SAL). We consider only two rules for now. The first is a functional dependency (FD) stating that the city values determine the values for the state attribute. We can see that cells in t_4 and t_6 present a violation for this FD: they have the same value for the city, but different states. We highlight the set S_1 of four cells involved in the violation in the figure. The second rule states that among employees having the same role, salaries in NYC should be higher. In this case cells in t_5 and t_6 are violating the rule, since employee Lee (in NYC) is earning less than White (in SJ). The set S_2 of six cells involved in the violation between Lee and White is also highlighted.

The two rules detect that at least one value in each set of cells is wrong, but taken individually they offer no knowledge of which cells are the erroneous ones. ◇


LocalEmployeesSJ (L)

	LID	FN	LN	RNK	DO	Y	CT	MID	SAL
t_1	1	Paul	Smith	A	2	5	SJ	1	100
t_2	2	Mark	White	B	5	8	SJ	1	80

GlobalEmployees (G)

	GID	FN	LN	ROLE	CITY	AC	ST	SAL
t_3	102	Paul J.	Smith	V	SJ	639	CA	100
t_4	105	Anne	Nash	M	NYC	234	NY	110
t_5	211	Mark	White	E	SJ	639	CA	80
t_6	386	Mark	Lee	E	NYC	552	AZ	75

Fig. 1: Local (L) and Global (G) relations for employees data.

Previously proposed data repairing algorithms focus on repairing violations that belong to each class of constraints in isolation, e.g., FD violation repairs [6]. These techniques miss the opportunity of considering the interaction among different classes of constraints violations. For the example above, a

⋆ Work done while interning at QCRI.

desirable repair would update the city attribute for t_6 with a new value, thus only one change in the database would fix the two violations. On the contrary, existing methods would repair the FD by changing one cell in S_1, with an equal chance to pick any of the four by being oblivious to violations in other rules. In particular, most algorithms would change the state value for t_6 to NY or the state for t_4 to AZ. Similarly, rule based approaches, when dealing with application-specific constraints such as the salary constraint above, would change the salaries of t_5 or t_6 in order to satisfy the constraints. None of these choices would fix the mistake for city in t_6, on the contrary, they would add noise to the existing correct data.

This problem motivates the study of novel methods to correct violations for different types of constrains with desirable repairs, where desirability depends on a cost model such as minimizing the number of changes, the number of invented values, or the distance between the value in the original instance and the repair. To this end, we need quality rules that are able to cover existing heterogeneous formalisms and techniques to holistically solve them, while keeping the process automatic and efficient.

Since the focus of this paper is the holistic repair of a set of integrity constraints more general than existing proposals, we introduce a model that accepts as input *Denial Constraints* (DCs), a declarative specification of the quality rules which generalizes and enlarge the current class of constraints for cleaning data. Cleaning algorithms for DCs have been proposed before [4], [20], but they are limited in *scope*, as they repair numeric values only, *generality*, only a subclass of DCs is supported, and in the *cost model*, as they aim at minimizing the distance between original database and repair only. On the contrary, we can repair any value involved in the constraints, we do not have limits on the allowed DCs, and we support multiple quality metrics (including cardinality minimality).

Example 1.2: The two rules described above can be expressed with the following DCs:

$$\begin{aligned} c_1: \quad & \neg(G(g,f,n,r,c,a,s), G(g',f',n',r',c',a',s'), \\ & \qquad (c = c'), (s \neq s')) \\ c_2: \quad & \neg(G(g,f,n,r,c,a,s), G(g',f',n',r',c',a',s'), \\ & \qquad (r = r'), (c = \text{“}NYC\text{”}), (c' \neq \text{“}NYC\text{”}), (s' > s)) \end{aligned}$$

The DC in c_1 corresponds to the FD: $G.CITY \rightarrow G.ST$ and has the usual semantics: if two tuples have the same value for city, they must have the same value for state, otherwise there is a violation. The DC in c_2 states that every time there are two employees with the same rank, one in NYC and one in a different city, there is a violation if the salary of the second is greater than the salary of the first. $\diamond$

Given a set of DCs and a database to be cleaned, our approach starts by compiling the rules into data violations over the instance, so that, by analyzing their interaction, it is possible to identify the cells that are more likely to be wrong. In the example, $t_6[CITY]$ is involved in both violations, so it is the candidate cell for the repair. Once we have identified what are the cells that are most likely to change, we process their violations to get information about how to repair them. In the last step, heterogeneous requirements from different constraints are holistically combined in order to fix the violations. In the case of $t_6[CITY]$, both constraints are satisfied by changing its value to a string different from “NYC”, so we update the cell with a new value.

A. Contributions

We propose a method for the automatic repair of dirty data, by exploiting the evidence collected with the holistic view of the violations:

- We introduce a compilation mechanism to project denial constraints on the current instance and capture the interaction among constraints as overlaps of the violations on the data instance. We compile violations into a Conflict Hypergraph (CH) which generalizes the one previously used in FD repairing [18] and is the first proposal to treat quality rules with different semantics and numerical operators in a unified artifact.
- We present a novel holistic repairing algorithm that repair all violations together w.r.t. one unified objective function. The algorithm is independent of the actual cost model and we present heuristics aiming at cardinality and distance minimality.
- We handle different repair semantics by using a novel concept of Repair Context (RC): a set of expressions abstracting the relationship among attribute values and the heterogeneous requirements to repair them. The RC minimizes the number of cells to be looked at, while guaranteeing soundness.

We verify experimentally the effectiveness and scalability of the algorithm. In order to compare with previous approaches, we use both real-life and synthetic datasets. We show that the proposed solution outperforms state of the art algorithms in all scenarios. We also verify that the algorithms scale well with the size of the dataset and the number of quality rules.

B. Outline

We discuss related work in Section II, introduce preliminary definitions in Section III, and give an overview of the solution in Section IV. Technical details of the repair algorithms are discussed in Section V. System optimizations are discussed in Section VI, while experiments are reported in Section VII. Finally, conclusions and future work are discussed in Section VIII.

II. Related Work

In industry, major database vendors have their own products for data quality management, e.g., IBM InfoSphere QualityStage, SAP BusinessObjects, Oracle Enterprise Data Quality, and Google Refine. These systems typically use simple, low-level ETL procedural steps [3]. On the other hand, in academia, researchers are investigating declarative, constraint-based rules [4], [5], [13], [8], [11], [12], [21], [18], which allow users to detect and repair complicated patterns in the data. However, a unified approach to data cleaning that combines evidence from heterogeneous rules is still missing and it is the subject of this work.

Interleaved application of FDs and MDs has been studied before [13] and some works (e.g., [6], [5], [8]) compute sets of cells that are connected by violations from different FDs. This connected component is usually called "equivalence class" and it is a special case of the notion of repair context that we introduce next. Another work [18] exploits the interaction among FDs by using a hypergraph. In our proposal we extend the use of hypergraphs to denial constraints, thus significantly generalizing the original proposal. Moreover, we simplify it, by considering only current violations, thus avoiding a large number of hyperedges that they compute in order to execute the repair process with a single iteration. In fact, by using multiple iterations we can be more general and compute interactions of rules happening in more than two steps.

In this work we compute repairs with a large class of operators in the quality rules: =, $\neq$, $<$, $>$, $\leq$, $\geq$, $\approx$ (similarity). Most of the previous approaches [6], [8] were dealing only with equality, and can be seen as special cases of our work. Exceptions are [4], [20], where the authors propose algorithms to repair numerical attributes for denial constraints. In our work we extend their results in three important aspects: (i) we treat both strings and numeric values together, thus not restricting updates to numeric values only; (ii) we do not limit the input constraints to local denial constrains; and (iii) we allow multiple quality metrics (including cardinality minimality), while still minimizing the distance between the numeric values in the original and the repaired instances. We show in the experimental study that our algorithms provide repairs of better quality, even for the quality metric in [4].

In general, denial constraints can be extracted from existing business rules with human intervention. Moreover, a source of constraints with numeric values from enterprise databases is data mining [2]. Inferred rules always have a confidence, which clearly points to data quality problems in the instances. For example, a confidence of 98.5% for a rule "discountedPrice<unitPrice" implies that 1.5% of the records require some cleaning.

III. Preliminaries

A. Background

Consider database schema of the form $\mathbb{S} = (\mathbb{U}, \mathbb{R}, \mathbb{B})$, where $\mathbb{U}$ is a set of database domains, $\mathbb{R}$ is a set of database predicates or relations, and $\mathbb{B}$ is a set of finite built-in predicates. In this paper, $\mathbb{B} = \{=, <, >, \neq, \leq, \approx\}$. For an instance I of $\mathbb{S}$, and an attribute $A \in \mathbb{U}$, and a tuple t, we denote by $Dom(A)$ the domain of attribute A. We denote by $t[A]$ or $I(t[A])$ the value of the cell of tuple t under attribute A.

In this work we support the subset of integrity constraints identified by *denial constraints* (DCs) over relational databases. Denial constraints are first-order formulae of the form $\varphi : \forall \overline{x} \neg (R_1(\overline{x}_1) \wedge \ldots \wedge R_n(\overline{x}_n) \wedge P_1 \wedge \ldots \wedge P_m)$, where $R_i \in \mathbb{R}$ is a relation atom, and $\overline{x} = \cup \overline{x_i}$, and each P_i of the form $v_1 \theta c$, or $v_1 \theta v_2$, where $v_1, v_2 \in \overline{x}$, c is a constant, and $\theta \in \mathbb{B}$. Similarity predicate $\approx$ is positive when the edit distance between two strings is above a user-defined threshold δ.

Single-tuple constraints (such as SQL CHECK constraints), Functional Dependencies, Matching Dependencies, and Conditional Functional Dependencies are special cases of unary and binary denial constraints with equality and similarity predicates.

Given a database instance I of schema $\mathbb{S}$ and a DC φ, if I satisfies φ, we write $I \models \varphi$. If we have a set of DC Σ, $I \models \Sigma$ if and only if $\forall \varphi \in \Sigma, I \models \varphi$. A *repair* I' of an inconsistent instance I is an instance that satisfies Σ and has the same set of tuple identifiers in I. Attribute values of tuples in I and I' can be different and, for infinite domains of attributes in R, there is an infinite number of possible repairs. Similar to [5], [18], we represent the infinite space of repairs as a finite set of instances with fresh attribute values. In a repair, each fresh value FV for attribute A can be replaced with a value from $Dom(A) \setminus Dom^a(A)$, where $Dom^a(A)$ is the domain of the values for A which satisfy at least a predicate for each denial constraints involving FV. In other words, fresh values are values of the domain for the actual attributes which do not satisfy any of the predicates defined over them.

Notice that our setting does not rely on restrictions such as *local constraints* [20] or certain regions [14]: it is possible that a repair for a denial constraint triggers a new violation for another constraint. In order to enforce termination of the cleaning algorithm fresh values are introduced in the repair. More details are discussed in the following sections.

B. Problem Definition

Since the number of possible repairs is usually very large and possibly infinite, it is important to define a criterion to identify desirable ones. In fact, we aim at solving the following *data cleaning problem*: given as input a database I and a set of denial constraints Σ, we compute a *repair* I_r of I such that $I_r \models \Sigma$ (consistency) and their distance *cost*(I_r, I) is minimum (accuracy). A popular *cost* function from the literature [6], [8] is the following:

$$\sum_{t \in I, t' \in I_r, A \in A^R} dis_A(I(t[A]), I(t'[A]))$$

where t' is the repair for tuple t and $dis_A(I(t[A]), I(t'[A]))$ is a distance between their values for attribute A (an exact match returns 0)[1]. There exist many similarity measurements for structured values (such as strings) and our setting does not depend on a particular approach, while for numeric values we rely on the squared Euclidian distance (i.e., the sum of the square of differences). We call this measure of the quality the **Distance Cost**. It has been shown that finding a repair of minimal cost is NP-complete even for FDs only [6]. Moreover,

[1]We omit the confidence in the accuracy of attribute A for tuple t because it is not available in many practical settings. While our algorithms can support confidence, for simplicity we will consider the cells with confidence value equals to one in the rest of the paper, as confidence does not add specific value to our solution.

minimizing the above function for DCs and numerical values only it is known to be a *MaxSNP*-hard problem [4].

Interestingly, if we rely on a binary distance between values (0 if they are equal, 1 otherwise), the above cost function corresponds to aiming at computing the repair with the minimal number of changes. The problem of computing such *cardinality-minimal repairs* is known to be NP-hard to be solved exactly, even in the case with FDs only [18]. We call this quality measure **Cardinality-Minimality Cost**.

Given the intractability of the problems, our goal is to compute nearly-optimal repairs. We rely on two directions to achieve it: approximation holistic algorithms to identify cells that need to be changed, and local exact algorithms within the cells identified by our notion of Repair Context. We detail our solutions in the following sections.

IV. Solution Overview

In this Section, we first present our system architecture, and we explain two data structures: the conflict hypergraph (CH) to encode constraint violations and the repair context (RC) to encode violation repairs.

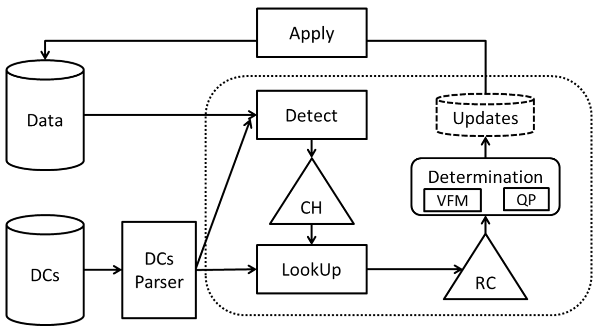


Fig. 2: Architecture of the system.

A. System Architecture

The overall system architecture is depicted in Figure 2. Our system takes as input a relational database (Data) and a set of denial constraints (DCs), which express the data quality rules that have to be enforced over the input database.

Example 4.1: Consider the LocalEmployee table (L for short) in Figure 1. Every tuple represents information stored for an employee of the company in one specific location: employee local id (LID), name (FN, LN), rank (RNK), number of days off (DO), number of years in the company (Y), city (CT), manager id (MID), and salary (SAL). LocalEmployee table and GlobalEmployee table constitute the input database. We introduce a third DC:

$$c_3 : \neg(L(l,f,n,r,d,y,c,m,s), L(l',f',n',r',d',y',c',m',s'),\\ G(g^*,f^*,n^*,r^*,c^*,a^*,s^*), (l \neq l'), (l = m'),\\ (f \approx f^*), (n \approx n^*), (c = c^*), (r^* \neq \text{“}M\text{”}))$$

The constraint states that a manager in the local database L cannot be listed with a status different from "M" in the global database G. The rule shows how different relations, similarity predicate, and self-joins can be used together. ◇

The DCs Parser provides rules for detecting violations (through the Detect module) and rules for fixing the violations to be executed by the LookUp module as we explain in the following example.

Example 4.2: Given the database in Figure 1, the DCs Parser processes constraint c_3 and provides the Detect module the rule to identify a violation spanning ten cells over tuples t_1, t_2, and t_3 as highlighted. Since every cell of this group is a possible error, DCs Parser dictates the LookUp module how to fix the violation if any of the ten cells is considered to be incorrect. For instance, the violation is repaired if "Paul Smith" is not the manager of "Mark White" in L (represented by the repair expression $(l \neq m')$), if the employee in L does not match the one in G because of a different city $(c \neq c^*)$, or if the role for the employee in G is updated to manager $(r^* = M)$. ◇

We described how each DC is parsed so that violations and fixes for that DC can be obtained. However, our goal is to consider violations from all DCs together and generate fixes holistically. For this goal we introduce two data structures: the Conflict Hypergraph (CH), which encodes all violations into a common graph structure, and the Repair Context (RC), which encodes all necessary information of how to fix violations holistically. The Detect module is responsible for building the CH that is then fed into the LookUp module, which in turn is responsible for building the RC. The RC is finally passed to a Determination procedure to generate updates. Depending on the content of the RC, we have two Determination cores, i.e., Value Frequency Map (VFP) and Quadratic Programming (QP). The updates to the database are applied, and the process is restarted until the database is clean (i.e., empty CH), or a termination condition is met.

B. Violations Representation: Conflict Hypergraph

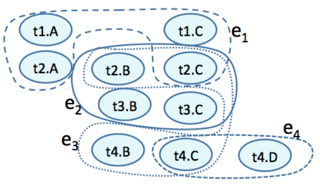
We represent the violations detected by the Detect module in a graph, where the nodes are the violating cells and the edges link cells involved in the same violation. As an edge can cover more than two nodes, we use a *Conflict Hypergraph* (CH) [18]. This is an undirected hypergraph with a set of nodes P representing the cells and a set of annotated hyperedges E representing the relationships among cells violating a constraint. More precisely, a *hyperedge* $(c; p_1, \ldots, p_n)$ is a set of violating cells such that one of them must change to repair the constraint, and contains: (a) the constraint c, which induced the conflict on the cells; (b) the list of nodes $p_1, \ldots, p_n$ involved in the conflict.

Example 4.3: Consider Relation R in Figure 3a and the following constraints (expressed as FDs and CFDs for readability): $\varphi_1 : A \rightarrow C$, $\varphi_2 : B \rightarrow C$, and $\varphi_3 : R[D = 5] \rightarrow R[C = 5]$. CH is built as in Figure 3b: φ_1 has 1 violation e_1; φ_2 has 2 violations e_2, e_3; φ_3 has 1 violation e_4. ◇

The CH represents the current state of the data w.r.t. the constraints. We rely on this representation to analyze the interactions among violations on the actual database. A hyperedge contains only violating cells: in order to repair it,

	A	B	C	D
t_1	**1**	2	**1**	1
t_2	**1**	**1**	**2**	1
t_3	2	**1**	**3**	1
t_4	3	**1**	**3**	**5**

(a) Data



(b) CH

Fig. 3: CH Example.

at least one of its cells must get a new value. Interestingly, we can derive a repair expression for each of the cell involved in a violation, that is, for each variable involved in a predicate of the DC. Given a DC $d : \forall \overline{x} \neg (P_1 \wedge \ldots \wedge P_m)$ and a set of violating cells (hyperedge) for it $V = \{v_1, \ldots, v_n\}$, for each $v_i \in V$ there is at least one *alternative* repair expression of the form $v_i \psi t$, where t is a constant or a connected cell in V. This leads to the following property of hyperedges.

Lemma 4.4: *All the repairs for hyperedge e have at most cardinality n with $n \leq m$, where m is the size of the biggest chain of connected variables among its repair expressions.* $\diamond$

Proof Sketch. We start with the case with one predicate only in the DC d. If it involves a constant, then the repair expression contains only one cell and its size coincides with the size of the hyperedge. If it is a predicate involving another cell, then at least one of them is going to change in the repair. We consider now the case with more than a predicate in d. In this case, as the predicates allowed are binary, there may be a chain of connected variables of size m in the repair expressions: when a value is changed for a cell, it may triggers changes in the connected ones. Therefore, in the worst case, to repair them m changes are needed. $\diamond$

The Lemma states an upper bound for the number of changes that are needed to fix a hyperedge. More importantly, it highlights that in most cases one change suffices as we show in the following example.

Example 4.5: In c_3 the biggest chain of variables in the repair expressions comes from $l = l'$ (from the first predicate) and $l \neq m'$ (from the second predicate). This means that, to repair violations for c_3, at most three changes are needed. Notice that only one change is needed for most of the cells involved in a violation. $\diamond$

A naïve approach to the problem is to compute the repair by fixing the hyperedges one after the other in isolation. This would lead to a valid repair, but, if there are interacting violations, it would certainly change more cells than the repair with minimal cost. As our goal is to minimize changes in the repair, we can rely on hyperedges for identifying cells that are very likely to be changed. The intuition here is that, in the spirit of [18], by using algorithms such as the Minimum Vertex Cover (MVC), we can identify at the global level what are the minimum number of violating cells to be changed in order to compute a repair.[2] For instance, a possible MVC for the CH in Figure 3b identifies $t_2[C]$ and $t_4[C]$.

After detecting all violations in the current database and building the CH, the next step is to generate fixes taking into account the interaction among violations. In order to facilitate a holistic repair, we rely on another data structure, which is discussed next.

C. Fixing Violation Holistically: Repair Context

We start from cells that MVC identifies as likely to be changed, and incrementally identify other cells that are involved in the current repair. We call the starting cells and the newly identified ones *frontier*. We call *repair expressions* the list of constant assignments and constraints among the frontier. The frontier and the repair expressions form a *Repair Context* (RC). We elaborate RC using the following example.

Example 4.6: Consider the database and CH in Example 4.3. Suppose we have $t_2[C]$ and $t_4[C]$ from the MVC as starting points. We start from $t_2[C]$, which is involved in 3 hyperedges. Consider e_1: given $t_2[C]$ to change, the expression $t_2[C] = t_1[C]$ must be satisfied to solve it, thus bringing $t_1[C]$ into frontier. Cell $t_1[C]$ is not involved in other hyperedges, so we stop. Similarly, $t_2[C] = t_3[C]$ must be satisfied to resolve e_2 and $t_3[C]$ is brought into the frontier. For e_3, $t_2[C] = t_4[C]$ is the expression to satisfy, however, $t_4[C]$ is involved also in e_4. We examine e_4 given $t_4[C]$ and we get another expression $t_4[C] = 5$. The resulting RC consists of frontier: $t_1[C], t_2[C], t_3[C], t_4[C]$, and repair expressions: $t_2[C] = t_1[C], t_2[C] = t_3[C], t_2[C] = t_4[C], t_4[C] = 5$.

Notice that by starting from $t_4[C]$ the same repair is obtained and the frontier contains only four cells instead of ten in the connected component of the hypergraph. $\diamond$

An RC is built from a starting cell c with violations from DCs D with a recursive algorithm (detailed in the next section) and has two properties: (i) there is no cell in its frontier that is not (possibly transitively) connected to c by a predicate in the repair expression of at least a $d \in D$, (ii) every cell that is (possibly transitively) connected to c by a predicate in the repair expression of at least a $d \in D$ is in its frontier. In other terms, RC contains exactly the information required by a repair algorithm to make an informed, holistic decision.

Lemma 4.7: *The Repair Context contains the sufficient and necessary information to repair all the cells in its frontier.* $\diamond$

Proof Sketch. We start with the necessity. By definition, the RC contains the union of the repair expressions over the cells in the frontier. If it is possible to find an assignment that satisfies the repair expressions, all the violations are solved. It is evident that, if we remove one expression, then it is not guaranteed that all violations can be satisfied. The repair expressions are sufficient because of the repair semantics of the DCs. As the frontier contains all the connected cells, any other cell from V would add an expression that is not needed to repair the violation for d and would require to change a cell that is not needed for the repair. $\diamond$

[2]In order to keep the execution time acceptable an approximate algorithm is used to compute the MVC.

We can now state the following result regarding the RC:

Proposition 4.8: *An RC always has a repair of cardinality n with $n \leq u$, where u is the size of its frontier.* ◇

Proof Sketch. From Lemmas 4.7 and 4.4 it can be derived that (i) it is always possible to find a repair for RC, and (ii) in the worst case the repair has the size of the union of the chains of the connected variables in its repair expressions. ◇

In practice, the number of cells in a DC is much smaller than the number of cells in the respective hyperedges. For $t_6[CITY]$, the size of the RC is one, while there are nine cells in the two hyperedges for c_1 and c_2.

Given the discussion above, for each cell in the MVC we exploit its violations with the LookUp module to get the RC. Once all the expressions are collected, a *Determination* step takes as input the RC and computes the valid assignments for the cells involved in it. In this step, we rely on a function to minimize the cost of changing strings (VFM) and on an external Quadratic Programming (QP) tool in order to efficiently solve the system of inequalities that may arise when numeric values are involved. The assignments computed in this step become the updates to the original database in order to fix the violations. The following example illustrates the use of QP, while LookUp and Determination processes will be detailed in the next section.

Example 4.9: Consider again the L relation in Figure 1. Two DCs are defined to check the number of extra days off assigned to each employee:

$$c_4 : \neg(L(l, f, n, r, d, y, c, m, s), (r = \text{“}A\text{”}), (d < 3)$$
$$c_5 : \neg(L(l, f, n, r, d, y, c, m, s), (y > 4), (d < 4))$$

In order to minimize the change, the QP formulation of the problem for $t_1[DO]$ is $(x - 2)^2$ with constraints $x \geq 3$ and $x \geq 4$. Value 4 is returned by QP and assigned to $t_1[DO]$. ◇

The holistic reconciliation provided by the RC has several advantages: the cells connected in the RC form a subset of the connected components of the graph and this leads to better efficiency in the computation and better memory management. Moreover, the holistic choice done in the RC minimizes the number of changes for the same cell; instead of trying different possible repairs, an informed choice is made by considering all the constraints on the connected cells. We will see how this leads to better repairs w.r.t. previous approaches.

V. Computing The Repairs

In this Section we give the details of our algorithms. We start by presenting the iterative algorithm that coordinates the detect and repair processes. We then detail the technical solutions we built for Detect, LookUp, and Determination.

A. Iterative Algorithm

Given a database and a set of DCs, we rely on Algorithm 1. It starts by computing violations, the CH, and the MVC over it. These steps bootstrap the outer loop (lines 5–26), which is repeated until the current database is clean (lines 19–22) or a termination condition is met (lines 23–26). Cells in the MVC are ranked in order to favor those involved in more violations

Algorithm 1 `Holistic Repair`

```
Input: Database data, Denial Constraints dcs
Output: Repair data
 1: Compute violations, conflict hypergraph, MVC.
 2: Let processedCells be a set of cells in the database that
    have already been processed.
 3: sizeBefore ← 0
 4: sizeAfter ← 0
 5: repeat
 6:     sizeBefore ← processedCell.$size()$
 7:     mvc ← Re-order the vertices in MVC in a priority
        queue according to the number of hyperedges
 8:     while mvc is not empty do
 9:         cell ← Get one cell from $mvc$
10:         rc ← Initialize a new repair context for that cell
11:         edges ← Get all hyperedges for that cell
12:         while edges is not empty do
13:             edge ← Get an edge from edges
14:             LookUp($cell, edge, rc$)
15:         end while
16:         assignments ← Determination($cell, exps$)
17:         data.$update(assignments)$
18:     end while
19:     reset the graph: re-build hyperedges, get new MVC
20:     if graph has no edges then
21:         return data
22:     end if
23:     tempCells ← graph.$getAllCellsInAllEdges()$
24:     processedCells ← processedCells ∪ tempCells
25:     sizeAfter ← processedCell.$size()$
26: until sizeBefore ≤ sizeAfter
27: return data.$PostProcess(tempCells, MVC)$
```

and are repaired in the inner loop (lines 8–18). In this loop, the RC for the cell is created with the LookUp procedure. When the RC is completed, the Determination step assigns the values to the cells that have a constant assignments in the repair expressions (e.g., $t_1[A] = 5$). Cells that do not have assignments with constants (e.g., $t_1[A] \neq 1$), keep their value and their repair is delayed to the next outer loop iteration. If the updates lead to a new database without violations, then it can be returned as a repair, otherwise the outer loop is executed again. If no new cells have been involved w.r.t. the previous loop, then the termination condition is triggered and the cells without assignments are updated with new fresh values in the post processing final step.

The outer loop has a key role in the repair. In fact, it is possible that an assignment computed in the determination step solves a violation, but raises a new one with values that were not involved in the original CH. This new violation is identified at the end of the inner loop and a new version of the CH is created. This CH has new cells involved in violations and therefore the termination condition is not met.

Before returning the repair, a post-processing step updates all the cells in the last MVC (computed at line 19) to fresh values. This guarantees the consistency of the repair and no new violations can be triggered. Pushing to the very last the assignment of a fresh value forces the outer loop to try to find a repair with constants until the termination condition is met, as we illustrate in the following example.

Example 5.1: Consider again only rules c_1 and c_2 in the running example. After the first inner loop iteration, the RC contains an assignment $t_6[\text{CITY}] \neq$ "NYC", which is not enforced by the determination step and therefore the database does not change. The HC is created again (line 19) and it still has violations for c_1 and c_2. The cells involved in the two violations go into *tempCells* and *sizeAfter* is set to 9. A new outer loop iteration sets *sizeBefore* to 9, the inner loop does not change the data, and it gets again the same graph at line 19. As *sizeBefore = sizeAfter*, it exits the outer loop and the post processing assigns $t_6[\text{CITY}] =$ "FV". $\diamond$

Proposition 5.2: *For every set of DCs, if the determination step is polynomial, then* `Holistic Repair` *is a polynomial time algorithm for the data cleaning problem.* $\diamond$

Proof sketch. It is easy to see that the output of the algorithm is a repair for an input database D with DCs dcs. In the outer loop we change cells to constants that satisfy the violations and in the post process we resolve violations that were not fixable with a constant by introducing fresh values. As fresh values do not match any predicate, the process eventually terminates and returns a repair which does not violate the DCs anymore.

The vertex cover problem is an NP-complete problem and there are standard approaches to find approximate solutions. We use a greedy algorithm with factor k approximation, where k is the maximum number of cells in a hyperedge of the HC. Our experimental studies show that a k approximation of the MVC lead to better results w.r.t. alternative ways to identify the seed cells for the algorithm. The complexity of the greedy algorithm is linear in the number of edges. In the worst case, the number of iterations of the outer loop is bounded by the number of constraints in *dc* plus one: it is possible to design a set of DCs that trigger a new violation at each repair, plus one extra iteration to verify the termination condition. The complexity of the algorithm is bounded by the polynomial time for the detection step: three atoms in the DC need a cubic number of comparisons in order to check all the possible triplets of tuples in the database. In practice, the number of tuples is orders of magnitude bigger than the number of DCs and therefore the size of the data dominates the complexity $O(|data|^c|dcs|)$, where c is the largest number of atoms in a rule of dcs. The complexity of the inner loop depends on the number of edges in the CH and on the complexity of LOOKUP and DETERMINATION that we discuss next. $\diamond$

Though Algorithm 1 is sound, it is not optimal, as it is illustrated in the following example.

Example 5.3: Consider again Example 4.3. We showed a repair with four changes obtained with our algorithm, but there exists a cardinality minimal repair with only three changes: $t_1[C] = 3, t_2[C] = 3, t_4[D] = NV$. $\diamond$

We now describe the functions to generate and manipulate the building blocks of our approach.

predicate in dcs	$=$	$\neq$	$>$	$>=$	$<$	$<=$	$\approx_t$
predicate in repair exps	$\neq$	$=$	$<=$	$<$	$>=$	$>$	$\not\approx_t$

TABLE I: Table of conversion of the predicates in a DC for their repair. Predicate $\not\approx_t$ states that the distance between two strings must be greater than t.

B. DETECT*: Identifying Violations*

Identifying violations is straightforward: every valid assignment for the denial constraint is tested, if all the atoms for an assignment are satisfied, then there is a violation.

However, the detection step is the most expensive operation in the approach as the complexity is polynomial with the number of atoms in the DC as the exponent. For example, in the case of simple pairwise comparisons (such as in FDs), the complexity is quadratic in the number of tuples, and it is cubic for constraints such as c_3 in Example 4.1. This is also exacerbated by the case of similarity comparisons, when, instead of equality check, there is the need to compute edit distances between strings, which is an expensive operation.

In order to improve the execution time on large relations, optimization techniques for matching records [10] are used. In particular, the blocking method partitions the relations into blocks based on discriminating attributes (or blocking keys), such that only tuples in the same block are compared.

C. LOOKUP*: Building the Repair Context*

Given a hyperedge $e = \{c; p_1, \ldots, p_n\}$ and a cell $p = t_i[A_j] \in P$, the repair expression r for p may involve other cells that need to be taken into account when assigning a value to p. In particular, given e and p, we can define a rule for the generation of repair expressions.

As $p \in A_\phi(c)$, then it is required that $r : p\phi^c c$, where ϕ^c is the predicate converted as described in Table I. Variable c can be a constant or another cell. For denial constraints, we defined a function *DC.Repair(e,c)*, based on the above rule, which automatically generates a repair expression for a hyperedge e and a cell c. We first show an example of its output when constants are involved in the predicate and then we discuss the case with variables.

Example 5.4: Consider the constraint c_2 from the example in Figure 1. We show below two examples of repair expressions for it.

$$\begin{aligned} &\textit{DC.Repair}((c_2; t_5[\text{ROLE}], t_5[\text{CITY}], t_5[\text{SAL}], \ldots, t_6[\text{SAL}]),\\ &\qquad t_5[\text{ROLE}]) = \{t_5[\text{ROLE}] \neq \text{“}E\text{”}\}\\ &\textit{DC.Repair}((c_2; t_5[\text{ROLE}], t_5[\text{CITY}], t_5[\text{SAL}], \ldots, t_6[\text{SAL}]),\\ &\qquad t_6[\text{SAL}]) = \{t_6[\text{SAL}] \geq 80\} \end{aligned}$$

In the first repair expression the new value for $t_5[\text{ROLE}]$ must be different from "E" to solve the violation. The repair expression does not state that its new value should be different from the active domain of ROLE (i.e., $t_5[\text{ROLE}] \neq$ {"E","S","M"}), because in the next iteration of the outer loop it is possible that another repair expression imposes

t_5[ROLE] to be equal to a constant already in the active domain (e.g., a MD used for entity resolution). If there is no other expression suggesting values for t_5[ROLE], in a following step the termination condition will be reached and the post-process will assign a fresh value to t_5[ROLE]. ◇

Given a cell to be repaired, every time another variable is involved in a predicate, at least another cell is involved in the determination of its new value. As these cells must be taken into account, we also collect their expressions, thus possibly triggering the inclusion of new cells. We call LOOKUP the recursive exploration of the cells involved in a decision.

Algorithm 2 LOOKUP

```
Input: Cell cell, Hyperedge edge, Repair Context rc
Output: updated rc
1: exps ← $Denial.repair(edge, cell)$
2: frontier ← $exps.getFrontier()$
3: for all cell ∈ frontier do
4:    edges ← $cell.getEdges()$
5:    for all edge ∈ edges do
6:       exps ← exps ∪ LOOKUP(cell,edge,rc).$getExps()$
7:    end for
8: end for
9: rc.$update(exps)$
```

Algorithm 2 describes how, given a cell c and a hyperedge e, LOOKUP processes recursively in order to move from a single repair expression for c to a *Repair Context.*

Proposition 5.5: *For every set of DCs and cell c,* LOOKUP *always terminates in linear time and returns a Repair Context for c.* ◇

Proof sketch. The correctness of the RC follows from the traversal of the entire graph. Cycles are avoided as in the expressions the algorithm keeps track of previously visited nodes. As it is a Depth-first search, its complexity is linear in the size of graph and is $O(2V-1)$, where V is the largest number of connected cells in an RC. ◇

Example 5.6: Consider the constraint c_3 from the example in Figure 1 and the DC:

$$c_6 : \neg(G(g, f, n, r, c, a, s), (r = \text{“}V\text{”}), (s < 200))$$

That is, a vice-president cannot earn less than 200. Given t_3[ROLE] as input, LOOKUP processes the two edges over it and collects the repair expressions $t_3[\text{ROLE}] \neq \text{“}V\text{”}$ from c_6 and $t_3[\text{ROLE}] = \text{“}M\text{”}$ from c_3. ◇

D. DETERMINATION*: Finding Valid Assignments*

Given the set of repair expressions collected in the RC, the DETERMINATION function returns an assignment for the frontier in the RC. The process for the determination is depicted in Algorithm 3: Given an RC and a starting cell, we first choose a (maximal) subset of the repair expressions that is satisfiable, then we compute the value for the cells in the frontier aiming at minimizing the cost function, and update the database accordingly later.

Algorithm 3 DETERMINATION

```
Input: Cell cell, Repair Context rc
Output: Assignments assigns
1: exps ← $rc.getExps()$
2: if exps contain $>, <, >=, <=$ then
3:    $QP$ ← computeSatisfiable(exps)
4:    assigns ← $QP.getAssigments()$
5: else
6:    $VFM$ ← computeSatisfiable(exps)
7:    assigns ← $VFM.getAssigments()$
8: end if
9: return assigns
```

In Algorithm 3, we have two determination procedures. One is Value Frequency Map (VFM), which deals with string typed expressions. The other is quadratic programming (QP), which deals with numerical typed expressions[3].

1) Function computeSatisfiable: Given the current set of expressions in the context, this function identifies the subset of expressions that are solvable.

Some edges may be needed to be removed from the RC to make it solvable. First, a satisfiability test verifies if the repair expressions are in contradiction. If the set is not satisfiable, the repair expressions coming from the hyperedge with the smallest number of cells are removed. If the set of expressions is now satisfiable, the removed hyperedge is pushed to the outer loop in the main algorithm for repair. Otherwise, the original set minus the next hyperedge is tested. The process of excluding hyperedges is then repeated for pairs, triples, and so on, until a satisfiable set of expressions is identified. In the worst case, the function is exponential in the number of edges in the current repair context. The following example illustrates how the function works.

Example 5.7: Consider the example in Figure 1 and two new DCs:

$$c_7 : \quad \neg(L(l, f, n, r, d, y, c, m, s), (r = \text{“}B\text{”}), (d > 4))$$
$$c_8 : \quad \neg(L(l, f, n, r, d, y, c, m, s), (y > 7), (d < 6))$$

That is, an employee after 8 years should have at least 6 extra days off, and an employee of rank “B” cannot have more than 4 days. Given t_2[DO] as input by the MVC, LOOKUP processes the two edges over it and collects the repair expressions $t_2[\text{DO}] \leq 4$ from c_7 and $t_2[\text{DO}] \geq 6$ from c_8. The satisfiability test fails ($x \leq 4 \wedge x \geq 6$) and the *computeSatisfiable* function starts removing expressions from the RC, in order to maximize the set of satisfiable constraints. In this case, it removes c_7 from the RC and sets $t_2[\text{DO}] = 6$ to satisfy c_8. Violation for c_7 is pushed to the outer loop, and, as in the new MVC there are no new cells involved, the post processing step updates t_2[RNK] to a fresh value. ◇

2) Function getAssignments: After getting the maximum number of solvable expressions, the following step aims at computing an optimal repair according to the cost model at

[3]We assume all numerical values to be integer for simplicity

hand. We therefore distinguish between string typed expressions and numerical typed expressions for both cost models: cardinality minimality and distance minimality

String Cardinality Minimality. In this case we want to minimize the number of cells to change. For string type, expressions consist only of $=$ and $\neq$, thus we create a mapping from each candidate value to the occurrence frequency (VFM). The value with biggest frequency count will be chosen.

Example 5.8: Consider a schema $R(A, B)$ with 5 tuples $t_1 = R(a, b), t_2 = R(a, b), t_3 = R(a, cde), t_4 = R(a, cdf), t_5 = R(a, cdg)$. R has an $FD : A \rightarrow B$. Suppose now we have an RC with set of expressions $t_1[B] = t_2[B] = t_3[B] = t_4[B] = t_5[B]$. VFM is created with $b \rightarrow 2, cde \rightarrow 1, cdf \rightarrow 1, cdg \rightarrow 1$. So value b is chosen. $\diamond$

String Distance Minimality. In this case we want to minimize the string edit distance. Thus we need a different VFM, which maps from each candidate value to the edit distance if this value were to be chosen.

Example 5.9: Consider the same database as Example 5.8. String cardinality minimality is not necessarily string distance minimality. Now VFM is created as follows: $b \rightarrow 12, cde \rightarrow 10, cdf \rightarrow 10, cdg \rightarrow 10$. So any of cde, cdf, cdg can be chosen. $\diamond$

Numerical Distance Minimality. In this case we want to minimize the squared distance. QP is our determination core. In particular, we need to solve the following objective function: for each cell with value v involved in a predicate of the DC, a variable x is added to the function with $(x - v)^2$. The expressions in the RC are transformed into constraints for the problem by using the same variable of the function. As the objective function given as a quadratic has a positive definite matrix, the quadratic program is efficiently solvable [19].

Example 5.10: Consider a schema $R(A, B, C)$ with a tuple $t_1 = R(0, 3, 2)$ and the two repair expressions: $r_1 : R[A] < R[B]$ and $r_2 : R[B] < R[C]$. To find valid assignments, we want to minimize the quadratic objective function $(x - 0)^2 + (y - 3)^2 + (z - 2)^2$ with two linear constraints $x < y$ and $y < z$, where x, y, z will be new values for $t_1[A], t_1[B], t_1[C]$. We get solution $x = 1, y = 2, z = 3$ with the value of objective function being 3. $\diamond$

Numerical Cardinality Minimality. In this case we want (i) to minimize the number of changed cells, and (ii) to minimize the distance for those changing cells. In order to achieve cardinality minimality for numerical values, we gradually increase the number of cells that can be changed until QP becomes solvable. For those variables we decide not to change, we add constraint to enforce it to be equal to original values. It can be seen that this process is exponential in the number of cells in the RC.

Example 5.11: Consider the same database as in Example 5.10.Numerical distance minimality is not necessary numerical cardinality minimum. It can be easily spotted that $x = 0, y = 1, z = 2$ whose squared distance is 4 only has one change, while $x = 1, y = 2, z = 3$ whose squared is 3 has three changes. $\diamond$

VI. Optimizations and Extensions

In this section, we briefly discuss two optimization techniques adopted in our system, followed by two possible extensions that may be of interest to certain application scenarios.

Detection Optimization. Violation detection for DCs checks every possible grounding of predicates in denial constraints. Thus improving the execution times for violation detection implies reducing the number of groundings to be checked. We face the issue by verifying predicates in a order based on their selectivity. Before enumerating all grounding combinations, predicates with constants are applied first to rule out impossible groundings. Then, if there is an equality predicate without constants, the database is partitioned according to two attributes in the equality predicate, so that grounding from two different partitions need not to be checked. Consider for example c_3. The predicate $(r^* \neq 'M')$ is applied first to rule out grounding with attribute r^* equals M. Then predicate $(l = m')$ is chosen to partition the database, so groundings with values of attributes l and m' not being in the same partition will not be checked.

Hypergraph Compression. The conflict hypergraph provides a violation representation mechanism, such that all information necessary for repairing can be collected by the LookUp module. Thus, the size of the hypergraph has an impact on the execution time of the algorithm. We therefore reduce the number of hyperedges without compromising the repair context by removing redundant edges. Consider for example a table $T(A, B)$ with 3 tuples $t1 = (a1, b1), t2 = (a1, b2), t3 = (a1, b3)$ and an FD: $A \rightarrow B$; it has three hyperedges and three expressions in the repair context, i.e., $t_1[B] = t_2[B], t_1[B] = t_3[B], t_2[B] = t_3[B]$. However, only two of them are necessary, because the expression for the third hyperedge can be deduced from the first two.

Custom Repair Strategy. The default repair strategy can easily be personalized with a user interface for the LookUp module. For example, if a user wants to enforce the increase of the salary for the NYC employee in rule c_2, she just needs to select the s variable in the rule. An alternative representation of the rule can be provided by sampling the rule with an instance on the actual data, for example $\neg(G(386, Mark, Lee, E, NYC, 552, AZ, 75)$, $G(Mark, White, E, SJ, 639, CA, 80), (80 > 75))$, and the user highlights the value to be changed in order to repair the violation.

We have shown how repair expressions can be obtained automatically for DCs. In general, the *Repair* function can be provided for any new kind of constraints that is plugged to the system. In case the function is not provided, the system would only detect violating cells with the Detect module. The iterative algorithm will try to fix the violation with repair expressions from other interacting constraints or, if it is not possible, it will delay its repair until the post-processing step.

Manual Determination. In certain applications, users may want to manually assign values to dirty cells. In general, if a user wants to verify the value proposed by the system for a repair, and eventually change it, she needs to analyze what are the cells involved in a violation. In this scenario, the RC can expose exactly the cells that need to be evaluated by the user in the manual determination. Even more importantly, the RC contains all the information (such as constants assignments and expressions over variables) that lead to the repair. In the same fashion, fresh values added in the post processing step can be exposed to the user with their RC for examination and manual determination.

VII. Experimental Study

The techniques have been implemented as part of the NADEEF data cleaning project at QCRI[4] and we now present experiments to show their performance. We used real-world and synthetic data to evaluate our solution compared to state-of-the-art approaches in terms of both effectiveness and scalability.

A. Experimental Settings

Datasets. In order to compare our solution to other approaches we selected three datasets.

The first one, HOSP, is from US Department of Health & Human Services [1]. HOSP has 100K tuples with 19 attributes and we designed 9 FDs for it. The second one, CLIENT [4], has 100K tuples, 6 attributes over 2 relations, and 2 DCs involving numerical values. The third one, EMP, contains synthetic data and follows the structure of the running example depicted in Figure 1. We generated up to 100K tuples for the 17 attributes over 2 relations. The DCs are $c_1, \ldots, c_6$ as presented in the paper.

Errors in the datasets have been produced by introducing noise with a certain rate, that is, the ratio of the number of dirty cells to the total number of cells in the dataset. An error rate $e\%$ indicates that for each cell, there is a $e\%$ probability we are going to change that cell. In particular, we update the cells containing strings by randomly picking a character in the string, and change it to "X", while cells with numerical values are updated with randomly changing a value from an interval.[5]

Algorithms. The techniques presented in the paper have been implemented in Java. As our holistic Algorithm 1 is modular with respect to the cost function that the user wants to minimize, we implemented the two semantics discussed in Section V-D. In particular we tested the *getAssigment* function both for cardinality minimality (*RC-C*) and for the minimization of the distance (*RC-D*).

We implemented also the following algorithms in Java: the FD repair algorithms from [5] (*Sample*), [6] (*Greedy*), [18] (*VC*) for HOSP; and the DC repair algorithm from [4] (*MWSC*) for CLIENT. As there is no available algorithm able to repair all the DCs in EMP, we compare our approach against a sequence of applications of other algorithms (*Sequence*). In particular, we ran a combination of three algorithms: *Greedy* for DCs c_1, *MWSC* for c_2, c_4, c_5, c_6, and a simple, ad-hoc algorithm to repair c_3 as it is not supported by any of the existing algorithms. In particular, for c_3 we implemented a simplified version of our Algorithm 1, without MVC and with violations fixed one after the other without looking at their interactions. As there are six alternative orderings, we executed all of them for each test and picked the results from the combination with the best performance. For $\approx_t$ we used string edit distance with $t = 3$: two strings were considered similar if the minimum number of single-character insertions, deletions and substitutions needed to convert a string into the other was smaller than 4.

Metrics. We measure performance with different metrics, depending on the constraints involved in the scenario and on the cost model at hand. The number of changes in the repair is the most natural measure for cardinality minimality, while we use the cost function in Section III-B to measure the distance between the original instance and its repair. Moreover, as the ground truth for these datasets is available, to get a better insight about repair quality we measured also precision (P, corrected changes in the repair), recall (R, coverage of the errors introduced with $e\%$), and F-measure ($F = 2 \times (P \times R)\ (P + R)$). Finally, we measure the execution times needed to obtain a repair.

As in [5], we count as *correct changes* the values in the repair that coincide with the values in the ground truth, but we count as a fraction (0.5) the number of *partially correct changes*: changes in the repair which fix dirty values, but their updates do not reflect the values in the ground truth. It is evident that fresh values will always be part of the partially correct changes.

All experiments were conducted on a Win7 machine with a 3.4GHz Intel CPU and 4GB of RAM. Gurobi Optimizer 5.0 has been used as the external QP tool [16] and all computations were executed in memory. Each experiment was run 6 times, and the results for the best execution are reported. We decided to pick the best results instead of the average in order to favor *Sample*, which is based on a sampling of the possible repairs and has no guarantee that the best repair is computed first.

B. Experimental Results

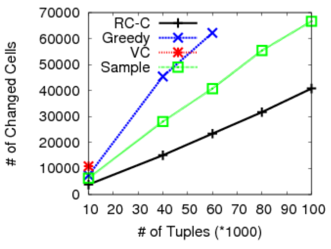
We start by discussing repair quality and scalability for each dataset. Depending on the constraints in the dataset, we were able to use at least two alternative approaches. We then show how the algorithms can handle a large number of constraints holistically. Finally, we show the impact of the MVC on our repairs.

Exp-1: FDs only. In the first set of experiments we show that the holistic approach has benefits even when the constraints are all of the same kind, in this case FDs. As in this example all the alternative approaches consider some kind of cardinality minimality as a goal, we ran our algorithm with the *getAssigment* function set for cardinality minimality (*RC-C*).

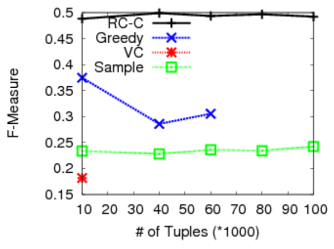
Figures 4(a-c) report results on the quality of the repairs generated for the HOSP data with four systems. Our system clearly outperforms all alternatives in every quality measure. This verifies that holistic repairs are more accurate than alternative fixes. The low values for the F-measure are expected: even if the precision is very high (about 0.9 for our approach

[4]http://da.qcri.org/NADEEF/

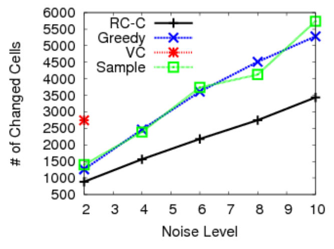
[5]Datasets can be downloaded at http://da.qcri.org/hc/data.zip



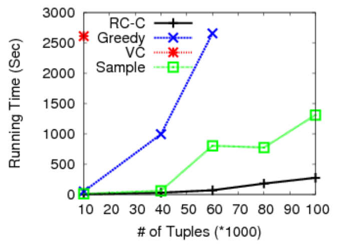
(a) HOSP # of changes



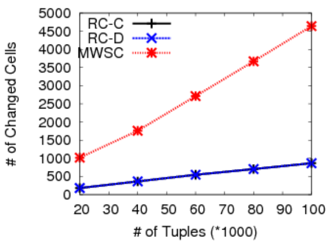
(b) HOSP F-measure



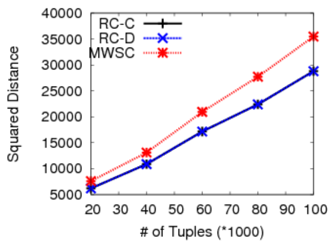
(c) HOSP % errors



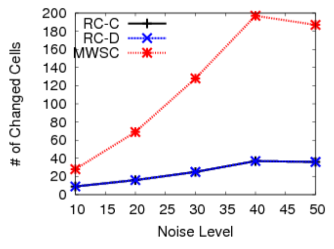
(d) HOSP Exec. time



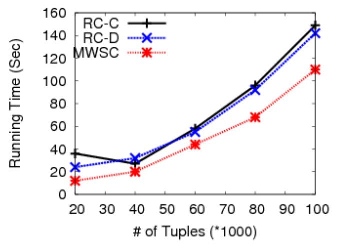
(e) CLIENT # of changes



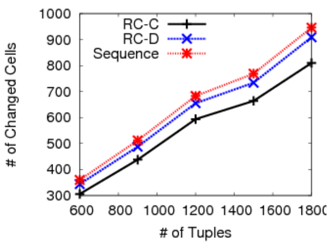
(f) CLIENT Distance



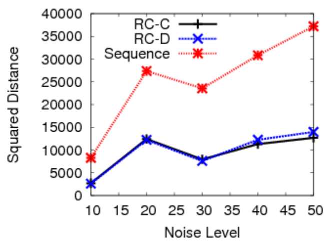
(g) CLIENT % errors



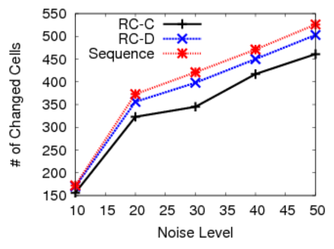
(h) CLIENT Exec. time



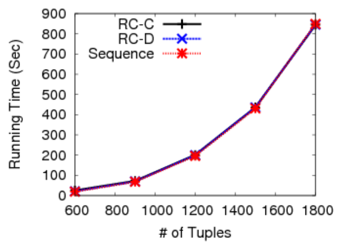
(i) EMP # of changes



(j) EMP Distance



(k) EMP % errors



(l) EMP Exec. time

Fig. 4: Experimental results for the data cleaning problem.

on 5% error rate), recall is always low because many randomly introduced error cannot be detected. Consider for example R(A,B), with an FD: $A \to B$, and two tuples R(1,2), R(1,3). An error introduced for a value in A does not trigger a violation, as there is not match in the left hand side of the FD, thus the erroneous value cannot be repaired.

Figure 4(c) shows the number of cells changed to repair input instances (of size 10K tuples) with increasing amounts of errors. The number of errors increases when $e\%$ increases for all approaches; however, *RC-C* benefits of the holism among the violations and is less sensitive to this parameter.

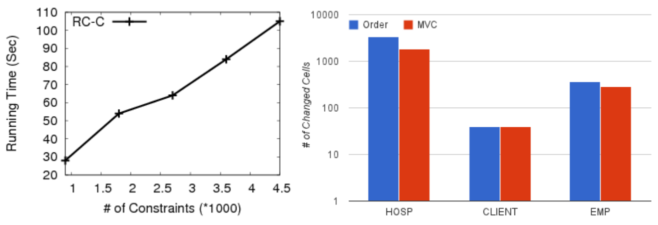
Execution times are reported in Figure 4(d), we set a timeout of 10 minutes and do not report executions over this limit. We can notice that our solution competes with the fastest algorithm and scales nicely up to large databases. We can also notice that *VC* does not scale to large instances due to the large size of their hypergraph, while our optimizations effectively reduces the number of hyperedges in *RC-C*.

Exp-2: DCs with numerical values. In the experiment for CLIENT data, we compare our solution against the state-of-the-art for the repair of DCs with numerical values (*MWSC*) [4]. As *MWSC* aims at minimizing the distance in the repair, we ran the two versions of our algorithm (*RC-C* and *RC-D*).

Figures 4(e-f) show that *RC-C* and *RC-D* provide more precise repairs, both in terms of number of changes and distance, respectively. As in Exp-1, the holistic approach shows significant improvements over the state-of-the-art even with constraints of the same kind only, especially in terms of cardinality minimality. This can be observed also with data with increasing amount of errors in Figures 4(g). Notice that *RC-C* and *RC-D* have very similar performances for this example. This is due to the fact that the dataset was designed for *MWSC*, which supports only local DCs. For this special class the cardinality minimization heuristic is not needed in order to obtain minimality. However, Figure 4(g) shows that the overhead in execution time for *RC-C* is really small and the execution times for our algorithms is comparable to *MWSC*.

Exp-3: Heterogeneous DCs. In the experiments for the EMP dataset, we compare *RC-C* and *RC-D* against *Sequence*. In this dataset we have more complex DCs and, as expected, Figures 4(i) and 4(k) show that *RC-C* performs best in terms of cardinality minimality. Figure 4(j) reports that both *RC-C* and *RC-D* perform significantly better than *Sequence* in terms of Distance cost. We observe that all approaches had low precision in this experiment: this is expected when numerical values are involved, as it is very difficult for an algorithm to repair a violation with exactly the correct value. Imagine an example with value x violating $x > 200$ and an original, correct value equals to 250; in order to minimize the distance from the input, value x is assigned 201 and there is a significant distance w.r.t. the true value.

Execution times in Figure 4(l) show that the three algorithms have the same time performances. This is not surprising, as they share the detection of the violations which is by far the most expensive operation due to the presence of a constraint with three atoms (c_3). The cubic complexity for the detection of the violations clearly dominates the computation. Techniques to improve the performances for the detection problem are out of the scope of this work and are currently under study in the context of parallel computation on distributed infrastructures [17].



(a) HOSP # of DCs (b) MVC vs Order (log. scale)

Fig. 5: Results varying the number of constraints and the ordering criteria in Algorithm 1.

Exp-4: Number of Rules. In order to test the scalability of our approach w.r.t. the number of constraints, we generated DCs for the HOSP dataset and tested the performance of the system. New rules have been generated as follows: randomly take one FD c from the original constraints for HOSP, one of its tuples t from the ground truth, and create a CFD c', such that all the attributes in c must coincide with the values in t (e.g., c' : Hosp[Provider#=10018] $\rightarrow$ Hosp[Hospital="C. E. FOUNDATION"]). We then generated an instance of 5k tuples with 5% error rate and computed a repair for every new set of DCs. For each execution, we increased the number of constraints as input. The results in Figure 5a verifies that the execution times increase linearly with the number of constraints.

Exp-5: MVC contribution. In order to show the benefits of MVC on the quality of repair, we compared the use of MVC to identify conflicting cells versus a simple ordering based on the number of violations a cell is involved (Order). For the experiment we used datasets with 10k tuples, 5% error rate and *RC-C*. Results are reported in Figure 5b. For the hospital dataset the number of changes is almost the double with the simple ordering (3382 vs 1833), while the difference is smaller for the other two experiments because they show fewer interactions between violations.

VIII. Conclusions and Future Work

Existing systems for data quality handle several formalisms for quality rules, but do not combine heterogeneous rules neither in the detection nor in their repair process. In this work we have shown that our approach to holistic repair improves the quality of the cleaned database w.r.t. the same database treated with a combination of existing techniques.

Datasets used in the experimental evaluation fit in the main memory, but, in case of larger databases, it may be needed to put the hypergraph in secondary memory and revise the algorithms to make scale in the new setting. This is a technical extension of our work that will be subject of future studies. Another subject worth of future study is how to automatically derive denial constraints from data, similarly to what has been done for other quality rules [7], [15], since experts designed constraints are not always readily available.

Finally, Repair Context can encode any constraint defined over constants and variables, thus opening a prospective beyond binary predicates. We believe that by enabling mathematical expressions and aggregates in the constraints we can make a step forward the goal of bridging the gap between the procedural business rules, used in the enterprise settings, and the declarative constraints studied in the research community.

References

[1] http://www.hospitalcompare.hhs.gov/, 2012.
[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2), 1993.
[3] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.
[4] L. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko. Complexity and approximation of fixing numerical attributes in databases under integrity constraints. In *DBPL*, 2005.
[5] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1), 2010.
[6] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.
[7] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1), 2008.
[8] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.
[9] W. W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. The Data Warehousing Institute, 2002.
[10] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1), 2007.
[11] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.
[12] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2(1), 2009.
[13] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD Conference*, 2011.
[14] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2), 2012.
[15] L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1), 2008.
[16] Gurobi. Gurobi optimizer reference manual, 2012.
[17] T. Kirsten, L. Kolb, M. Hartung, A. Gross, H. Köpcke, and E. Rahm. Data partitioning for parallel entity matching. In *QDB*, 2010.
[18] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.
[19] M. Kozlov, S. Tarasov, and L. Khachiyan. Polynomial solvability of convex quadratic programming. *USSR Computational Mathematics and Mathematical Physics*, 20(5), 1980.
[20] A. Lopatenko and L. Bravo. Efficient approximation algorithms for repairing inconsistent databases. In *ICDE*, pages 216–225, 2007.
[21] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5), 2011.

The LLUNATIC Data-Cleaning Framework

Floris Geerts[1] Giansalvatore Mecca[2] Paolo Papotti[3] Donatello Santoro[2,4]

[1] University of Antwerp – Antwerp, Belgium [2] Università della Basilicata – Potenza, Italy
[3] Qatar Computing Research Institute (QCRI) – Doha, Qatar [4] Università Roma Tre – Roma, Italy

ABSTRACT

Data-cleaning (or data-repairing) is considered a crucial problem in many database-related tasks. It consists in making a database consistent with respect to a set of given constraints. In recent years, repairing methods have been proposed for several classes of constraints. However, these methods rely on ad hoc decisions and tend to hard-code the strategy to repair conflicting values. As a consequence, there is currently no general algorithm to solve database repairing problems that involve different kinds of constraints and different strategies to select preferred values. In this paper we develop a uniform framework to solve this problem. We propose a new semantics for repairs, and a chase-based algorithm to compute minimal solutions. We implemented the framework in a DBMS-based prototype, and we report experimental results that confirm its good scalability and superior quality in computing repairs.

1. INTRODUCTION

In the constraint-based approach to data quality, a database is said to be dirty if it contains inconsistencies with respect to some set of constraints. The *data-cleaning* (or *data-repairing*) process consists in removing these inconsistencies in order to clean the database. It represents a crucial activity in many real-life information systems as unclean data often incurs economic loss and erroneous decisions [15].

Data cleaning is a long-standing research issue in the database community. Focusing on recent years, many interesting proposals have been put forward, all with the goal of handling the many facets of the data-cleaning process.

– A plenitude of constraint languages has been devised to capture various aspects of dirty data as inconsistencies of constraints. These constraint languages range from standard database dependency languages such as functional dependencies and inclusion dependencies [1], to conditional functional dependencies [16] and conditional inclusion dependencies [15], to matching dependencies [14] and editing-rules [18], among others. Each of these languages allows to capture different forms of dirtiness in data.

– Various repairing strategies have been proposed for these constraint languages. One of the distinguishing features of these strategies is how they use the constraints to modify the dirty data by changing values into "*preferred*" values. Preferred values can be found from, e.g., master data [24], tuple-certainty and value-accuracy [19], freshness and currency [17], just to name a few.

– Repairing strategies also differ in the kind of repairs that they compute. Since the computation of all possible repairs is infeasible in practice, conditions are imposed on the computed repairs to restrict the search space. These conditions include, e.g., various notions of (cost-based) minimality [7, 8, 10] and certain fixes [18]. Alternatively, sampling techniques are put in place to randomly select repairs [7].

It is thus safe to say that there is already a good arsenal of approaches and techniques for data cleaning at our disposal. In this paper, we want to capitalize on this wealth of knowledge about the subject, and investigate the following foundational problem: *what happens to the data administrator facing a complex data-cleaning problem that requires to bring together several of the techniques discussed above?* This problem is illustrated in the following example.

Example 1: Consider the database shown in Fig. 1 consisting of customer data (CUSTOMERS), with their addresses and credit-card numbers, and medical treatments paid by insurance plans (TREATMENTS). We refer to these two tables as the *target database* to be cleaned. As is common in corporate information systems [24], an additional *master-data table* is available; this table contains highly-curated records whose values have high accuracy and are assumed to be clean. In our approach, master data is referred to as the *source database*, since it is a source of reliable clean data.

CUSTOMERS

	SSN	NAME	PHONE	CONF	STR	CITY	CC#
t_1	111	M. White	408-3334	*0.8*	Red Ave.	NY	112321
t_2	222	L. Lennon	122-1876	*0.9*	NULL	SF	781658
t_3	222	L. Lennon	000-0000	*0.0*	Fry Dr.	SF	784659

TREATMENTS

	SSN	SALARY	INSUR.	TREAT	DATE
t_4	111	10K	Abx	Dental	10/1/2011
t_5	111	25K	Abx	Cholest.	8/12/2012
t_6	222	30K	Med	Eye surg.	6/10/2012

MASTER DATA *(Source Table)*

	SSN	NAME	PHONE	STR	CITY
t_m	222	F. Lennon	122-1876	Sky Dr.	SF

Figure 1: Customers, Treatments and Master Data.

We first illustrate the problem of specifying a set of constraints under which the target database is regarded to be clean, as follows:

$(a1)$ Standard functional dependencies (FD): $d_1 = (\text{SSN}, \text{NAME} \rightarrow \text{PHONE})$ and $d_2 = (\text{SSN}, \text{NAME} \rightarrow \text{CC\#})$ on table CUSTOMERS. The pair of tuples $\{t_2, t_3\}$ in the target database violates both d_1 and d_2; the database is thus dirty.

$(a2)$ A conditional FD (CFD): $d_3 = (\text{INSUR}[\text{Abx}] \rightarrow \text{TREAT}[\text{Dental}])$ on table TREATMENTS, expressing that insurance company 'Abx' only offers dental treatments ('Dental'). Tuple t_5 violates d_3, adding more dirtiness to the target database.

$(a3)$ A master-data based editing rule (eR), d_4, stating that whenever a tuple t in CUSTOMERS agrees on the SSN and PHONE attributes with some master-data tuple t_m, then the tuple t must take its NAME, STR, CITY attribute values from t_m. Tuple t_2 does not adhere to this rule.

$(a4)$ An inter-table CFD d_5 between TREATMENTS and CUSTOMERS, stating that the insurance company 'Abx' only accepts customers who reside in San Francisco (SF). Tuple pairs $\{t_1, t_4\}$ and tuples $\{t_1, t_5\}$ violate this constraint.

With the dirty target database at hand, we are faced with the problem of repairing it. The main problem is to identify and select "preferred values" as modifications to repair the data.

$(b1)$ Consider FD d_1. To repair the target database one may want to equate $t_2[\text{PHONE}]$ and $t_3[\text{PHONE}]$. The FD does not tell, however, to which phone number these attribute values should be repaired: '122-1876' or '000-0000', or even a completely different value. As it happens in this kind of problems, we assume that the PHONE attribute values in the CUSTOMERS table come with a *confidence* (*Conf.*) value. If we assume that one prefers values with higher confidence, we can repair $t_3[\text{PHONE}]$ by changing it to '122-1876'.

$(b2)$ Similarly, when working with the TREATMENTS table, we may use dates of treatments to infer the currency of other attributes. If the target database is required to store the most recent value for the salary by FD $d_6 = (\text{SSN} \rightarrow \text{SALARY})$, this may lead us to repair the obsolete salary value '10K' in t_4 with the more recent (and preferred) value '25K' in t_5.

$(b3)$ Notice that we don't always have a clear policy to choose preferred values. For example, when repairing t_2[CC#] and t_3[CC#] for FD d_2, there is no information available to resolve the conflict. This means that the best we can do is to "mark" the conflict, and then, perhaps, ask for user-interaction in order to solve it.

Another crucial aspect that complicates matters is the interaction between dependencies: repairing them in different orders may generate different repairs.

$(c1)$ Consider dependencies d_1 and d_4. As discussed above, we can use d_1 to repair tuples t_2, t_3 such that both have phone-number '122-1876'; then, since t_2 and t_3 agree with the master-data tuple t_m, we can use d_4 to fix names, streets and cities, to obtain: (222, F. Lennon, 122-1876, Sky Dr., SF, 781658), for t_2, and (222, F. Lennon, 122-1876, Sky Dr., SF, 784659), for t_3. However, if, on the contrary, we apply d_4 first, only t_2 can be repaired as before; then, since t_2 and t_3 do not share the same name anymore, d_1 has no violations. We thus get a different result, of inferior quality.

A first, striking observation about our example is that, despite many studies on the subject, there is currently no way to handle this kind of scenarios. This is due to several strong limitations of the known techniques.

Problem 1: Missing Semantics First, although repairing strategies exist for each of the individual classes of constraints discussed at items $(a1)$, $(a2)$ and $(a3)$, there is currently no formal semantics for their combination. In fact, the interactions shown in $(c1)$ require a uniform treatment of the repairing process and a clear definition of what is a repair. Aside from the generic notion of a repair as an updated database that satisfies the constraints, it is not possible to say what represents a "good" repair in this case.

Problem 2: Missing Repair Algorithms Since there is no semantics, we have no algorithms at our disposal to compute repairs. Notice that combining the repairing algorithms available for each of the constraints in isolation does not really help, since repairing a constraint of one type may break one of a different type. Also, current algorithms tend to hard-code the way in which preferred values are used for the purpose of repairing the database. As a consequence, there is no way to incorporate the different strategies illustrated in $(b1)$ and $(b2)$ into existing repairing algorithms in a principled way.

Problem 3: Main-Memory Implementations and Scalability Third, even if we were able to devise a reasonable semantics for this kind of scenarios, we would still face a paramount problem, i.e., computing solutions in a scalable way despite the high complexity of the problem. Computing repairs requires to explore a space of solutions of exponential size wrt the size of the database. In fact, previous proposals have mainly adopted main-memory implementations to speed-up the computation, with a rather limited scalability (in the order of the tens of thousands of tuples).

Contributions The main contribution of this paper consists in developing a uniform framework for data-cleaning problems that solves the issues discussed above. More specifically:

(i) We introduce a language to specify constraints based on *equality generating dependencies (egds)* [4] that generalizes many of the constraints used in the literature. This standardizes the way to express dependencies, and extends them to express inter-table constraints, with several benefits in terms of scalability, as discussed in our experiments.

(ii) The core contribution of the paper consists in the definition of a novel semantics for the data-cleaning problem. The definition of such a semantics is far from trivial, since our goal is to formalize the process of cleaning an instance as the process of *upgrading* its quality, regardless of the specific notions of value preference adopted in a given scenario. Our semantics builds on two main concepts. First, we show that seeing repairs simply as cell updates is not sufficient. On the contrary, we introduce the new notion of a *cell group*, that is essentially a "partial repair with lineage"; then, we formalize the notion of an upgrade by introducing a very general notion of a *partial order* over cell groups; the partial order nicely abstracts all of the most typical strategies to decide when a value should be preferred to another, including master data, certainty, accuracy, freshness and currency. In the paper, we show how users can easily plug-in their preference strategies for a given scenario into the semantics. Finally, by introducing a new category of values, called *lluns*, we are able to complete the lattice of instances induced by the partial order, and to provide a natural hook for incorporating user feedbacks into the process.

(iii) We introduce the notion of a *minimal solution* and develop algorithms to compute minimal solutions, based on a parallel-chase procedure. The definition of the chase is far from trivial, since our goal is to guarantee both generality and proper scalability. To start, we chase violations not at tuple level, but at *equivalence-class level* [8]. This allows us to introduce a notion of a *cost manager* as a plug-in for the chase algorithm that selects which repairs should be kept and which ones should be discarded. The cost manager abstracts and generalizes all of the popular solution-selection strategies, including similarity-based cost, set-minimality, set-cardinality minimality, certain regions, sampling, among others. In Example 1, our semantics generates minimal solutions as

	DEPENDENCY LANGUAGE				REPAIR STRATEGY		VALUE PREFERENCE			SOLUTION SELECTION			
System	FDs	CFDs	ERs	Int.T.CFDs	RHS	LHS	Confid.	Currency	Master	Cost	Certain	Card.Min	Sampling
[8]	√				√		√	√		√			
[10]	√	√			√	√	√	√		√			
[23]	√	√			√	√				√			
[18]			√		√				√		√		
[7]	√				√	√						√	√
LLUNATIC	√	√	√	√	√	√	√	√	√	√	√	√	√
	ext. dependencies				chase proced.		partial order			cost manager			

Table 1: Feature Comparison.

the ones in Figures 2 and 3, where L_i values represent lluns (confidence values have been omitted); notice that other minimal solutions exist for this example. Cost managers allow users to differentiate between these two solutions, which have completely different costs in terms of chase computation, and ultimately to fine-tune the tradeoff between quality and scalability of the repair process.

CUSTOMERS

	SSN	NAME	PHONE	STR	CITY	CC#
t_1	111	M. White	408-3334	Red Ave.	**SF**	112321
t_2	222	**F. Lennon**	122-1876	**Sky Dr.**	SF	$\mathbf{L_0}$
t_3	222	**F. Lennon**	**122-1876**	**Sky Dr.**	SF	$\mathbf{L_0}$

TREATMENTS

	SSN	SALARY	INSUR.	TREAT	DATE
t_4	111	**25K**	Abx	Dental	10/1/2011
t_5	111	25K	Abx	**Dental**	8/12/2012
t_6	222	30K	Med	Eye surg.	6/10/2012

Figure 2: Repaired Instance #1.

CUSTOMERS

	SSN	NAME	PHONE	STR	CITY	CC#
t_1	$\mathbf{L_1}$	M. White	408-3334	Red Ave.	NY	112321
t_2	$\mathbf{L_2}$	L. Lennon	122-1876	NULL	SF	781658
t_3	222	L. Lennon	000-0000	Fly Dr.	SF	784659

TREATMENTS

	SSN	SALARY	INSUR.	TREAT	DATE
t_4	111	**25K**	Abx	Dental	10/1/2011
t_5	111	25K	$\mathbf{L_3}$	Choles.	8/12/2012
t_6	222	30K	Med	Eye surg.	6/10/2012

Figure 3: Repaired Instance #2.

(iv) We develop an implementation of the chase engine, called LLUNATIC. To the best of our knowledge, LLUNATIC is the first system that runs over the DBMS to compute repairs. We devote special care in implementing our parallel chase – which may generate large trees of repairs – in a scalable way. A key ingredient of our solution is the development of an ad-hoc representation systems for solutions, called *delta relations*. In our experiments, we show that the chase engine scales to databases with millions of tuples, a considerable advancement in scalability wrt previous main-memory implementations.

We believe that these contributions make a significant advancement with respect to the state-of-the-art. To start, our proposal generalizes many previous approaches. Table 1 summarizes the features of LLUNATIC with respect to some of these approaches. LLUNATIC is the first proposal to achieve such a level of generality. Even more important, this work sheds some light on the crucial aspect of data-cleaning problems, namely the trade-offs between the quality of solutions and the complexity of repairing algorithms. This allows us to select data-repairing algorithms with good scalability and superior quality with respect to previous proposals, as our experiments show.

Organization of the Paper The preliminaries are in Section 2. In Sections 3, 4, and 5 we introduce the key components of the semantics of a cleaning scenario, which is defined in Section 6. The chase algorithm is described in Sections 7 and 8. Our experiments are reported in Section 9. Related work is described in Section 10.

2. PRELIMINARIES

We start by presenting some background notions and introducing the constraint language used in the paper.

A *schema* $\mathcal{S}$ is a finite set $\{R_1, \ldots, R_k\}$ of relation symbols, with each R_i having a fixed arity $n_i \geq 0$. Let CONSTS be a countably infinite domain of constant values, typically denoted by lowercase letters a, b, c, ... Let NULLS be a countably infinite set of labeled nulls, distinct from CONSTS. An *instance* $I = (I_1, \ldots, I_k)$ of $\mathcal{S}$ consists of finite relations $I_i \subset (\text{CONSTS} \cup \text{NULLS})^{n_i}$, for $i \in [1, k]$. Let R be a relation symbol in $\mathcal{S}$ with attributes $A_1, \ldots, A_n$ and I an instance of R. A *tuple* is an element of I and we denote by $t.A_i$ the value of tuple t in attribute A_i. Furthermore, we always assume the presence of *unique tuple identifiers* for tuples in an instance. That is, t_{tid} denotes the tuple with id "*tid*" in I. Given two disjoint schemas, $\mathcal{S}$ and $\mathcal{T}$, if I is an instance of $\mathcal{S}$ and J is an instance of $\mathcal{T}$, then the pair $\langle I, J\rangle$ is an instance of $\langle \mathcal{S}, \mathcal{T}\rangle$.

A relational atom over $\mathcal{T}$ is a formula of the form $R(\overline{x})$ with $R \in \mathcal{T}$ and $\overline{x}$ is a tuple of (not necessarily distinct) variables. Traditionally, an *equality generating dependency (egd)* over $\mathcal{T}$ is a formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow x_i = x_j)$ where $\phi(\overline{x})$ is a conjunction of relational atoms over $\mathcal{T}$ and x_i and x_j occur in $\overline{x}$.

To express data-cleaning contraints, we rely on a specific form of egd. More specifically, besides relation atoms, we also consider *equation atoms* of the form $t_1 = t_2$, where t_1, t_2 are either constants in CONSTS or variables, and allow for both source and target atoms in the premise. In our approach, a *cleaning egd* is then a formula of the form $\forall \overline{x}(\phi(\overline{x}) \rightarrow t_1 = t_2)$ where $\phi(\overline{x})$ is a conjunction of relational *and* equation atoms over $\langle \mathcal{S}, \mathcal{T}\rangle$, and $t_1 = t_2$ is of the form $x_i = c$ or $x_i = x_j$, for some variables x_i, x_j in $\overline{x}$ and constant $c \in$ CONSTS. Furthermore, at most one variable in the conclusion of an egd can appear in the premise as part of a relation atom over $\mathcal{S}$. The latter condition is to ensure that the egd specifies a constraint on the target database rather than on the fixed source database. With an abuse of notation, in the following we shall often refer to these cleaning egds simply as egds.

Egds for our running example are expressed as follows:

e_1. $\textit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p, s, c, cc), \textit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p', s', c', cc') \rightarrow p = p'$
e_2. $\textit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p, s, c, cc), \textit{Cust}(\boldsymbol{ssn}, \boldsymbol{n}, p', s', c', cc') \rightarrow cc = cc'$
e_3. $\textit{Treat}(ssn, s, ins, tr, d), ins = \text{'}\textit{Abx}\text{'} \rightarrow tr = \text{'}\textit{Dental}\text{'}$
e_4. $\textit{Cust}(\boldsymbol{ssn}, n, \boldsymbol{p}, s, c, cc), \textit{MD}(\boldsymbol{ssn}, n', \boldsymbol{p}, s', c') \rightarrow n = n'$
e_5. $\textit{Cust}(\boldsymbol{ssn}, n, \boldsymbol{p}, s, c, cc), \textit{MD}(\boldsymbol{ssn}, n', \boldsymbol{p}, s', c') \rightarrow s = s'$
e_6. $\textit{Cust}(\boldsymbol{ssn}, n, \boldsymbol{p}, s, c, cc), \textit{MD}(\boldsymbol{ssn}, n', \boldsymbol{p}, s', c') \rightarrow c = c'$
e_7. $\textit{Cust}(\boldsymbol{ssn}, n, p, str, c, cc), \textit{Treat}(\boldsymbol{ssn}, sal, ins, tr, d),$
$\quad ins = \text{'}\textit{Abx}\text{'} \rightarrow c = \text{'}\textit{SF}\text{'}$
e_8. $\textit{Treat}(\boldsymbol{ssn}, s, ins, tr, d), \textit{Treat}(\boldsymbol{ssn}, s', ins', tr', d') \rightarrow s = s'$

An immediate observation is that constants in egds can be avoided altogether, by encoding them in additional tables in the source database. Consider dependency e_3 in our example in which two constants appear: 'Abx' in attribute INSUR and 'Dental' in attribute TREAT. We extend $\mathcal{S}$ with an additional binary source table, denoted by CST_{e3} with attributes INSUR and TREAT, corresponding to the "constant" attributes in e_3. Furthermore, we instantiate CST_{e3} with the single tuple t_{e_3} : (Abx, Dental). Given this, e_3 can be expressed as an egd without constants, as follows:

$$e'_3.\ \mathit{Treat}(ssn, s, \boldsymbol{ins}, tr, d), \mathit{Cst}_{e_3}(\boldsymbol{ins}, tr') \rightarrow tr = tr'$$

In general, $\mathcal{S}$ can be extended with such constants tables, one for each CFD, and their source tables contain tuples for the constants used to define the CFD. In other words, these tables coincide with the pattern tableaux associated with the CFDs [16]. Of course, one needs to provide a proper semantics of egds such that whenever such constant tables are present, egds have the same semantics as CFDs. We give such semantics later in the paper.

Further extensions of egds with, e.g., built-in predicates, matching functions and negated atoms, are needed to encode matching dependencies and constraints for numerical attributes [20, 9]. We do not consider them in this paper for simplicity of exposition.

3. CLEANING SCENARIOS AND LLUNS

Our uniform framework for data repairing is centered around the concept of a *cleaning scenario*. A cleaning scenario consists essentially of a source schema $\mathcal{S}$, a target schema $\mathcal{T}$, and a set of constraints Σ. Here, $\mathcal{S}$ and $\mathcal{T}$ represent the two databases involved in the repairing process (see Example 1): (i) $\mathcal{S}$, the *source database*, provides clean and reliable information as input for the repairing process (like, for example, master data). We assume that source databases cannot be changed and consist of constants from CONSTS only; (ii) $\mathcal{T}$, the *target database*, corresponds to the database that is dirty relative to Σ, and that needs to be repaired. The target database may contain constants from CONSTS and null values from NULLS. Such null values indicate missing or unknown values. However, we also allow the target database to contain a third class of values, called *lluns* (pronounced "loons"), which we introduce next.

Recall from Example 1 that t_2 and t_3 form a violation for the dependency e_2 (stating that customers with equal ssns and names should have equal credit-card numbers), and that the target database could be repaired by equating t_2.CC# $= t_3$.CC#. However, as discussed before, no information is available as to which value should be taken in the repair. In such case, we repair the target database (for e_2) by changing t_2.CC# and t_3.CC# into the llun L_0, that is to indicate that we need to introduce a new value for the credit-card number that may be either 781658 or 784659, or some other preferred value. In this case, such value is currently unknown and we mark it so that it might be resolved later on into a constant, e.g., by asking for user input.

We denote by LLUNS $= \{L_1, L_2, \ldots\}$ an infinite set of symbols, called *lluns*, distinct from CONSTS and NULLS. Lluns can be regarded as the opposite of nulls since lluns carry "more information" than constants. In our approach, they play two important roles: (i) they allow us to complete the lattice induced by our partial orders, as it will be discussed in the next section; (ii) they provide a clean way to record inconsistencies in the data that require the intervention of users to be resolved.

With this in mind, given an instance J of $\mathcal{T}$, along with an instance I of $\mathcal{S}$, the goal is to compute a *repair* of J, i.e., a set of updates to J such that the resulting instance satisfies the constraints in Σ.

Early works about database repairing [3, 21] followed an approach that relied on tuple-insertions and tuple-deletions. Since tuple-deletions may bring to unnecessary loss of information, the recent literature has concentrated on tuple updates, instead. Roughly speaking, we may say that the semantics adopted in these works are centered around three main ideas. First, a repair is seen as a set of changes to the *cells* of the database (each cell being an attribute of a tuple). Second, the logic to repair conflicting values is hardcoded into the semantics. Third, cost functions are used to (heuristically) compare different repairs and choose the "good" ones.

In the following sections, we develop a new semantics for cleaning scenarios that departs from this standard in three significant ways. Our first intuition is that, in order to generalize the semantics to larger classes of constraints and different ways to pick-up preferred values, it is not sufficient to reason about single-cell updates. On the contrary, we need to introduce a notion of "repairs with a lineage", called *cell groups*, in the sense that: (a) we keep track of cells that need to be repaired together; (b) we keep track of the provenance of their values, especially if they come from the source database.

A second, key idea, is that the strategy to select preferred values and repair conflicts should be factored-out of the actual repairing algorithm. Our solution to do that is to introduce a notion of a *partial order* over cell groups. The partial order plays a central role in our semantics, since it allows us to identify when a repair is an actual "upgrade" of the original database.

Finally, we introduce a principled way to check when a repaired instance satisfies the constraints, and to compare repairs with one another. This is based on an extension to data cleaning of the notion of instance homomorphism [12] that is typically used to compare the relative information content of database instances.

The next sections are devoted to these notions.

4. CELL GROUPS AND REPAIRS

Given instance $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{T}\rangle$, we represent the set of changes made to repair the target database J in terms of *cell groups*. As the name suggests, cell groups are groups of *cells*, i.e., locations in a database specified by tuple/attribute pairs $t_{tid}.A_i$. For example, t_2.CC# and t_3.CC# are two cells in the CUSTOMERS table. Observe the following:

(a) As our example shows, to repair inconsistencies, different cells are often changed *together*, i.e., they are either changed all at the same time or not changed at all. For example, t_2.CC# and t_3.CC# are both modified to the same llun value in Figure 2. Cell groups thus need to specify a set of target cells, called *occurrences* of the group, and a value to update them.

(b) In addition, in some cases the target cells to repair receive their value from the source database; consider Example 1 and dependency e_5. When repairing t_2, cell t_2.STREET gets the value 'Sky Dr.' from cell t_m.STREET. Since source cells contain highly reliable information, it is important to keep track of the relationships among changes to target cells and values in the source. To do this, a cell group c has a set of associated source cells carrying provenance information about the repair in terms of cells of the source database. We call these source cells the *justifications* for c, since they provide lineage information for the change we make to the target cells in c, i.e., to its occurrences. Occurrences and justifications need to be kept separate since we can only update target cells, while source cells are immutable.

(c) Cell groups provide an elegant way of describing repairs. Indeed, in order to specify a repair it suffices to provide the original target database together with the set of cell groups to modify. In other words, cell groups can be seen as partial repairs with lineage.

These observations are captured by the following definitions:

Definition 1 [CELL GROUP] A *cell group g* over an instance $\langle I, J\rangle$ of $\langle \mathcal{S}, \mathcal{T}\rangle$ is a triple $\langle v \rightarrow \mathcal{C}, by\, \mathcal{C}_s\rangle$ where: (i) v is a value in CONSTS $\cup$ NULLS $\cup$ LLUNS; (ii) $\mathcal{C}$ is a finite set of cells of the target instance, J, called the *occurrences* of g, denoted by $occ(g)$;

(iii) $\mathcal{C}_s$ is a finite set of cells of the source instance, I, called the *justifications* of g, denoted by *just*(g).

A cell group $g = \langle v \rightarrow \mathcal{C}, by\, \mathcal{C}_s \rangle$ can be read as "change the target cells in $\mathcal{C}$ to value v, justified by the source cells in $\mathcal{C}_s$". We define a *repair* to an instance $\langle I, J \rangle$ as a set of cell groups.

Definition 2 [REPAIR] A *repair* $\mathsf{Rep} = \{g_0, \ldots, g_k\}$ for instances $\langle I, J \rangle$ is a (possibly empty) finite set of cell groups over $\langle I, J \rangle$, such that each cell of J occurs in at most one cell group g_i.

That is, each cell in J is either unchanged in a repair or it is modified in a unique way as described by the cell group to which it belongs. We denote by $g_{\mathsf{Rep}}(c)$ the cell-group of cell c according to Rep.

Example 2: In our example, consider the repair Rep_1, consisting of the following cell groups referred to as $g_1, \ldots, g_7$:

$$\begin{aligned}\mathsf{Rep}_1 = \{\, & g_1 : \langle L_0(\mathit{781658, 784659}) \rightarrow \{t_2.\text{CC\#}, t_3.\text{CC\#}\}, by\ \emptyset\rangle \\ & g_2 : \langle \mathit{F.Lennon} \rightarrow \{t_2.\text{NAME}, t_3.\text{NAME}\}, by\ \{t_m.\text{NAME}\}\rangle \\ & g_3 : \langle \mathit{122\text{-}1876} \rightarrow \{t_2.\text{PHN}, t_3.\text{PHN}\}, by\ \emptyset\rangle \\ & g_4 : \langle \mathit{Sky\ Dr.} \rightarrow \{t_2.\text{STR}, t_3.\text{STR}\}, by\ \{t_m.\text{STR}\}\rangle \\ & g_5 : \langle \mathit{Dental} \rightarrow \{t_5.\text{TREAT}\}, by\ \{t_{c_3}.\text{TREAT}\}\rangle \\ & g_6 : \langle \mathit{SF} \rightarrow \{t_1.\text{CITY}\}, by\ \{t_{c_7}.\text{CITY}\}\rangle\} \\ & g_7 : \langle \mathit{25K} \rightarrow \{t_4.\text{SALARY}, t_5.\text{SALARY}\}, by\ \emptyset\rangle\}\end{aligned}$$

Cell group g_1 fixes credit-card numbers for dependency e_2; it has empty justifications because no source relation is involved in e_2. On the contrary, cell groups g_2 and g_4 repair tuples t_2, t_3 for dependencies e_4, e_5; justifications for these groups contain the respective cells in the master-data tuple t_m. Similarly, for cell groups g_5, g_6; here t_{c_3}, t_{c_7} are the tuples in the CST_{e_3}, CST_{e_7} tables, encoding the constants in the original CFDs.

When applied to the original database in Figure 1, repair Rep_1 yields the repaired instance shown in Figure 2. Clearly, other repairs are possible. For example, to resolve e_3 one may consider changing the value of the cell $t_5.\text{INSURANCE}$ into a new llun value L_1, i.e., an unknown value that improves 'Abx'. The following repair, Rep_2, follows the same approach to satisfy all dependencies, and yields the repaired instance shown in Figure 3:

$$\begin{aligned}\mathsf{Rep}_2 = \{& g_7, \langle L_1 \rightarrow \{t_1.\text{SSN}\}, by\ \emptyset\rangle, \langle L_2 \rightarrow \{t_2.\text{SSN}\}, by\ \emptyset\rangle \\ & \langle L_3 \rightarrow \{t_5.\text{INSURANCE}\}, by\ \emptyset\rangle\}\end{aligned}$$

Note that J itself can be seen as the empty repair, $\mathsf{Rep}_\emptyset$.

Given $\langle I, J \rangle$, we say that a repair is *complete* if each cell of J occurs in a cell group in Rep, i.e., all cells in J are covered by the repair. We may assume, without loss of generality, that a repair is always complete. Indeed, a repair Rep can be easily completed into a complete repair Rep$'$, as follows:

(i) initially, we let $\mathsf{Rep}' = \mathsf{Rep}$;

(ii) for each cell c of J that is not changed by Rep, if $\mathit{val}(c) \in$ CONSTS, then we add to Rep$'$ the cell group $\langle \mathit{val}(c) \rightarrow \{c\}, by\ \emptyset\rangle$;

(iii) for each cell c of J that is not changed by Rep, if $\mathit{val}(c) \in$ NULLS, then we add to Rep$'$ the cell group of c with value $\mathit{val}(c)$, occurrences consisting of all cells of J in which $\mathit{val}(c)$ occurs and empty justifications.

From now on, we always assume a repair to be complete, and we blur the distinction between a repair Rep and the instance $\mathsf{Rep}(J)$ obtained by applying Rep to J.

5. THE PARTIAL ORDER

We are now ready to introduce another crucial ingredient of our framework: the partial order. The partial order is the core element of the semantics of our repairs and, as already mentioned, is used to indicate preferred upgrades to the target database. We want users to be able to specify different partial orders for different repairing problems in a simple manner. To do this, the user only has to specify for each attribute in the target schema when two values are preferred over each other. This is done by specifying an assignment $\sqcap$ of so-called *ordering attributes* to $\mathcal{T}$. As we will see shortly, such an assignment automatically induces a partial order on cell groups.

A Hierarchy of Information Content. In order to define our partial order, let us first introduce a simple hierarchy between the three kinds of values that appear in a database, namely nulls, constants, and lluns. More specifically, given two values $v_1, v_2 \in$ NULLS $\cup$ CONSTS $\cup$ LLUNS, we say that v_2 *is more informative* than v_1, in symbols $v_1 \lhd v_2$ if v_1 and v_2 are of different types, and one of the following holds: (i) $v_1 \in$ NULLS, i.e., the first value is a null value; or (ii) $v_2 \in$ LLUN, i.e., the second value is a llun.

User-Specified preferred values. We say that an attribute A of $\mathcal{T}$ has *ordered values* if its domain $\mathcal{D}_A$ is a partially ordered set. To specify which values should be preferred during the repair, users may associate with each attribute A_i of $\mathcal{T}$ a partially ordered set $\mathcal{P}_{A_i} = \langle D, \leq \rangle$. The poset $\mathcal{P}_{A_i}$ associated with attribute A_i may be the empty poset, or its domain $\mathcal{D}_{A_i}$ if A_i has ordered values, or the domain of a different attribute $\mathcal{D}_{A_j}$ that has ordered values. In the latter case, we call A_j the *ordering attribute* for A_i. Intuitively, $\mathcal{P}_{A_i}$ specifies the order of preference for values in the cells of A_i. An *assignment* of ordering attributes to attributes in $\mathcal{T}$ is denoted by $\sqcap$. For reasons that become clear shortly, $\sqcap$ is referred to as the *partial order specification*.

In our example, the DATE attribute in the TREATMENTS table, and the confidence column, CONF, in the CUSTOMERS table have ordered values (to simplify the treatment, we consider CONF as an attribute of the table). For these attributes, we choose the corresponding domain as the associated poset (i.e., we opt to prefer more recent dates and higher confidences). Other attributes, like the PHONE attribute in the CUSTOMERS table, have unordered values; we choose CONF as the ordering attribute for PHONE (a phone number will be preferred if its corresponding confidence value is higher). Notice that there may be attributes, like SALARY in TREATMENTS, that have ordered values; however, the natural ordering of values does not reflect our notion of a preferred value. To model the correct notion of preference, we use DATE as the ordering attribute for SALARY (we prefer most recent salaries). Finally, attributes like SSN will have an empty associated poset, i.e., all constant values are equally preferred. Below is a summary of the assignment $\sqcap$ of ordering attributes in our example (attributes not listed have an empty poset):

$$\sqcap = \left\{ \begin{array}{lcl} \mathcal{P}_{\text{CUSTOMERS.CONF}} & = & \mathcal{D}_{\text{CUSTOMERS.CONF}} \\ \mathcal{P}_{\text{TREATMENTS.DATE}} & = & \mathcal{D}_{\text{TREATMENTS.DATE}} \\ \mathcal{P}_{\text{CUSTOMERS.PHONE}} & = & \mathcal{D}_{\text{CUSTOMERS.CONF}} \\ \mathcal{P}_{\text{TREATMENTS.SALARY}} & = & \mathcal{D}_{\text{TREATMENTS.DATE}} \\ \mathcal{P}_{\text{CUSTOMERS.SSN}} & = & \emptyset \end{array} \right\}$$

Partial order on cell values. Given an assignment $\sqcap$, we can define a corresponding partial order $\preceq_J^{\sqcap}$ for the values of cells of the target instance J as follows. For any pair of values v_1, v_2 we say that $v_1 \preceq_J^{\sqcap} v_2$ iff one of the following holds:

(i) either $v_1 = v_2$ or $v_1 \lhd v_2$, i.e., the values are equal or the second one is more informative than the first;

(ii) v_1 appears in cell $t_1.A_1$, v_2 in cell $t_2.A_2$ in J, and both are constants in CONSTS; then, assume the ordering attributes for A_1 and A_2, called A'_1, A'_2 have the same poset, i.e., $\mathcal{P}_{A'_1} = \mathcal{P}_{A'_2}$; call v'_1, v'_2 the values of cells $t_1.A'_1, t_2.A'_2$. Then, $v_1 \preceq_J^{\sqcap} v_2$ iff $v_1 = v_2$ or $v'_1 < v'_2$ according to $\mathcal{P}_{A'_1} = \mathcal{P}_{A'_2}$.

We also consider values of the source instance I. In our approach, source values are immutable, and all equally preferable. So, we assume that the partial order $\preceq_I$ over values in I is based on rule (i) only. We call $\preceq^{\sqcap}_{\langle I,J\rangle}$ the partial order over values of cells in $\langle I, J\rangle$ obtained by the union of $\preceq^{\sqcap}_J$ and $\preceq_I$, with the additional rule that values of source cells are always preferable to values of target cells, i.e., for each target cell $t.A_t$ and source cell $t'.A_s$, it is always the case that $val(t.A_t) \preceq^{\sqcap}_{\langle I,J\rangle} val(t'.A_s)$. In fact, we always give preference to values from the source, like master-data or constant values in dependencies.

Given the partial order $\preceq^{\sqcap}_{\langle I,J\rangle}$, in the following we want to be able to compute upper bounds for cell values. To do this, we use lluns. Indeed, for any set C of cells we denote by $lub_{\preceq^{\sqcap}_{\langle I,J\rangle}}(C)$ the value that is (i) either the least upper bound for values of all cells in C according to $\preceq^{\sqcap}_{\langle I,J\rangle}$, if it exists; (ii) a new value N_i not in J, if all cells in C have null values; (iii) a new llun value L_j otherwise.

Partial order on cells groups. The partial order $\preceq^{\sqcap}_{\langle I,J\rangle}$ over cell values induces a partial order on the cell groups of $\langle I, J\rangle$. Before we turn to the definition, we want to exclude from the comparison cell groups that correspond to unjustified ways of changing the target. In order to do this, we say that a cell group g has a *valid value* if one of the following conditions holds. Consider the value val_{lub} that is the least upper bound of values in $occ(g) \cup just(g)$ according to $\preceq^{\sqcap}_{\langle I,J\rangle}$, i.e., $val_{lub} = lub_{\preceq^{\sqcap}_{\langle I,J\rangle}}(just(g) \cup occ(g))$. Then, either $val(g) = val_{lub}$, i.e., the cell group takes the value of the least upper bound, or $val_{lub} \lhd val(g)$, i.e., the cell group takes an even more informative value.

Given cell groups g and g' with valid values, we say that $g \preceq_{\sqcap} g'$ iff (i) $occ(g) \subseteq occ(g')$ and $just(g) \subseteq just(g')$, and (ii) either $val(g)$ and $val(g')$ are values of the same type (null, constant, or llun), or $val(g) \lhd val(g')$. In essence, we say that a cell group g' can only be preferred over a cell group g according to the partial order, if a *containment property* is satisfied, and the value of g' is at least as informative as the value of g. If the containment property is not satisfied for g and g' then these cell groups are incomparable relative to the partial order. Indeed, cell groups that change unrelated groups of cells represent incomparable ways to modify a target instance.

Example 3: Consider a simple relation $R(A, B)$, with three dependencies: (i) an FD $A \to B$, and two CFDs: (ii) $A[a] \to B[x], A[a] \to B[y]$. Notice that the two CFDs clearly contradict each other. Assume R contains two tuples: $t_1 : R(a, 1), t_2 : R(a, 2)$, and that $\mathcal{P}_A$ is A itself. Following is a set of ordered cell groups:

$$\begin{array}{cc}
\langle 1 \to \{t_1.\mathrm{B}\}, by\ \emptyset\rangle & \preceq_{\sqcap} \\
\langle 2 \to \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, by\ \emptyset\rangle & \preceq_{\sqcap} \\
\langle x \to \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, by\ \{t_{c1}.x\}\rangle & \preceq_{\sqcap} \\
\langle L \to \{t_1.\mathrm{B}, t_2.\mathrm{B}\}, by\ \{t_{c1}.x, t_{c2}.y\}\rangle &
\end{array}$$

Partial order on repairs. Given an instance $\langle I, J\rangle$, a partial order $\preceq_{\sqcap}$ over cell groups in $\langle I, J\rangle$, and two complete repairs, Rep, Rep', we say that Rep' *upgrades* Rep, denoted by $\mathsf{Rep} \preceq_{\sqcap} \mathsf{Rep}'$, if for each group $g \in \mathsf{Rep}$ there exists a group $g' \in \mathsf{Rep}'$ such that $g \preceq_{\sqcap} g'$. If $\mathsf{Rep} \preceq_{\sqcap} \mathsf{Rep}'$ and the converse does not hold, then we write $\mathsf{Rep} \prec_{\sqcap} \mathsf{Rep}'$. A repair Rep' is thus preferable to Rep whenever $\mathsf{Rep} \preceq_{\sqcap} \mathsf{Rep}'$. This is where the real strength of the partial order lies: it provides a uniform way of incorporating information on preferred repairs.

Proposition 1: *Given an assignment $\sqcap$ of ordering attributes to attributes in $\mathcal{T}$, the corresponding partial order $\preceq^{\sqcap}_{\langle I,J\rangle}$ over values of cells in $\langle I, J\rangle$ induces a partial order $\preceq_{\sqcap}$ over the cell groups and repairs of $\langle I, J\rangle$. In fact, $\preceq_{\sqcap}$ is semi-join lattice.*

Notice that, besides the standard rules above, users may specify additional custom rules to plug-in other value-selection strategies and refine the lattice of cell groups. As an example, a *frequency rule* may state that the lub of cell groups g and g' with constant values c_1 and c_2 and empty justifications should take as value c_1 (c_2, resp.) if $|occ(g_1)| > |occ(g_2)|$ ($|occ(g_2)| > |occ(g_1)|$, resp.).

6. SEMANTICS

With the partial order specification $\sqcap$ in place, we now define a cleaning scenario as a quadruple $\mathcal{CS} = \{\langle \mathcal{S}, \mathcal{T}\rangle, \Sigma, \sqcap\}$. Given a cleaning scenario and an instance $\langle I, J\rangle$, we address the problem of defining a *solution* for $\mathcal{CS}$ over $\langle I, J\rangle$. Intuitively, a solution is a repair for $\langle I, J\rangle$ that satisfies the set Σ of egds and is an upgrade of the original target instance J relative to $\preceq_{\sqcap}$. We next formalize these notions.

Consider an instance $\langle I, \mathsf{Rep}(J)\rangle$ and a set Σ of constraints. Usually, $\mathsf{Rep}(J)$ is called a solution if $\langle I, \mathsf{Rep}(J)\rangle$ satisfies Σ using the standard semantics of first-order logic. Since we want to —rather ambitiously— ensure that there is *always* a solution we need to revise this semantics. In contrast, previous proposals often fail to return a repair or are stuck in an endless loop during repairing, as is illustrated next.

Consider dependency e_3 from Example 1. Suppose that a contradictory dependency $e''_3.Treat(ssn, s, ins, tr, d), ins =$ '*Abx*' $\to tr =$ '*Cholest.*' is specified. In addition, assume that only modifications to the TREAT attribute-values are allowed. Clearly, there is no repair made of constants that can satisfy both dependencies [16]. However, one may consider of changing 'Dental' and 'Cholest.' to a llun L that improves both original values. In essence, the llun has the role of indicating to the user that the constraints are contradictory. In our setting, we want to regard this repair as a solution of a conflicting cleaning scenario.

Consider an egd $e : \forall x\, \phi(\bar{x}) \to x = x'$. First, recall that, in the standard semantics, $\langle I, \mathsf{Rep}(J)\rangle$ satisfies e if for any *homomorphism* h that maps the variables $\bar{x}$ into values of $\langle I, \mathsf{Rep}(J)\rangle$ such that $\phi(h(\bar{x}))$ is true, then also $h(x) = h(x')$ must be true. We want this to hold in our semantics as well. However, we want more. That is, we allow $h(x) \neq h(x')$ as long as the cell group corresponding to $h(x)$ is an *upgrade* to the cell groups corresponding to $h(x')$, or vice versa.

To make this precise, we need to extend h to a mapping from variables to cell groups. Since h associates values to variables, it also associates with each variable $x_i \in \bar{x}$ a set of cells from $\langle I, \mathsf{Rep}(J)\rangle$, called $cells_h(x_i)$, one for each occurrence of x_i and all with the same value, $h(x_i)$. We use these to define the cell group of x_i according to h, as follows.

Given a formula $\phi(\bar{x})$, a repair Rep, an homomorphism h of $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J)\rangle$, and a variable $x_i \in \bar{x}$, the *cell group of x_i according to h* is defined as $g_h(x_i) = \langle h(x_i) \to \mathcal{C}, by\ \mathcal{C}_s\rangle$ where $\mathcal{C}$ (resp. $\mathcal{C}_s$) is the union of all occurrences (resp. justifications) of cell groups $g_{\mathsf{Rep}}(c_i)$ in Rep, for each cell $c_i \in cells_h(x_i)$. In addition, $\mathcal{C}_s$ contains all cells in $cells_h(x_i)$ that belong to the source I.

We are now ready to introduce our extended notion of satisfaction, namely *satisfaction after repairs*:

Definition 3 [SATISFACTION AFTER REPAIRS] Given an egd $e : \forall\bar{x}\ \phi(\bar{x}) \to x = x'$, an instance $\langle I, J\rangle$, and a repair Rep, we say that $\langle I, \mathsf{Rep}(J)\rangle$ *satisfies after repairs* e wrt the partial order $\preceq_{\sqcap}$ if, whenever there is an homomorphism h of $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J)\rangle$, then (i) either the value of $h(x)$ and $h(x')$ are equal, or (ii) it is the case that $g_h(x) \preceq_{\sqcap} g_h(x')$ or $g_h(x') \preceq_{\sqcap} g_h(x)$.

We can now find a repair that satisfies the conflicting egds e_3 and e_3'' above. Given a tuple t in the target, consider Rep that repairs t.TREAT with L, and justifies the change with both cells in the source corresponding to constants '*Dental*' and '*Cholest.*'. Now, despite the fact that L is not equal to any of the constants in the dependencies, both dependencies are satisfied after repairs by $\mathsf{Rep}(J)$.

Definition 4 [SOLUTION] Given a cleaning scenario $\mathcal{CS} = \{\langle \mathcal{S}, \mathcal{T} \rangle, \Sigma, \sqcap\}$ and instance $\langle I, J \rangle$ a *solution* for $\mathcal{CS}$ over $\langle I, J \rangle$ is a repair Rep such that: (i) $J \preceq_{\sqcap} \mathsf{Rep}$, i.e., Rep upgrades J; and (ii) $\langle I, \mathsf{Rep}(J) \rangle$ satisfies after repairs Σ wrt $\preceq_{\sqcap}$.

An important property of cleaning scenarios is that every input instance has a solution, albeit a solution that is not necessarily minimal and is rather uninformative.

Theorem 2: *Given a scenario $\mathcal{CS} = \{\langle \mathcal{S}, \mathcal{T} \rangle, \Sigma, \sqcap\}$ and an input instance $\langle I, J \rangle$, there always exists a solution for $\mathcal{CS}$ and $\langle I, J \rangle$.*

Proof: Indeed, there is always a solution corresponding to the repair that changes all cells of J to a single llun L, and justifies it by all cells in I, i.e., $\mathsf{Rep}_{trivial} = \langle L \rightarrow cells(J), by\ cells(I) \rangle$. □

Among all possible repairs, we are interested in those that minimally upgrade the dirty instance.

Definition 5 [MINIMAL SOLUTION] A *minimal solution* for a cleaning scenario is any solution Rep that is minimal wrt $\prec_{\sqcap}$, i.e., such that there exists no other solution Rep' such that $\mathsf{Rep}' \prec_{\sqcap} \mathsf{Rep}$.

The repair Rep_1 in Example 2 is a minimal solution for the scenario in Example 1: it is an upgrade of J, it satisfies the dependencies, and by undoing any of its changes violations arise. Minimal solutions are not unique. Indeed, also repair Rep_2 in Example 2 is a minimal solution. As an example of a non-minimal solution, one can add to Rep_2 the cell group $\langle L_4 \rightarrow \{t_2.\text{NAME}\}, by\ \emptyset \rangle$. The resulting repair Rep_3 is still a solution but not a minimal one ($\mathsf{Rep}_2 \prec_{\sqcap} \mathsf{Rep}_3$). Consider now repair Rep_4, obtained by adding a cell group $\langle \mathit{111111} \rightarrow \{t_2.\ \text{CC\#}\}, by\ \emptyset \rangle$ to Rep_2. In this case, Rep_4 is not a solution because the last cell group is totally unjustified wrt the partial order, and therefore it is not true that $\mathsf{Rep}_4(J)$ is an upgrade of the original target instance.

An important property is that two repairs can be efficiently compared wrt to the partial order. We assume here that the partial order of two values $v \preceq^{\sqcap}_{\langle I, J \rangle} v'$ can be checked in constant time.

Theorem 3: *Given two solutions $\mathsf{Rep}, \mathsf{Rep}'$ for a scenario $\mathcal{CS}$ over instance $\langle I, J \rangle$, one can check $\mathsf{Rep} \preceq_{\sqcap} \mathsf{Rep}'$ in $O(n + kmlog(m))$ time, where n is the number of cells in J, k is the maximum number of cell groups in $\mathsf{Rep}, \mathsf{Rep}'$, and m is the maximum size of a cell group in $\mathsf{Rep}, \mathsf{Rep}'$.*

Given a cleaning scenario $\mathcal{CS}$ and an instance $\langle I, J \rangle$, the *data repairing problem* consists of computing all minimal solutions for $\mathcal{CS}$ over $\langle I, J \rangle$. We provide a chase-based algorithm for the data repairing problem in the next section.

What are Lluns, in the End? The role and the importance of lluns should now be apparent. While lluns are nothing more than symbols from a distinguished set, like constants and nulls, their use in conjunction with cell groups makes them a powerful addition to the semantics. Not only they allow us to complete the lattice of cell-groups and repairs, but, when appearing inside cell-groups, they also provide important lineage information to support users in the delicate task of resolving conflicts. Consider again Example 3 in Section 5. The cell group $\langle L \rightarrow \{t_1.\text{B}, t_2.\text{B}\}, by\ \{t_{c1}.x, t_{c2}.y\} \rangle$ is a clear indication that it was not possible to fully resolve the conflicts, and therefore user interventions are needed to complete the repair. In addition, the cell-group provides complete information about the conflict, both in terms of which target cells – and therefore which original values – where involved, and also in terms of source values that justify the change.

7. COMPUTING SOLUTIONS

In order to generate solutions for cleaning scenarios, we resort to a variant of the traditional chase procedure for egds [12]. However, our chase is a significant departure from the standard one, for several reasons: (i) during the chase, we shall make extensive use of the partial order, $\preceq_{\sqcap}$; (ii) to generate all possible solutions, a dependency may be chased both *forward*, to satisfy its conclusion, or *backward*, to falsify its premise; this, in turn, means that the we need to consider a *disjunctive chase*, which generates a tree of alternative repairs; (iii) finally, and most important, we shall not consider violations at the tuple level, as it is common [12], but at the higher level of *equivalence classes*.

To explain this latter difference, consider a simple functional dependency $A \rightarrow B$ over relation $R(A, B, C)$, with tuples $t_1 = R(1, 2, x)$, $t_2 = R(1, 2, y)$, $t_3 = R(1, 4, z)$, $t_4 = R(2, 5, w)$, $t_5 = R(2, 5, v)$. It is highly inefficient to analyze the violations of this FD at the tuple level; in fact, eventually, the B value of t_1, t_2, t_3 will all become equal, and therefore one may prefer grouping and fixing them together. In the literature [8, 15] this has been formalized by means of *equivalence classes*. We want to introduce a similar concept into our chase algorithm. Given the higher generality of our dependency language, we need a number of preliminary definitions.

Preliminary Notions Recall that, given an homomorphism h of a formula $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J) \rangle$, we denote by $g_h(x)$ the cell group associated by h with variable x. We first introduce the notions of *witness* and *witness variable* for a dependency e. Intuitively, the witness variables are those variables upon which the satisfiability of the dependency premise depends; these are all variables that have more than one occurrence in the premise, i.e., they are involved in a join or in a selection.

Definition 6 [WITNESS] Let $e : \forall \bar{x}\ (\phi(\bar{x}) \rightarrow x = x')$ be an egd. A *witness variable* for e is a variable $x \in \bar{x}$ that has multiple occurrences in $\phi(\bar{x})$. For an homomorphism h of $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J) \rangle$, we call a *witness*, w_h for e and h, the vector of values $h(\bar{x}_w)$ for the witness variables $\bar{x}_w$ of e.

Consider, for example, dependency e_8 in Example 1 (we omit some of the variables for the sake of conciseness): e_8. $\mathit{Treat}(\mathbf{ssn}, s, \ldots), \mathit{Treat}(\mathbf{ssn}, s', \ldots) \rightarrow s = s'$. Assume that the target instance TREATMENTS contains tuples $t_4 = (\mathit{ssn} : 222, \mathit{salary} : 10K, \ldots)$, $t_5 = (\mathit{ssn} : 222, \mathit{salary} : 25K, \ldots)$. We have an homomorphism h that maps the first atom of e_8 into t_4, and the second one into t_5. In this case, the witness variable, i.e., the variable that imposes the constraint that the two tuples have the same SSN, is *ssn*, and its value is 222.

Definition 7 [EQUIVALENCE CLASS] Given a repair Rep, and an egd $e : \forall \bar{x}\ (\phi(\bar{x}) \rightarrow x = x')$, let $\bar{x}_w \subseteq \bar{x}$ be the witness variables of e. An *equivalence* class for Rep and e, $\mathcal{H}$, is a set of homomorphisms of $\phi(\bar{x})$ into $\langle I, \mathsf{Rep}(J) \rangle$ such that all $h_i \in \mathcal{H}$ have equal witness values $h_i(\bar{x}_w)$.

Notice that equivalence classes induce classes of tuples in a natural way. In our example above, the tuples are partitioned into two equivalence classes, as follows: $ec_1 = \{t_1, t_2, t_3\}$ (with witness 1) and $ec_2 = \{t_4, t_5\}$ (with witness 2).

To identify a violation, we look for different values in the conclusion of e. To see an example, consider the equivalence class

ec_1 (witness 1), composed of the three tuples $\{t_1, t_2, t_3\}$: to identify the violation, we notice that they have two different values for the B attribute, 2 and 4, respectively. To formalize this, we introduce the set of *witness groups*, $\text{w-groups}_{\mathcal{H}}$, and *conclusion groups*, $\text{c-groups}_{\mathcal{H}}$, for $\mathcal{H}$ and e, as the set of cell groups associated by any homomorphism $h \in \mathcal{H}$ with the witness variables, $\bar{x}_w$, and the conclusion variables, x, x', respectively:

$$\text{w-groups}_{\mathcal{H}} = \bigcup_{h \in \mathcal{H}, x_w \in \bar{x}_w} g_h(x_w)$$
$$\text{c-groups}_{\mathcal{H}} = \bigcup_{h \in \mathcal{H}} g_h(x) \cup \bigcup_{h \in \mathcal{H}} g_h(x')$$

We say that an equivalence class for Rep and e generates a *violation* if it has at least two conclusion groups with different values and such that there is no ordering among them, i.e, there exist $g_1, g_2 \in \text{c-groups}_{\mathcal{H}}$ such that $val(g_1) \neq val(g_2)$ and neither $g_1 \preceq_\sqcap g_2$ nor $g_2 \preceq_\sqcap g_1$. In this case, we say that e is *applicable* to $\langle I, \text{Rep}(J)\rangle$ with $\mathcal{H}$.

The Chase We are now ready to define the notion of a chase step. Our goal is to define the chase in such a way that it is as general as possible, but at the same time it allows to plug-in optimizations to tame the exponential complexity. In order to do this, we introduce the crucial notion of a *repair strategy* for an equivalence class, which provides the hook to introduce the notion of a cost manager in the next section.

A *repair strategy* $rs_{\mathcal{H}}$ for $\mathcal{H}$ is a mapping from the set of conclusion cell-groups, $\text{c-groups}_{\mathcal{H}}$ of Rep and $\mathcal{H}$, into the set $\{f, b\}$ (where f stands for "forward", and b for "backward"). We call the *forward groups*, $\text{forw-g}_{rs_{\mathcal{H}}}$, of $rs_{\mathcal{H}}$ the set of groups g_i such that $rs_{\mathcal{H}}(g_i) = f$, and the *backward groups*, $\text{back-g}_{rs_{\mathcal{H}}}$, those such that $rs_{\mathcal{H}}(g_i) = b$.

For each backward group $g \in \text{back-g}_{rs_{\mathcal{H}}}$ and for each target cell $c_i \in g$, we assume that the repair strategy $rs_{\mathcal{H}}$ also identifies (whenever this exists) one of the witness cells in $\text{w-groups}_{\mathcal{H}}$ to be backward-repaired. This cell, denoted by $\text{w-cell}_{rs_{\mathcal{H}}}(c_i)$, must be such that:

(i) it belongs to the same tuple as c_i;

(ii) the corresponding cell group g_i according to Rep has a constant value, i.e., $val(g_i) \in$ CONSTS;

(iii) the corresponding cell group g_i has empty justifications, i.e., $just(g_i) = \emptyset$.

Observe that we do not chase backward in two cases: first, when cells contain nulls or lluns; in fact, nulls and lluns are essentially placeholders, and there is no need to replace a placeholder by another one, since this is does not represent an upgrade of the repair; second, when cell values have a justification from the source; since we use the source to model high-reliability data, we consider it unacceptable to disrupt a value coming from the source in favor of a llun.

Each chase step is defined based on a specific repair strategy.

Definition 8 [CHASE STEP] Given a cleaning scenario $\mathcal{CS} = \{\mathcal{S}, \mathcal{T}, \Sigma, \sqcap\}$, and a complete repair Rep of J, let $e : \forall \bar{x}\ (\phi(\bar{x}) \rightarrow x = x')$ be an egd in Σ, applicable to $\langle I, \text{Rep}(J)\rangle$ with $\mathcal{H}$. For each repair strategy $rs_{\mathcal{H}}$, a *chase step* generates a new repair $\text{Rep}_{rs_{\mathcal{H}}}$ defined as follows:

(i) to start, we initialize $\text{Rep}_{rs_{\mathcal{H}}} = \text{Rep}$

(ii) then, we replace all forward groups by their least upper bound:

$$\text{Rep}_{rs_{\mathcal{H}}} = \text{Rep}_{rs_{\mathcal{H}}} - \text{forw-g}_{rs_{\mathcal{H}}} \cup lub_{\preceq_\sqcap}(\text{forw-g}_{rs_{\mathcal{H}}})$$

(iii) finally, we add the backward repairs, i.e, for each backward group $g \in \text{back-g}_{rs_{\mathcal{H}}}$, and cell $c_i \in occ(g)$, we replace $g_i = g_{\text{Rep}}(\text{w-cell}_{rs_{\mathcal{H}}}(c_i))$ by the cell group g_i' that is an immediate successor of g_i according to $\preceq_\sqcap$ as follows:

$$\text{Rep}_{rs_{\mathcal{H}}} = \text{Rep}_{rs_{\mathcal{H}}} - \{g_i\} \cup \{g_i'\}$$

Note that such a successor always exists. Indeed, $g_i' = \langle L_i \rightarrow occ(g_i), by\ \emptyset\rangle$, where L_i is a new LLUN value, is a successor of g_i.

Given Rep, each repair strategy $rs^i_{\mathcal{H}}$ for $\mathcal{H}$ generates a different step, $\text{Rep}_{rs^i_{\mathcal{H}}}$. We simultaneously consider all these chase steps, in parallel, and write $\text{Rep} \rightarrow_{e,\mathcal{H}} \text{Rep}_{rs^0_{\mathcal{H}}}, \text{Rep}_{rs^1_{\mathcal{H}}} \ldots, \text{Rep}_{rs^n_{\mathcal{H}}}$.

Consider again dependency e_8 in Example 1, and the equivalence class associated with witness *ssn* $= 222$. The cell groups for the conclusion cells are: $g = \langle 10K \rightarrow \{t_4.\text{SALARY}\}, by\ \emptyset\rangle$ and $g' = \langle 25K \rightarrow \{t_5.\text{SALARY}\}, by\ \emptyset\rangle$. Notice that the two cell groups are incomparable, and therefore we have a violation. The chase procedure generates three different repairs for the violation: (a) the forward repair is: $\text{Rep}_{f,f} = \langle 25K \rightarrow \{t_4.\text{SALARY}, t_5.\text{SALARY}\}, by\ \emptyset\rangle$ ($25K$ is more recent than $10K$ as a salary, and therefore it is preferred); as you can see, the least upper bound is constructed in such a way that it contains the union of occurrences and the union of justifications of the two conflicting groups; (b) the first backward repair, which changes the first occurrence of the witness variable *ssn* to a llun L_1: $\text{Rep}_{b,f} = \langle L_1 \rightarrow \{t_4.\text{SSN}\}, by\ \emptyset\rangle$; (c) the second backward repair, changing the second occurrence of *ssn* to L_2: $\text{Rep}_{f,b} = \langle L_2 \rightarrow \{t_5.\text{SSN}\}, by\ \emptyset\rangle$.

Definition 9 [CHASE TREE] Given a cleaning scenario $\mathcal{CS} = \{\mathcal{S}, \mathcal{T}, \Sigma, \sqcap\}$, a *chase* of $\langle I, J\rangle$ with Σ is a tree whose root is $\langle I, J\rangle$, i.e., the empty repair, and for each node Rep, the children of Rep are the repairs $\text{Rep}_0, \text{Rep}_1, \ldots, \text{Rep}_n$ such that, for some $e \in \Sigma$ and some $\mathcal{H}$, it is the case that $\text{Rep} \rightarrow_{e,\mathcal{H}} \text{Rep}_0, \text{Rep}_1, \ldots, \text{Rep}_n$. The leaves are repairs Rep_ℓ such that there is no dependency applicable to $\langle I, \text{Rep}_\ell(J)\rangle$ with some equivalence class $\mathcal{H}$. Any leaf in the chase tree is called a *result* of the chase of $\langle I, J\rangle$ with Σ.

Note that, as usual, the chase procedure is sensitive to the order of application of the dependencies. Different orders of application of the dependencies may lead to different chase sequences and therefore to different results.

We next show that the chase procedure always generates solutions, i.e., it is sound, and it terminates after a finite number of steps. Furthermore, all minimal solutions can be obtained in this way, i.e., the chase is complete for minimal solutions.

Theorem 4: *Given a cleaning scenario $\mathcal{CS} = \{\mathcal{S}, \mathcal{T}, \Sigma, \sqcap\}$ and an instance $\langle I, J\rangle$, the chase of $\langle I, J\rangle$ with Σ (i) terminates; (ii) it generates a finite set of results, each of which is a solution for $\mathcal{CS}$ over $\langle I, J\rangle$; and (iii) it generates all minimal solutions.*

Complexity It is well-known [5] that a database can have an exponential number of solutions, even for a cleaning scenario with a single FD and when no backward chase steps are allowed. In general, it is readily verified that a cleaning scenario can have at most an exponential number of solutions. When considering the disjunctive chase procedure, as outlined above, one can verify that each solution is computed in a number of steps that is polynomial in the size of the data. For this, it suffices to observe that one can associate an integer-valued function f on repairs such that $f(\text{Rep}) < f(\text{Rep}')$ whenever $\text{Rep} \rightarrow_{e,\mathcal{H}} \text{Rep}'$ during the chase. Intuitively, f depends on the number of llun values and sizes of cell groups in the repairs. Since both the number of lluns and size of cell groups is bounded by the input instance, we may infer that f cannot be increased further after polynomially many steps, i.e., when a solution is obtained.

In contrast, computing all solutions by means of the chase takes exponential time in the size of instance. Indeed, given the polynomial size of each branch in the chase tree, as argued above, and the fact that the branching factor is polynomially bounded by the input, the overall chase tree is exponential in size. We discuss techniques to handle this high complexity in the next section.

8. A SCALABLE CHASE

The chase procedure defined in the previous section provides an elegant operational semantics for cleaning scenarios. However, as argued above, computing all solutions has very high complexity, which makes the chase often impractical. In this section, we introduce a number of techniques that improve the scalability of the chase, namely: a central component of our framework, the *cost manager*, and a representation systems for chase trees, called *delta databases*.

8.1 Introducing the Cost Manager

Chasing at the equivalence-class level is more efficient than chasing at the tuple level, but by itself it does not reduce the total number of solutions, and ultimately the complexity of the whole chase process. In fact, previous proposals have chosen many different and often ad-hoc ways to reduce the complexity by discarding some of the solutions in favor of others. Among these we mention various notions of minimality of the repairs [3, 8, 7], certain regions [18], and sampling [7]. We propose to incorporate these pruning methods into the chase process in a more principled and user-customizable way by introducing a component, called the *cost manager*.

Definition 10 [COST MANAGER] Given a cleaning scenario, $\mathcal{CS}$ and instance $\langle I, J\rangle$, a *cost manager* for $\mathcal{CS}$ and $\langle I, J\rangle$ is a predicate CM over repair strategies to be used during the chase. For each repair strategy $rs_{\mathcal{H}}$ for equivalence class $\mathcal{H}$, it may either accept it ($\text{CM}(rs_{\mathcal{H}}) = \mathit{true}$), or refuse it ($\text{CM}(rs_{\mathcal{H}}) = \mathit{false}$).

During the chase, we shall systematically make use of the cost manager. Whenever we need to chase an equivalence class, we only generate repairs corresponding to repair strategies accepted by the cost manager. The standard cost manager is the one that accepts all repair strategies, and may be used for very small scenarios. As an alternative, our implementation offers a rich library of cost managers. Among these, we mention the following, that have been used in experiments:

– a *maximum size* cost manager (SN): it accepts repair strategies as long as the number of leaves in the chase tree (i.e., the repairs produced so far) are less than N; as soon as the size of the chase tree exceeds N, it accepts only the first one of them, and rejects the rest; as a specific case, the S1 cost manager only considers one order of application of the dependencies, and ignores other permutations;

– a *frequency* cost manager (FR): in order to repair equivalence class $\mathcal{H}$ for dependency e, FR adopts the following rules; it relies on the frequency of values appearing in conclusion cells, and on a similarity measure for values (based on the Levenshtein distance for strings); then: (i) it rejects repair strategies that backward-chase cells with the most frequent conclusion value; (ii) for every other conclusion cell, if its value is similar (distance below a fixed threshold) to the most frequent one, the cell is forward-chased; otherwise, it is backward chased; this is typically used with a frequency rule in the partial order of cell-groups;

– a *forward-only* cost manager (FO): it accepts forward-only repair strategies, and rejects those that perform backward repairs.

Notice that combinations of these strategies are possible, to obtain, e.g., a FR-S5 or a FR-S1-FO cost manager. The FR-S5 relies on value frequencies and, in addition, it considers five different permutations of the dependencies, and for each of them will compute one repair. Alternative cost managers may implement different pruning strategies, to incorporate the notion of a *certain region*, and refute all steps in which changes are made to attributes of the target that are considered to be "fixed", i.e., reliable, or perform different forms of sampling. In the following, we shall always assume that a cost manager has been selected in order to perform the chase.

8.2 Delta Databases

Even with cost managers in place, the parallel nature of our chase algorithm imposes to store a possibly large tree of repairs. A naive approach in which new copies of the whole database are created whenever we need to generate a new node in the tree, is clearly inefficient. To solve this problem, we introduce an ad-hoc *representation system* for nodes in our chase trees, called *delta databases*. Delta databases are a formalism to store a finite set of worlds into a single relational database. Intuitively, they allow to store "deltas", i.e., modifications to the original database, rather than entire instances as is done in the naive approach.

Delta relations rely on an attribute-level storage system, inspired by *U-relations* [2], modified to efficiently store cell groups and chase sequences. More specifically, (i) each column in the original database is stored in a separate *delta relation*, to be able to record cell-level changes; (ii) chase steps are identified by a function with a prefix property, such that the id of the father of n is a prefix of the encoding of n; this allows to quickly reconstruct the state of the database at any given step, using fast SQL queries; (iii) additional tables are used to store cell groups, i.e., occurrences and justifications.

More formally, we introduce a function *stepId*$()$ that associates a string id with each chase step, i.e., with each node in the chase tree, and has the prefix property such that, *stepId*$(father(n))$ is a prefix of *stepId*(n), for each n. For this, we use the function that assigns the id *r* to the root, *r.0, r.1, . . ., r.n* to its children, and so on.

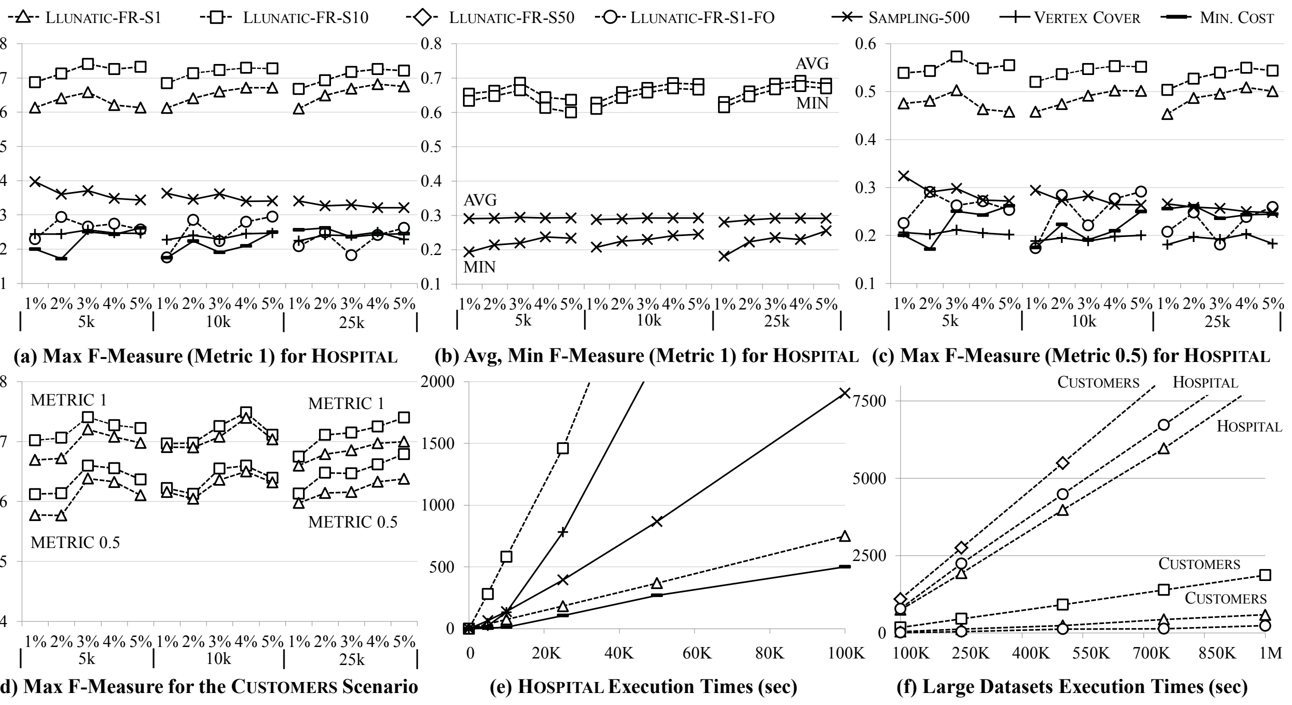
Definition 11 [DELTA DATABASE] Given a target database schema $\mathcal{R} = \{R_1, \ldots, R_k\}$, a *delta database* for $\mathcal{R}$ contains the following tables: (i) a *delta table* $R_i\_A_j$ with attributes (t_{id}, *stepId, value*), for each R_i and each attribute A_j of R_i; (ii) a table *occurrences*, with schema (*stepId, value,* t_{id}*, table, attr*); (iii) a table *justifications*, with schema (*stepId, value,* t_{id}*, table, attr*).

During the chase, we store the whole chase tree into the delta database. We do not perform updates, which are slow, but execute inserts instead. Whenever, at step s, a cell $t_{id}.A$ in table R is changed to value v, we store a new tuple in the delta table R_A with value (t_{id}, *stepId*, v). Using this representation, it is possible to store trees of hundreds of nodes quite efficiently. In addition, it is relatively easy to find violations using SQL (the actual queries are omitted for space reasons).

In the next section we show how the combination of our advanced chase procedure and its implementation under the form of delta databases scale to large repairing problems with millions of tuples and large chase trees.

9. EXPERIMENTS

The proposed algorithms have been implemented in a working prototype of the LLUNATIC system, written in Java. In this section, we consider several cleaning scenarios, of different nature and sizes, and study both the quality of the repairs computed by our system, and the scalability of the chase algorithm. We show that our algorithm produce repairs of better quality with respect to other systems in the literature, and at the same time scales to large databases. All experiments have been executed on a Intel i7 machine with 2.6Ghz processor and 8GB of RAM under Linux. The DBMS was PostgreSQL 9.2.



(a) Max F-Measure (Metric 1) for HOSPITAL **(b) Avg, Min F-Measure (Metric 1) for HOSPITAL** **(c) Max F-Measure (Metric 0.5) for HOSPITAL**

(d) Max F-Measure for the CUSTOMERS Scenario **(e) HOSPITAL Execution Times (sec)** **(f) Large Datasets Execution Times (sec)**

Figure 4: Experimental results for HOSPITAL and CUSTOMERS.

Datasets and Scenarios. We selected two scenarios. (i) The first one, HOSPITAL, is based on a dataset from US Department of Health & Human Services (http://www.medicare.gov/hospitalcompare/). The database contains a single table with 100K tuples and 19 attributes, over which we specified 9 functional dependencies. (ii) The second one, CUSTOMERS, corresponds to our running example in Figure 1. The database schema contains 3 tables with 16 attributes, plus 2 additional tables encoding constants in CFDs. Dependencies are the ones in Section 1. We synthetically generated up to 1M tuples for the 2 target relations, with a proportion of 40% in the CUSTOMERS table, and 60% in TREATMENTS; the master-data table contains 20% of the tuples present in CUSTOMERS. We consider master-data tuples outside the total, as they cannot be modified. For this scenario, we defined the partial order as discussed in Section 5.

It is worth noting that these scenarios somehow represent opposite extremes of the spectrum of data-repairing problems. In fact, the HOSPITAL scenario contains functional dependencies only, and therefore is quite standard in terms of constraints; however, it can be considered a worst-case in terms of scalability, since all data are stored as a single, non-normalized table, with many attributes and lots of redundancy; over this single table, the 9 dependencies interact in various ways, and there is no partial-order information that can be used to ameliorate the cleaning process.

On the contrary, the CUSTOMERS scenario contains a complex mix of dependencies; this increased complexity of the constraints is compensated by the fact that data is stored as two normalized tables, with no redundancy, and clear preference strategies are given for some of the attributes.

Errors. In order to test our algorithms with different levels of noise, we introduced errors in the two datasets. Part of these errors were generated by a random-noise generator. However, in order to be as close as possible to real scenarios, in the HOSPITAL dataset we also used a different source of noise. We asked workers from Mechanical Turk (MT) (https://www.mturk.com/mturk/) to perform data entry for a random sample of tuples from the original database. Workers were shown the original tuple under the form of a jpeg image, and needed to manually copy values into a form. We used different groups of workers with different approval rates; approval rates measure the quality of a worker in terms of the percentage of previous jobs positively evaluated within MT. Approval rates varied between 50% and 99%; for these, we observed a percentage of wrong values between 5% and 1%. These errors were then complemented with those generated by the random noise generator.

For both datasets, we generated dirty copies with a number of noisy cells ranging from 1% to 5% of the total. Changes to the original values were done only for attributes involved in dependencies, in order to maximize the probability of generating detectable violations.

Algorithms. We tested LLUNATIC with several cost managers chosen among those presented in Section 8. We chose variants of the LLUNATIC-FR-SN cost manager – the frequency cost-manager that generates up to N solutions – with $N = 1, 10, 50$, and the LLUNATIC-FR-S1-FO, the forward-only variant of LLUNATIC-FR-S1. We do not report results obtained by the standard cost manager, as it only can be used with small instances due to its high computing times.

In order to compare our system to previous approaches, we tested also the following FD repair algorithms from the literature, implemented as separate systems: (a) *Mimimum Cost* [8] (MIN. COST); (b) *Vertex Cover* [23] (VERTEX COVER); (c) *Repair Sampling* [7] (SAMPLING), for which, for each experiment, we took 500 samples, as done in the original paper.

Notice that these systems support a smaller class of constraints wrt to the ones expressible with cleaning egds (essentially FDs and, in some cases, CFDs). Several of the constraints in the CUSTOMERS scenario are outside of this class, and therefore cannot handled by these algorithms. We therefore performed the comparison on the HOSPITAL scenario only.

Quality Metrics. We used precision-recall metrics. More specifically, for each clean database, we generated the set $\mathcal{C}_p$ of perturbated cells. Then, we run each algorithm to generate a set of repaired cells, $\mathcal{C}_r$, and computed precision (P), recall (R), and F-measure ($F = 2 \times (P \times R)/(P + R)$) of $\mathcal{C}_r$ wrt $\mathcal{C}_p$. Since several of the algorithms may introduce variables as repairs – like our lluns – we calculated two different metrics.

The first one is the one adopted in [7], which we call *Metric 0.5*: (i) for each cell $c \in \mathcal{C}_r$ repaired to the original value in $\mathcal{C}_p$, the score was 1; (ii) for each cell $c \in \mathcal{C}_r$ changed into a value different from the one in $\mathcal{C}_p$, the score was 0; (iii) for each cell $c \in \mathcal{C}_r$ repaired to a variable value, if the cell was also in $\mathcal{C}_p$, the score was 0.5. In essence, a llun or a variable is counted as a partially correct change. This gives an estimate of precision and recall when variables are considered as a partial match.

Since our scenarios may require a consistent number of variables, due to the need for backward repairs, and this metric disfavors variables, we also adopt a different metric, which counts all correctly identified cells. In this metric, called *Metric 1.0*, item (iii) above becomes: for each cell $c \in \mathcal{C}_r$ repaired to a variable value, if the cell was also in $\mathcal{C}_p$, the score was 1.

Whenever an algorithm returned more than one repair for a database, we calculated P, R, and F for each repair; in the graphs, we report the maximum, minimum, and average values.

Quality Figure 4 shows quality and scalability results. We start by showing that LLUNATIC produces repairs of significantly higher quality with respect to those produced by previous algorithms. We ran LLUNATIC with the cost managers listed above, and the three competing algorithms on samples of the HOSPITAL dataset with increasing size (5k to 25k tuples) and increasing percentage of errors (1% to 5%). We do not report values for the LLUNATIC-FR-S50 cost manager, since they differ for less than one percentage point from those of LLUNATIC-FR-S10.

The maximum F-measure for Metric 1 is in Figure 4.(a); for the two algorithms that return more than one solution, the minimum and average F-measures are reported in Figure 4.(b). The maximum F-measure for Metric 0.5 is in Figure 4.(c). Quality results for algorithms MIN. COST, VERTEX COVER, and REP. SAMPLING are consistent with those reported in [7], which also conducted a comparison of these three algorithms on scenarios in which left and right-hand-side repairs were necessary.

It is not surprising that the F-measure in these cases is quite low. Consider, in fact, a relation $R(A, B)$ with FD $A \rightarrow B$ and a tuple $R(a, 1)$; suppose the first cell is changed to introduce an error, so that the tuple becomes $R(x, 1)$. There are many cases in which this error is not fixed by repairing algorithms. This happens, in fact, whenever the new tuple, $R(x, 1)$, does not get involved in any conflict, and therefore the error goes undetected. In addition, even if a violation is raised, an algorithm may choose to repair the right-hand side of the dependency, thus missing the correct repair. Finally, even when a left-hand-side repair is correctly identified, algorithms have no clue about the right value for the A attribute, and may do little more than introducing a variable – a llun in our case – to fix the violation. All of these cases contribute to lower precision and recall.

The superior quality achieved by LLUNATIC variants can be explained by first noticing that algorithms capable of repairing both right and left-hand sides of dependencies obtained better results than those that only perform forward repairs. Besides LLUNATIC, the only other algorithm capable of backward repairs is SAMPLING. However, this algorithm picks up repairs in a random way. On the contrary, LLUNATIC's chase algorithm explores the space of solutions is a more systematic way, and this explains its improvements in quality.

Figures 4.(d) reports results for the CUSTOMERS scenario. Recall that LLUNATIC is the first system that is able to handle such kind of scenarios with complex constraints. We notice that quality results are better than those on HOSPITAL; this is a consequence of the clear user-specified preference rules.

Scalability The trade-offs between quality and scalability are shown in Figures 4.(e) and 4.(f). Figure 4.(e) compares execution times for the various algorithms on the HOSPITAL scenario up to 100K tuples, with 1% perturbation. Recall that LLUNATIC is the first DBMS-based implementation of a data repairing algorithm. Therefore, our implementation is somehow disfavored in this comparison. To see this, consider that, when producing repairs, main-memory algorithms may aggressively use hash-based data structures to speed-up the computation of repairs, at the cost of using more memory. Using the DBMS, our algorithm is constrained to use SQL for accessing and repairing data; to see how this changes the cost of a repair, consider that even updating a single cell (a very quick operation when performed in main memory) when using the DBMS requires to perform an UPDATE, and therefore a SELECT to locate the right tuple.

Nevertheless, the LLUNATIC-FR-S1 cost manager scales nicely and had better performances than some of the main memory implementations. We may therefore say that graphs (c) and (e) in Figure 4 give us a concrete perception of the trade-offs between complexity and accuracy, and allow us to say that the LLUNATIC-FR-S1 is the best compromise for the HOSPITAL scenario. Other algorithms do not allow to fine tune this trade-off. To see an example, consider the REP. SAMPLING algorithm: we noticed that taking 1000 samples instead of 500 doubles execution times, but it does not produce significant improvements in quality.

Figure 4.(f) clearly shows the benefits that come with a DBMS implementation wrt main-memory ones, namely the possibility of scaling up to very large databases. While previous works [8, 7] have reported results up to a few thousand tuples, we were able to investigate the performance of the system on databases of millions of tuples. The figure shows that LLUNATIC scales in both scenarios to large databases. For the HOSPITAL scenario we replicated the original dataset ten times with 1% errors. In these cases, execution times in the order of the hours for millions of tuples can be considered as a remarkable result, since no system had been able to achieve them before on problems of such exponential complexity. It is interesting to note that performances were significantly better on the CUSTOMERS scenario. This is not surprising: as we discussed above, the CUSTOMERS database contains non redundant, normalize tables. In fact, this clearly shows the benefit of a constraint language that allows to express inter-table cleaning constraints.

It is also worth noting that storing chase trees as delta databases is crucial in order to achieve such a level of scalability. Without such a representation system times would be orders of magnitude higher.

10. RELATED WORK

Several classes of constraints have been proposed to characterize and improve the quality of data (see [13, 15] for surveys). Most relevant to this paper are the (semi-)automated repairing algorithms for these constraints [7, 8, 10, 18, 19, 23]. These methods differ in the constraints that they admit, e.g., FDs [7, 8], CFDs [10, 23], inclusion dependencies [8], and editing rules [18], and the underlying techniques used to improve their effectiveness and efficiency, e.g., statistical inference [10], measures of the reliability of the data [8, 18], and user interaction [10, 25].

All of these methods work for a specific class of constraints only, with the exception of [19, 9]. These works explore the interaction among different kinds of dependencies, but they do not have a unified formal semantics with a definition of solution, neither the generality of our partial order to model preferences.

In industrial settings, most data quality related tasks are executed with ETL tools (e.g, Talend, and Informatica PowerCenter). These systems are employed for data transformations and have low-level modules for specific data quality tasks, such as verification of addresses and phone numbers. More complex operations are also partially available, but lack the support for constraints.

We do allow for forward and backward chasing. Similarly, [10, 23, 7] resolve violations by changing values for attributes in both the premise and conclusion of constraints. They do, however, only support a limited class of constraints. Previous works [23, 7] have used variables in order to repair the left-hand side of dependencies. With respect to variables, our lluns are a more sophisticated tool. In our approach, the full power of lluns is achieved in conjunction with cell-groups: for each llun, the corresponding cell group provides complete provenance data for the llun, both in terms of target and source cells. Therefore, it represents an ideal support for user intervention, when the value of the llun must be resolved to some constant. In fact, lluns and cell-groups can be seen as a novel representation system [22] for solutions, that stands in between of the naive tables of data exchange, and of the more expressive c-tables, trying to strike a balance between complexity and expressibility.

An approach similar to ours has been proposed in [6], with respect to a different cleaning problem. The authors concentrate on scenarios with matching dependencies and matching functions, where the main goal is to merge together values based on attribute similarities, and develop a chase-based algorithm. They show that, under proper assumptions, matching functions provide a partial order over database values, and that the partial order can be lifted to database instances and repairs. A key component of their approach is the availability of matching functions that are essentially total, i.e., they are able to merge any two comparable values. In fact, the problem they deal with can be seen as an instance of the entity-resolution problem. In this paper, we deal with the different problem of data-repairing under a large class of data-cleaning constraints, and have a more ambitious goal, i.e., to embed different forms of value preference into a general semantics for the cleaning process. Our main intuition is that the notion of a partial order is an effective way to let users specify value preferences, and to incorporate them into the semantics in a principled way. In order to do this, we have shown that reasoning on the ordering of values – as in [6] – or on the ordering of single cells is not enough. On the contrary, it is necessary to devise a more sophisticated notion of a partial order for cell-groups, i.e., groups of cells that need to be repaired together and for which lineage information is maintained. Also, we do not make strong assumptions about the possibility of resolving all conflicts among values in the database, and therefore introduce lluns as a third category of values besides nulls and constants.

A comparison of the features supported by existing methods and our repairing method is given in Table 1. We believe that this work makes a concrete step forward towards the goal of developing a uniform formalism for data cleaning, and may stimulate further research on this subject. With a similar spirit, [11] has developed a unifying view of previous approaches by abstracting different classes of constraints with respect to a different problem, that of query answering over inconsistent data.

Our framework can be seen as an extension of the data exchange setting [12]. With respect to the standard chase algorithms for egds, our chase *always terminates* and *never fails*, by leveraging the partial order. We are not aware of any extension of the data exchange setting that allows the introduction of special values (like lluns) to avoid failing chase computations. In fact, we are currently extending our formalism to accommodate for *mapping and cleaning* scenarios, in such a way to maintain the results from the data exchange literature and enlarge them to data repairing.

11. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and Simple Relational Processing of Uncertain Data. In *ICDE*, pages 983–992, 2008.

[3] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, pages 68–79, 1999.

[4] C. Beeri and M. Vardi. A Proof Procedure for Data Dependencies. *J. of the ACM*, 31(4):718–741, 1984.

[5] L. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool, 2011.

[6] L. Bertossi, S. Kolahi, and L. Lakshmanan. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. In *ICDT*, pages 268–279, 2011.

[7] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3:197–207, 2010.

[8] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.

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

[10] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

[11] T. Eiter, M. Fink, G. Greco, and D. Lembo. Repair Localization for Query Answering from Inconsistent Databases. *ACM TODS*, 33(2):1–51, 2008.

[12] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[13] W. Fan. Dependencies Revisited for Improving Data Quality. In *PODS*, pages 159–170, 2008.

[14] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDB J.*, 20(4):495–520, 2011.

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

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

[17] W. Fan, F. Geerts, and J. Wijsen. Determining the Currency of Data. In *PODS*, pages 71–82, 2011.

[18] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *PVLDB*, 3(1):173–184, 2010.

[19] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction Between Record Matching and Data Repairing. In *SIGMOD*, pages 469–480, 2011.

[20] S. Flesca, F. Furfaro, and F. Parisi. Querying and Repairing Inconsistent Numerical Databases. *TODS*, pages 1–77, 2010.

[21] G. Greco, S. Greco, and E. Zumpano. A Logical Framework for Querying and Repairing Inconsistent Databases. *TKDE*, 15(6):1389–1408, 2003.

[22] T. Imieliński and W. Lipski. Incomplete Information in Relational Databases. *J. of the ACM*, 31(4):761–791, 1984.

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

[24] D. Loshin. *Master Data Management*. Knowl. Integrity, Inc., 2009.

[25] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing

Xu Chu[1*] John Morcos[1*] Ihab F. Ilyas[1*]
Mourad Ouzzani[2] Paolo Papotti[2] Nan Tang[2] Yin Ye[2]
[1]University of Waterloo [2]Qatar Computing Research Institute
{x4chu,jmorcos,ilyas}@uwaterloo.ca {mouzzani,ppapotti,ntang,yye}@qf.org.qa

ABSTRACT

Classical approaches to clean data have relied on using integrity constraints, statistics, or machine learning. These approaches are known to be limited in the cleaning accuracy, which can usually be improved by consulting master data and involving experts to resolve ambiguity. The advent of knowledge bases (KBs), both general-purpose and within enterprises, and crowdsourcing marketplaces are providing yet more opportunities to achieve higher accuracy at a larger scale. We propose KATARA, a knowledge base and crowd powered data cleaning system that, given a table, a KB, and a crowd, interprets table semantics to align it with the KB, identifies correct and incorrect data, and generates top-k possible repairs for incorrect data. Experiments show that KATARA can be applied to various datasets and KBs, and can efficiently annotate data and suggest possible repairs.

1. INTRODUCTION

A plethora of data cleaning approaches that are based on integrity constraints [2,7,9,20,36], statistics [30], or machine learning [43], have been proposed in the past. Unfortunately, despite their applicability and generality, they are best-effort approaches that cannot ensure the accuracy of the repaired data. Due to their very nature, these methods do not have enough evidence to precisely identify and update the errors. For example, consider the table of soccer players in Fig. 1 and a functional dependency $B \to C$, which states that B (country) uniquely determines C (capital). This would identify a problem for the four values in tuple t_1 and t_3 over the attributes B and C. A repair algorithm would have to guess which value to change so as to "clean" the data.

To increase the accuracy of such methods, a natural approach is to use external information in tabular master data [19] and domain experts [19,35,40,44]. However, these resources may be scarce and are usually expensive to employ. Fortunately, we are witnessing an increased availability of both general purpose knowledge bases (KBs) such as

[*]Work partially done while interning/working at QCRI.

	A	B	C	D	E	F	G
t_1	Rossi	Italy	Rome	Verona	Italian	Proto	1.78
t_2	Klate	S. Africa	Pretoria	Pirates	Afrikaans	P. Eliz.	1.69
t_3	Pirlo	Italy	Madrid	Juve	Italian	Flero	1.77

Figure 1: A table $\mathcal{T}$ for soccer players

Yago [21], DBpedia [31], and Freebase, as well as special-purpose KBs such as RxNorm[1]. There is also a sustained effort in the industry to build KBs [14]. These KBs are usually well curated and cover a large portion of the data at hand. In addition, while access to an expert may be limited and expensive, crowdsourcing has been proven to be a viable and cost-effective alternative solution.

Challenges. Effectively exploring KBs and crowd in data cleaning raises several new challenges.

(*i*) Matching (dirty) tables to KBs is a hard problem. Tables may lack reliable, comprehensible labels, thus requiring the matching to be executed on the data values. This may lead to ambiguity; more than one mapping may be possible. For example, Rome could be either city, capital, or club in the KB. Moreover, tables usually contain errors. This would trigger problems such as erroneous matching, which will add uncertainty to or even mislead the matching process.

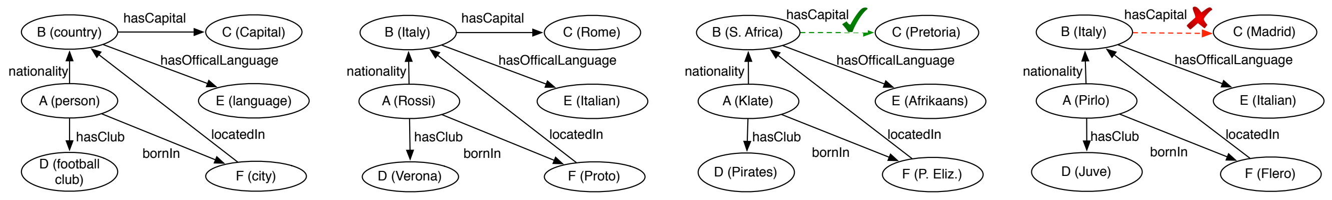
(*ii*) KBs are usually incomplete in terms of the coverage of values in the table, making it hard to find correct table patterns and associate KB values. Since we consider data that could be dirty, it is often unclear, in the case of failing to find a match, whether the database values are erroneous or the KB does not cover these values.

(*iii*) Human involvement is needed to validate matchings and to verify data when the KBs do not have enough coverage. Effectively involving the crowd requires dealing with traditional crowdsourcing issues such as forming easy-to-answer questions for the new data cleaning tasks and optimizing the order of issuing questions to reduce monetary cost.

Despite several approaches for understanding tables with KBs [13,28,39], to the best of our knowledge, they do not explicitly assume the presence of dirty data. Moreover, previous work exploiting reference information for repair has only considered full matches between the tables and the master data [19]. On the contrary, with KBs, partial matches are common due to the incompleteness of the reference.

To this end, we present KATARA, the first data cleaning system that leverages prevalent trustworthy KBs and crowdsourcing for data cleaning. Given a dirty table and a KB,

[1]https://www.nlm.nih.gov/research/umls/rxnorm/



(a) A table pattern φ_s (b) t_1: KB validated (c) t_2: KB & crowd validated (d) t_3: erroneous tuple

Figure 2: Sample solution overview

KATARA first discovers *table patterns* to map the table to an RDF graph obtained from the KB. For instance, consider the table of soccer players in Fig. 1 and the KB Yago. Our table patterns state that the types for columns A, B, and C in the KB are person, country, and capital, respectively, and that two relationships between these columns hold, *i.e.,* A is related to B via nationality and B is related to C via hasCapital. With table patterns, KATARA can annotate tuples as being either correct or incorrect by interleaving KBs and crowdsourcing. For incorrect tuples, KATARA will extract top-k mappings from the KB as possible repairs. In addition, a by-product of KATARA is that data annotated by the crowd as being valid, and which is not found in the KB, provides new facts to enrich the KB. KATARA actively and efficiently involve human experts when automatic approaches cannot capture or face ambiguity, for example, to involve humans to validate patterns discovered, and to involve humans to select from the top-k possible repairs.

Contributions. We built KATARA for annotating and repairing data using KBs and crowd, with the following contributions.

1. *Table pattern definition and discovery.* We propose a new class of table patterns to explain table semantics using KBs (Section 3). Each table pattern is a directed graph, where a node represents a type of a column and a directed edge represents a binary relationship between two columns. We present a new rank-join based algorithm to efficiently discover table patterns with high scores. (Section 4).
2. *Table pattern validation via crowdsourcing.* We devise an efficient algorithm to validate the best table pattern via crowdsourcing (Section 5). To minimize the number of questions, we use an entropy-based scheduling algorithm to maximize the uncertainty reduction of candidate table patterns.
3. *Data annotation.* Given a table pattern, we annotate data with different categories (Section 6): (*i*) correct data validated by the KB; (*ii*) correct data jointly validated by the KB and the crowd; and (*iii*) erroneous data jointly identified by the KB and the crowd. We also devise an efficient algorithm to generate top-k possible repairs for those erroneous data identified in (*iii*).
4. We conducted extensive experiments to demonstrate the effectiveness and efficiency of KATARA using real-world datasets and KBs (Section 7).

2. AN OVERVIEW OF KATARA

KATARA consists of three modules (see Fig. 9 in Appendix): pattern discovery, pattern validation, and data annotation. The *pattern discovery* module discovers table patterns between a table and a KB. The *pattern validation* module uses crowdsourcing to select one table pattern. Using the selected table pattern, the *data annotation* module interacts with the KB and the crowd to annotate data. It also generates possible repairs for erroneous tuples. Moreover, new facts verified by crowd will be used to enrich KBs.

Example 1: Consider a table $\mathcal{T}$ for soccer players (Fig. 1). $\mathcal{T}$ has opaque values for the attributes' labels, thus its semantics is completely unknown. We assume that we have access to a KB $\mathcal{K}$ (*e.g.,* Yago) containing information related to $\mathcal{T}$. KATARA works in the following steps.

(1) Pattern discovery. KATARA first discovers table patterns that contain the types of the columns and the relationships between them. A table pattern is represented as a labelled graph (Fig. 2(a)) where a node represents an attribute and its associated type, *e.g.,* "C (capital)" means that the type of attribute C in KB $\mathcal{K}$ is capital. A directed edge between two nodes represents the relationship between two attributes, *e.g.,* "B hasCapital C" means that the relationship from B to C in $\mathcal{K}$ is hasCapital. A column could have multiple candidate types, *e.g.,* C could also be of type city. However, knowing that the relationship from B to C is hasCapital indicates that capital is a better choice. Since KBs are often incomplete, the discovered patterns may not cover all attributes of a table, *e.g.,* attribute G of table $\mathcal{T}$ is not captured by the pattern in Fig. 2(a).

(2) Pattern validation. Consider a case where pattern discovery finds two similar patterns: the one in Fig. 2(a), and its variant with type location for column C. To select the best table pattern, we send the crowd the question "*Which type (*capital *or* location*) is more accurate for values (*Rome, Pretoria *and* Madrid*)?*". Crowd answers will help choose the right pattern.

(3) Data annotation. Given the pattern in Fig. 2(a), KATARA annotates a tuple with the following three labels:

(i) *Validated by the* KB. By mapping tuple t_1 in table $\mathcal{T}$ to $\mathcal{K}$, KATARA finds a full match, shown in Fig. 2(b) indicating that Rossi (resp. Italy) is in $\mathcal{K}$ as a person (resp. country), and the relationship from Rossi to Italy is nationality. Similarly, all other values in t_1 *w.r.t.* attributes A-F are found in $\mathcal{K}$. We consider t_1 to be correct *w.r.t.* the pattern in Fig. 2(a) and only to attributes A-F.

(ii) *Jointly validated by the* KB *and the crowd.* Consider t_2 about Klate, whose explanation is depicted in Fig. 2(c). In $\mathcal{K}$, KATARA finds that S. Africa is a country, and Pretoria is a capital. However, the relationship from S. Africa to Pretoria is missing. A positive answer from the crowd to the question "*Does* S. Africa hasCapital Pretoria*?*" completes the missing mapping. We consider t_2 correct and generate a new fact "S. Africa hasCapital Pretoria".

(iii) *Erroneous tuple.* For tuple t_3, there is also no link from Italy to Madrid in $\mathcal{K}$ (Fig. 2(d)). A negative answer from the crowd to the question "*Does* Italy hasCapital Madrid*?*" confirms that there is an error in t_3, At this point, however, we cannot decide which value in t_3 is wrong, Italy or Madrid. KATARA will then extract related evidences from $\mathcal{K}$, such as Italy hasCapital Rome and Spain hasCapital Madrid, and use these evidences to generate a set of possible repairs for this tuple. □

The pattern discovery module can be used to select the more relevant KB for a given dataset. If the module cannot find patterns for a table and a KB, KATARA will terminate.

3. PRELIMINARIES

3.1 Knowledge Bases

We consider knowledge bases (KBs) as RDF-based data consisting of *resources*, whose schema is defined using the Resource Description Framework Schema (RDFS). A *resource* is a unique identifier for a real-word entity. For instance, Rossi, the soccer player, and Rossi, the motorcycle racer, are two different resources. Resources are represented using URIs (Uniform Resource Identifiers) in Yago and DBPedia, and mids (machine-generated ids) in Freebase. A *literal* is a string, date, or number, *e.g.,* 1.78. A *property* (*a.k.a. relationship*) is a binary predicate that represents a relationship between two resources or between a resource and a literal. We denote the property between resource x and resource (or literal) y by $P(x, y)$. For instance, locatedIn(Milan, Italy) indicates that Milan is in Italy.

An RDFS ontology distinguishes between classes and instances. A *class* is a resource that represents a set of objects, *e.g.,* the class of countries. A resource that is a member of a class is called an *instance* of that class. The type relationship associates an instance to a class *e.g.,* type(Italy) = country.

A more specific class c can be specified as a *subclass* of a more general class d by using the statement subclassOf(c, d). This means that all instances of c are also instances of d, *e.g.,* subclassOf(capital, location). Similarly, a property P_1 can be a sub-property of a property P_2 by the statement subpropertyOf(P_1, P_2). Moreover, we assume that the property between an entity and its readable name is labeled with "label", according to the RDFS schema.

Note that an RDF ontology naturally covers the case of a KB without a class hierarchy such as IMDB. Also, a more expressive languages, such as OWL (Web Ontology Language), can offer more reasoning opportunities at a higher computational cost. However, KBs in industry [14] as well as popular ones, such as Yago, Freebase, and DBpedia, use RDFS.

3.2 Table Patterns

Consider a table $\mathcal{T}$ with attributes denoted by A_i. There are two basic semantic annotations on a relational table.

(1) *Type of an attribute A_i.* The type of an attribute is an annotation that represents the class of attribute values in A_i. For example, the type of attribute B in Fig. 1 is country.

(2) *Relationship from attribute A_i to attribute A_j.* The relationship between two attributes is an annotation that represents how A_i and A_j are related through a directed binary relationship. A_i is called the *subject* of the relationship, and A_j is called the *object* of the relationship. For example, the relationship from attribute B to C in Fig. 1 is hasCapital.

Table pattern. A *table pattern* (*pattern* for short) φ of a table $\mathcal{T}$ is a labelled directed graph $G(V, E)$ with nodes V and edges E. Each node $u \in V$ corresponds to an attribute in $\mathcal{T}$, possibly typed, and each edge $(u, v) \in E$ from u to v has a label P, denoting the relationship between two attributes that u and v represent. For a pattern φ, we denote by φ_u a node u in φ, $\varphi_{(u,v)}$ an edge in φ, φ_V all nodes in φ, and φ_E all edges in φ.

We assume that a table pattern is a connected graph. When there exist multiple disconnected patterns, *i.e.,* two table patterns that do not share any common node, we treat them independently. Hence, in the following, we focus on discussing the case of a single table pattern.

Semantics. A tuple t of $\mathcal{T}$ *matches* a table pattern φ containing m nodes $\{v_1, \ldots, v_m\}$ *w.r.t.* a KB $\mathcal{K}$, denoted by $t \models \varphi$, if there exist m distinct attributes $\{A_1, \ldots, A_m\}$ in $\mathcal{T}$ and m resources $\{x_1, \ldots, x_m\}$ in $\mathcal{K}$ such that:

1. there is a one-to-one mapping from A_i (and x_i) to v_i for $i \in [1, m]$;
2. $t[A_i] \approx x_i$ and either type(x_i) = type(v_i) or subclassOf(type(x_i), type(v_i));
3. for each edge (v_i, v_j) in φ_E with property P, there exists a property P' for the corresponding resources x_i and x_j in $\mathcal{K}$ such that $P' = P$ or subpropertyOf(P', P).

Intuitively, if t matches φ, each corresponding attribute value of t maps to a resource r in $\mathcal{K}$ under a domain-specific similarity function ($\approx$), and r is a (sub-)type of the type given in φ (conditions 1 and 2). Moreover, for each property P in a pattern, the property between the two corresponding resources must be P or its sub-properties (condition 3).

Example 2: Consider tuple t_1 in Fig. 1 and pattern φ_s in Fig. 2(a). Tuple t_1 matches φ_s, as in Fig. 2(b), since for each attribute value (*e.g.,* $t_1[A]$ = Rossi and $t_1[B]$ = Italy) there is a resource in $\mathcal{K}$ that has a similar value with corresponding type (person for Rossi and country for Italy) for conditions 1 and 2, and the property nationality holds from Rossi to Italy in $\mathcal{K}$ (condition 3). Similarly, conditions 1–3 hold for other attribute values in t_1. Hence, $t_1 \models \varphi_s$. □

We say that a tuple t of $\mathcal{T}$ *partially matches* a table pattern φ *w.r.t.* $\mathcal{K}$, if at least one of condition 2 and condition 3 holds.

Example 3: Consider t_2 in Fig. 1 and φ_s in Fig. 2(a). We say that t_2 partially matches φ_s, since the property hasCapital from $t_2[B]$ = S. Africa to $t_2[C]$ = Pretoria does not exist in $\mathcal{K}$, *i.e.,* condition 3 does not hold. □

Given a table $\mathcal{T}$, a KB $\mathcal{K}$, and a pattern φ, Fig. 3 shows how KATARA works on $\mathcal{T}$.

(1) *Attributes covered* by $\mathcal{K}$. Attributes A–F in Fig. 1 are covered by the pattern in Fig. 2(a). We consider two cases for the tuples.

(a) *Fully covered by $\mathcal{K}$.* We annotate such tuples as semantically correct relative to φ and $\mathcal{K}$ (Fig. 2(b)).

(b) *Partially covered by $\mathcal{K}$.* We use crowdsourcing to verify whether the non-covered data is caused by the incompleteness of $\mathcal{K}$ (Fig. 2(c)) or by actual errors (Fig. 2(d)).

(2) *Attributes not covered* by $\mathcal{K}$. Attribute G in Fig. 1 is not

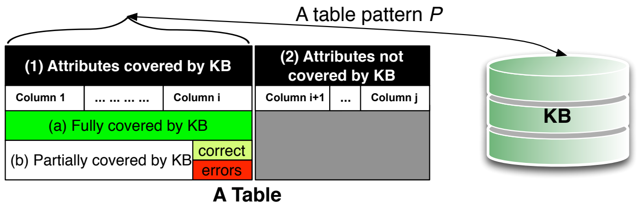


Figure 3: Coverage of a table pattern

covered by the pattern in Fig. 2(a). In this case, KATARA cannot annotate G due to the missing information in $\mathcal{K}$.

For non-covered attributes, we could ask the crowd open-ended questions, such as "*What are the possible relationships between* Rossi *and* 1.78*?*". While approaches have been proposed for open-ended questions to the crowd [38], we leave the problem of extending the structure of the KBs to future work, as discussed in Section 9.

4. TABLE PATTERN DISCOVERY

We first describe candidate types and candidate relationships generation (Section 4.1). We then discuss the scoring to rank table patterns (Section 4.2). We also present a rank-join algorithm to efficiently compute top-k table patterns (Section 4.3) from the candidate types and relationships.

4.1 Candidate Type/Relationship Discovery

We focus on cleaning tabular data for which the schema is either unavailable or unusable. This is especially true for most Web tables and in many enterprise settings where cryptic naming conventions are used. Thus, for table-KB mapping, we use a more general instance based approach that does not require the availability of meaningful column labels. For each column A_i of table $\mathcal{T}$ and for each value $t[A_i]$ of a tuple t, we map this value to several resources in the KB $\mathcal{K}$ whose type can then be extracted. To this end, we issue the following SPARQL query which returns the types and supertypes of entities whose label (*i.e.,* value) is $t[A_i]$.

```
Q_types   select ?c_i
          where {?x_i rdfs:label t[A_i],
                 ?x_i rdfs:type/rdfs:subClassOf* ?c_i}
```

Similarly, the relationship between two values $t[A_i]$ and $t[A_j]$ from a KB $\mathcal{K}$ can be retrieved via the two following SPARQL queries.

```
Q^1_rels   select ?P_ij
           where {?x_i rdfs:label t[A_i], ?x_j rdfs:label t[A_j],
                  ?x_i ?P_ij/rdfs:subPropertyOf* ?x_j}
Q^2_rels   select ?P_ij
           where {?x_i rdfs:label t[A_i],
                  ?x_i ?P_ij/rdfs:subPropertyOf* t[A_j]}
```

Query Q^1_{rels} retrieves relationships where the second attribute is a resource in KBs and Q^2_{rels} retrieves relationships where the second attribute is a literal value, *i.e.,* untyped.

Example 4: In Fig. 1, both Italy and Rome are stored as resources in K, thus their relationship hasCapital would be discovered by Q^1_{rels}; while numerical values such as 1.78 are stored as literals in the KBs, thus the relationship between Rossi and 1.78 would be discovered by query Q^2_{rels}. □

In addition, for two values $t[A_i]$ and $t[A_j]$, we consider them as an ordered pair, thus in total four queries are issued.

Ranking Candidates. We use a normalized version of tf-idf (term frequency-inverse document frequency) [29] to rank the candidate types of a column A_i. We simply consider each cell $t[A_i], \forall t \in \mathcal{T}$, as a query term, and each candidate type T_i as a document whose terms are the entities of T_i in $\mathcal{K}$. The tf-idf score of assigning T_i as the type for A_i is the sum of all tf-idf scores of all cells in A_i:

$$\textsf{tf-idf}(T_i, A_i) = \sum_{t\in\mathcal{T}} \textsf{tf-idf}(T_i, t[A_i])$$

where $\textsf{tf-idf}(T_i, t[A_i]) = \textsf{tf}(T_i, t[A_i]) \cdot \textsf{idf}(T_i, t[A_i])$.

The term frequency $\textsf{tf}(T_i, t[A_i])$ measures how frequently $t[A_i]$ appears in document T_i. Since every type has a different number of entities, the term frequency is normalized by the total number of entities of a type.

$$\textsf{tf}(T_i, t[A_i]) = \begin{cases} 0 & \text{if } t[A_i] \text{ is not of Type } T_i \\ \frac{1}{\log(\text{Number of Entities of Type } T_i)} & \text{otherwise} \end{cases}$$

For example, consider a column with a single cell Italy that has both type Country and type Place. Since there is a smaller number of entities of type Country than that of Place, Country is more likely to be the type of that column.

The inverse document frequency $\textsf{idf}(T_i, t[A_i])$ measures how important $t[A_i]$ is. Under local completeness assumption of KBs [15], if the KB knows about one possible type of $t[A_i]$, the KB should have all possible types of $t[A_i]$. Thus, we define $\textsf{idf}(T_i, t[A_i])$ as follows:

$$\textsf{idf}(T_i, t[A_i]) = \begin{cases} 0 & \text{if } t[A_i] \text{ has no type} \\ \log\frac{\text{Number of Types in } \mathcal{K}}{\text{Number of Types of } t[A_i]} & \text{otherwise} \end{cases}$$

Intuitively, the less the number of types $t[A_i]$ has, the more contribution $t[A_i]$ makes. For example, consider a column that has two cells "Apple" and "Microsoft". Both have Type Company, however, "Apple" has also Type Fruit. Therefore, "Microsoft" being of Type Company says more about the column being of Type Company than "Apple" says about the column being of Type Company.

The tf-idf scores of all candidate types for A_i are normalized to $[0, 1]$ by dividing them by the largest tf-idf score of the candidate type for A_i. The tf-idf score $\textsf{tf-idf}(P_{ij}, A_i, A_j)$ of candidate relationship P_{ij} assigned to column pairs A_i and A_j are defined similarly.

4.2 Scoring Model for Table Patterns

A table pattern contains types of attributes and properties between attributes. The space of all candidate patterns is very large (up to the Cartesian product of all possible types and relationships), making it expensive for human verification. Since not all candidate patterns make sense in practice, we need a meaningful scoring function to rank them and consider only the top-k ones for human validation.

A naive scoring model for a candidate table pattern φ, consisting of type T_i for column A_i and relationship P_{ij} for column pair A_i and A_j, is to simply add up all tf-idf scores of the candidate types and relationships in φ:

$$\textsf{naiveScore}(\varphi) = \Sigma_{i=0}^{m}\textsf{tf-idf}(T_i, A_i) + \Sigma_{ij}\textsf{tf-idf}(P_{ij}, A_i, A_j)$$

However, columns are not independent of each other. The choice of the type for a column A_i affects the choice of the relationship for column pair A_i and A_j, and vice versa.

Example 5: Consider the two columns B and C in Fig. 1. B has candidate types economy, country, and state, C has candidate types city and capital, and B and C have a candidate relationship hasCapital. Intuitively, country as a candi-

Algorithm 1 PDiscovery

```
Input: a table $\mathcal{T}$, a KB $\mathcal{K}$, and a number $k$.
Output: top-$k$ table patterns based on their scores
 1: $\mathsf{types}(A_i) \leftarrow$ get a ranked list of candidate types for $A_i$
 2: $\mathsf{properties}(A_i, A_j) \leftarrow$ get a ranked list of candidate relationships for $A_i$ and $A_j$
 3: Let $\mathcal{P}$ be the top-$k$ table patterns, initialized empty
 4: for all $T_i \in \mathsf{types}(A_i)$, and $P_{ij} \in \mathsf{properties}(A_i, A_j)$ in descending order of tf-idf scores do
 5:     if $|\mathcal{P}| > k$ and TypePruning($T_i$) then
 6:         continue
 7:     generate all table patterns $\mathcal{P}'$ involving $T_i$ or $P_{ij}$
 8:     compute the score for each table pattern $P$ in $\mathcal{P}'$
 9:     update $\mathcal{P}$ using $\mathcal{P}'$
10:     compute the upper bound score $\mathcal{B}$ of all unseen patterns, and let $\varphi_k \in \mathcal{P}$ be the table pattern with lowest score
11:     halt when $\mathsf{score}(\varphi_k) > \mathcal{B}$
12: return $\mathcal{P}$
```

date type for column B is more compatible with hasCapital than economy since capitals are associated with countries, not economies. In addition, capital is also more compatible with hasCapital than city since not all cities are capitals. □

Based on the above observation, to quantify the "compatibility" between a type T and relationship P, where T serves as the type for the resources appearing as *subjects* of the relationship P, we introduce a coherence score $\mathsf{subSC}(T, P)$. Similarly, to quantify the "compatibility" between a type T and relationship P, where T serves as the type for the entities appearing as *objects* of the relationship P, we introduce a coherence scores $\mathsf{objSC}(T, P)$. $\mathsf{subSC}(T, P)$ (resp. $\mathsf{objSC}(T, P)$) measures how likely an entity of Type T appears as a subject (resp. object) of the relationship P.

We use pointwise mutual information (PMI) [10] as a proxy for computing $\mathsf{subSC}(T, P)$ and $\mathsf{objSC}(T, P)$. We use the following notations: $\mathsf{ENT}(T)$ - the set of entities in $\mathcal{K}$ of type T, $\mathsf{subENT}(P)$ - the set of entities in $\mathcal{K}$ that appear in the subject of P, $\mathsf{objENT}(P)$ - the set of entities in $\mathcal{K}$ that appear in the object of P, and $\mathcal{N}$ - the total number of entities in $\mathcal{K}$. We then consider the following probabilities: $\mathsf{Pr}(T) = \frac{|\mathsf{ENT}(T)|}{\mathcal{N}}$, the probability of an entity belonging to T, $\mathsf{Pr_{sub}}(P) = \frac{|\mathsf{subENT}(P)|}{\mathcal{N}}$, the probability of an entity appearing in the subject of P, $\mathsf{Pr_{obj}}(P) = \frac{|\mathsf{objENT}(P)|}{\mathcal{N}}$, the probability of an entity appearing in the object of P, $\mathsf{Pr_{sub}}(P \cap T) = \frac{|\mathsf{ENT}(T) \cap \mathsf{subENT}(P)|}{\mathcal{N}}$, the probability of an entity belonging to type T and appearing in the subject of P, and $\mathsf{Pr_{obj}}(P \cap T) = \frac{|\mathsf{ENT}(T) \cap \mathsf{objENT}(P)|}{\mathcal{N}}$, the probability of an entity belonging to type T and appearing in the object of P. Finally, we can define $\mathsf{PMI_{sub}}(T, P)$:

$$\mathsf{PMI_{sub}}(T, P) = \log \frac{\mathsf{Pr_{sub}}(P \cap T)}{\mathsf{Pr_{sub}}(P)\mathsf{Pr}(T)}$$

The PMI can be normalized into $[-1, 1]$ as follows [3]:

$$\mathsf{NPMI_{sub}}(T, P) = \frac{\mathsf{PMI_{sub}}(T, P)}{-\mathsf{Pr_{sub}}(P \cap T)}$$

To ensure that the coherence score is in $[0, 1]$, we define the *subject semantic coherence* of T for P as

$$\mathsf{subSC}(T, P) = \frac{\mathsf{NPMI_{sub}}(T, P) + 1}{2}$$

The *object semantic coherence* of T for P can be defined similarly.

Example 6: Below are sample coherence scores computed from Yago.

Algorithm 2 TypePruning

```
Input: current top-$k$ table patterns $\mathcal{P}$, candidate type $T_i$.
Output: a boolean value, true/false means $T_i$ can/cannot be pruned
 1: $\mathsf{curMinCohSum}(A_i) \leftarrow$ minimum sum of all coherence scores involving column $A_i$ in current top-$k$ $\mathcal{P}$
 2: $\mathsf{maxCohSum}(A_i, T_i) \leftarrow$ maximum sum of all coherence scores if the type of column $A_i$ is $T_i$
 3: if $\mathsf{maxCohSum}(A_i, T_i) < \mathsf{curMinCohSum}(A_i)$ then
 4:     return true
 5: else
 6:     return false
```

$\mathsf{subSC}(\mathsf{economy}, \mathsf{hasCapital}) = 0.84$
$\mathsf{subSC}(\mathsf{country}, \mathsf{hasCapital}) = 0.86$
$\mathsf{objSC}(\mathsf{city}, \mathsf{hasCapital}) = 0.69$
$\mathsf{objSC}(\mathsf{capital}, \mathsf{hasCapital}) = 0.83$

These scores reflect our intuition in Example 5: country is more suitable than economy to act as a type for the subject resources of hasCapital; and capital is more suitable than city to act as a type for the object resources of hasCapital. □

We now define the *score* of a pattern φ as follows:

$$\mathsf{score}(\varphi) = \Sigma_{i=0}^{m}\mathsf{tf\text{-}idf}(T_i, A_i) + \Sigma_{ij}\mathsf{tf\text{-}idf}(P_{ij}, A_i, A_j)$$
$$+\Sigma_{ij}(\mathsf{subSC}(T_i, P_{ij}) + \mathsf{objSC}(T_j, P_{ij}))$$

4.3 Top-k Table Pattern Generation

Given the scoring model of table patterns, we describe how to retrieve the top-k table patterns with the highest scores without having to enumerate all candidates. We formulate this as a rank-join problem [22]: given a set of sorted lists and join conditions of those lists, the rank-join algorithm produces the top-k join results based on some score function for early termination without consuming all the inputs.

Algorithm. The algorithm, referred as PDiscovery, is given in Algorithm 1. Given a table $\mathcal{T}$, a KB $\mathcal{K}$, and a number k, it produces top-k table patterns. To start, each input list, *i.e.,* candidate types for a column, and candidate relationships for a column pair, is ordered according to the respective tf-idf scores (lines 1-2). When two candidate types (resp. relationships) have the same tf-idf scores, the more discriminative type (resp. relationship) is ranked higher, *i.e.,* the one with less number of instances in $\mathcal{K}$.

Two lists are joined if they agree on one column, *e.g.,* the list of candidate types for A_i is joined with the list of candidate relationships for A_i and A_j. A join result is a candidate pattern φ, and the scoring function is $score(\varphi)$. The rank-join algorithm scans the ranked input lists in descending order of their tf-idf scores (lines 3-4), table patterns are generated incrementally as we move down the input lists. Table patterns that cannot be used to produce top-k patterns will be pruned (lines 5-6). For each join result, *i.e.,* each table pattern φ, the score $\mathsf{score}(\varphi)$ is computed (lines 7-8). We also maintain an upper bound $\mathcal{B}$ of the scores of all unseen join results, *i.e.,* table patterns (line 10). Since each list is ranked, $\mathcal{B}$ can be computed by adding up the support scores of the current positions in the ranked lists, plus the maximum coherence scores a candidate relationship can have with any types. We terminate the join process if either we have exhaustively scanned every input list, or we have obtained top-k table patterns and the score of the k^{th} table pattern is greater than or equal to $\mathcal{B}$ (line 11).

Lines 5-6 in Algorithm 1 check whether a candidate type T_i for column A_i can be pruned without generating table patterns involving T_i by calling Algorithm 2. The intuition behind type pruning (Algorithm 2) is that a candidate type T_i is useful if it is more coherent with any relationship P_{ix} than previously examined types for A_i. We first calculate the current minimum sum of coherence scores involving column A_i in the current top-k patterns, *i.e.,* $\mathsf{curMinCohSum}(A_i)$ (line 1). We then calculate the maximum possible sum of coherence scores involving type T_i, *i.e.,* $\mathsf{maxCohSum}(A_i, T_i)$ (line 2). T_i can be pruned if $\mathsf{maxCohSum}(A_i, T_i) < \mathsf{curMinCohSum}(A_i)$ since any table pattern having T_i as the type for A_i will have a lower score than the scores of the current top-k patterns (lines 3-6).

Example 7: Consider the rank-join graph in Fig. 4 ($k = 2$) for a table with just two columns B and C as in Fig. 1. The tf-idf scores for each candidate type and relationship are shown in the parentheses. The top-2 table patterns φ_1, φ_2 are shown on the top. $score(\varphi_1) = \mathsf{sup}(\mathsf{country}, B) + \mathsf{sup}(\mathsf{capital}, C) + \mathsf{sup}(\mathsf{hasCapital}, B, C) + 5 \times (\mathsf{subSC}(\mathsf{country}, \mathsf{hasCapital}) + \mathsf{objSC}(\mathsf{capital}, \mathsf{hasCapital})) = 1.0 + 0.9 + 0.9 + 0.86 + 0.83 = 4.49$. Similarly, we have $score(\varphi_2) = 4.47$.

Suppose we are currently examining type state for column B. We do not need to generate table patterns involving state since the maximum coherence between state and hasCapital or isLocatedIn is less than the the current minimum coherence score between type of column B and relationship between B and C in the current top-2 patterns.

Suppose we are examining type whole for column C, and we have reached type state for B and hasCapital for relationship B, C. The bound score for all unseen patterns is $\mathcal{B} = 0.7 + 0.5 + 0.9 + 0.86 + 0.83 = 3.78$, where 0.7, 0.9 and 0.5 are the tf-idf scores for state, whole and hasCapital respectively, and 0.86 (resp. 0.83) is the maximum coherence score between any type in $\mathsf{types}(B)$ (resp. $\mathsf{types}(C)$) and any relationship in $\mathsf{properties}(B, C)$. Since $\mathcal{B}$ is smaller than $score(\varphi_2) = 4.47$, we terminate the rank join process. □

Correctness. Algorithm 1 is guaranteed to produce the top-k table patterns since we keep the current top-k patterns in $\mathcal{P}$, and we terminate when we are sure that it will not produce any new table pattern with a higher score. In the worst case, we still have to exhaustively go through all the ranked lists to produce the top-k table patterns. However, in most cases the top ranked table patterns involve only candidate types/relationships with high tf-idf scores, which are at the top of the lists.

Computing coherence scores for a type and a relationship is an expensive operation that requires set intersection. Therefore, for a given $\mathcal{K}$, we compute offline the coherence score for every type and every relationship. For each relationship, we also keep the maximum coherence score it can achieve with any type, to efficiently compute the bound $\mathcal{B}$.

5. PATTERN VALIDATION VIA CROWD

We now study how to use the crowd to validate the discovered table patterns. Specifically, given a set $\mathcal{P}$ of candidate patterns, a table $\mathcal{T}$, a KB $\mathcal{K}$, and a crowdsourcing framework, we need to identify the most appropriate pattern for $\mathcal{T}$ *w.r.t.* $\mathcal{K}$, with the objective of minimizing the number of crowdsourcing questions. We assume that the crowd workers are experts in the semantics of the reference KBs, *i.e.,* they can verify if values in the tables fit into the KBs.

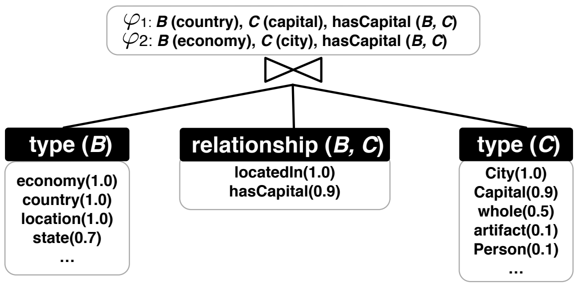


Figure 4: Encoding top-k as a rank-join

5.1 Creating Questions for the Crowd

A naive approach to generate crowdsourcing questions is to express each candidate table pattern as a whole in a single question to the crowd who would then select the best one. However, table pattern graphs can be hard for crowd users to understand (*e.g.,* Fig. 2(a)). Also, crowd workers are known to be good at answering simple questions [41]. A practical solution is to decompose table patterns into simple tasks: (1) type validation, *i.e.,* to validate the type of a column in the table pattern; and (2) binary relationship validation, *i.e.,* to validate the relationship between two columns.

Column type validation. Given a set of candidate types $\mathsf{candT}(A_i)$ for column A_i, one type $T_i \in \mathsf{candT}(A_i)$ needs to be selected. We formulate the following question to the crowd about the type of a column: *What is the most accurate type of the highlighted column?*; along with k_t randomly chosen tuples from $\mathcal{T}$ and all candidate types from $\mathsf{candT}(A_i)$. A sample question is given as follows.

Q_1 : *What is the most accurate type of the highlighted column?*
(A, B, C, D, E, F, ...)
(Rossi, Italy, Rome, Verona, Italian, Proto, ...)
(Pirlo, Italy, Madrid, Juve, Italian, Flero,, ...)
○ country ○ economy
○ state ○ none of the above

After q questions are answered by the crowd workers, the type with the highest support from the workers is chosen.

Crowd workers, even if experts in the reference KB, are prone to mistakes when $t[A_i]$ in tuple t is ambiguous, *i.e.,* $t[A_i]$ belongs to multiple types in $\mathsf{candT}(A_i)$. However, this is mitigated by two observations: (i) it is unlikely that all values are ambiguous and (ii) the probability of providing only ambiguous values diminishes quickly with respect to the number of values. Consider two types T_1 and T_2 in $\mathsf{candT}(A_i)$, the probability that randomly selected entities belong to both types is $p = \frac{|ENT(T_1) \cap ENT(T_2)|}{|ENT(T_1) \cup ENT(T_2)|}$. After q questions are answered, the probability that all $q \cdot k_t$ values are ambiguous is $p^{q \cdot k_t}$. Suppose $p = 0.8$, a very high for two types in K, and five questions are asked with each question containing five tuples, *i.e.,* $q = 5, k_t = 5$, the probability $p^{q \cdot k_t}$ becomes as low as 0.0038.

For each question, we also expose some contextual attribute values that help workers better understand the question. For example, we expose the values for A, C, D, E in question Q_1 when validating the type of B. If the the number of attributes is small, we show them all; otherwise, we use off-the-shelf technology to identify attributes that are related to the ones in the question [23]. To mitigate the risk of workers making mistakes, each question is asked three times, and the majority answer is taken. Indeed, our empirical study in Section 7.2 shows that five questions are enough to pick the correct type in all the datasets we experimented.

Relationship validation. We validate the relationship for column pairs in a similar fashion, with an example below.

Q_2 : *What is the most accurate relationship for* highlighted columns (A, B, C , D, E, F, ...)
(Rossi, Italy, Rome , Verona, Italian, Proto, ...)
(Pirlo, Italy, Madrid , Juve, Italian, Flero, ...)
○ B hasCapital C ○ C locatedIn B ○ none of the above

Candidate types and candidate relationships are stored as URIs in KBs; thus not directly consumable by the crowd workers. For example, the type capital is stored as http://yago-knowledge.org/resource/wordnet_capital_10851850, and the relationship hasCapital is stored as http://yago-knowledge.org/resource/hasCapital. We look up type and relationship descriptions, *e.g.,* capital and hasCapital, by querying the KB for the labels of the corresponding URIs. If no label exists, we process the URI itself by removing the text before the last slash and punctuation symbols.

5.2 Question Scheduling

We now turn our attention to how to minimize the total number of questions to obtain the correct table pattern by scheduling which column and relationship to validate first.

Note that once a type (resp. relationship) is validated, we can prune from $\mathcal{P}$ all table patterns that have a different type (resp. relationship) for that column (resp. column pair). Therefore, a natural choice is to choose those columns (resp. column pairs) with the maximum uncertainty reduction [45].

Consider φ as a variable, which takes values from $\mathcal{P} = \{\varphi_1, \varphi_2, \dots, \varphi_k\}$. We translate the score associated with each table pattern to a probability by normalizing the scores, *i.e.,* $Pr(\varphi = \varphi_i) = \frac{score(\varphi_i)}{\Sigma_{\varphi_j \in \mathcal{P}} score(\varphi_j)}$. Our translation from scores to probabilities follows the general framework of interpreting scores in [25]. Specifically, our translation is rank-stable, *i.e.,* for two patterns φ_1 and φ_2, if $score(\varphi_1) > score(\varphi_2)$, then $Pr(\varphi = \varphi_1) > Pr(\varphi = \varphi_2)$.

We define the uncertainty of φ *w.r.t.* $\mathcal{P}$ as the entropy.

$$H_{\mathcal{P}}(\varphi) = -\Sigma_{\varphi_i \in \mathcal{P}} Pr(\varphi = \varphi_i) \log_2 Pr(\varphi = \varphi_i)$$

Example 8: Consider an input list of five table patterns $\mathcal{P} = \{\varphi_1, \dots, \varphi_5\}$ as follows with the normalized probability of each table pattern shown in the last column.

	type (B)	type (C)	$P(B,C)$	score	prob
φ_1	country	capital	hasCapital	2.8	0.35
φ_2	economy	capital	hasCapital	2	0.25
φ_3	country	city	locatedIn	2	0.25
φ_4	country	capital	locatedIn	0.8	0.1
φ_5	state	capital	hasCapital	0.4	0.05

□

We use variables v_{A_i} and $v_{A_iA_j}$ to denote the type of the column A_i and the relationship between A_i and A_j respectively. The set of all variables is denoted as V. In Example 8, $V = \{v_B, v_C, v_{BC}\}$, $v_B \in \{\text{country}, \text{economy}, \text{state}\}$, $v_C \in \{\text{capital}, \text{city}\}$ and $v_{BC} \in \{\text{hasCapital}, \text{isLocatedIn}\}$. The probability of an assignment of a variable v to a is obtained by aggregating the probability of those table patterns that have that assignment for v. For example, $Pr(v_B = \text{country}) = Pr(\varphi_1) + Pr(\varphi_3) + Pr(\varphi_4) = 0.35 + 0.25 + 0.1 = 0.7$, $Pr(v_B = \text{economy}) = 0.25$, and $Pr(v_B = \text{state}) = 0.05$.

After validating a variable v to have value a, we remove

Algorithm 3 PatternValidation

```
Input: a set of table patterns $\mathcal{P}$
Output: one table pattern $\varphi \in \mathcal{P}$
 1: $\mathcal{P}_{re}$ be the remaining table patterns, initialized $\mathcal{P}$
 2: initialize all variables $V$, representing column or column pairs,
    and calculate their probability distributions.
 3: while $|\mathcal{P}_{re}| > 1$ do
 4:   $E_{best} \leftarrow 0$
 5:   $v_{best} \leftarrow null$
 6:   for all $v \in V$ do
 7:     compute the entropy $H(v)$.
 8:     if $H(v) > E_{best}$ then
 9:       $v_{best} \leftarrow v$
10:       $E_{best} \leftarrow H(v)$
11:   validate the variable $v$, suppose the result is $a$, let $\mathcal{P}_{v=a}$
      to be the set of table patterns with $v = a$
12:   $\mathcal{P}_{re} = \mathcal{P}_{v=a}$
13:   normalize the probability distribution of patterns in $\mathcal{P}_{re}$.
14: return the only table pattern $\varphi$ in $\mathcal{P}_{re}$
```

from $\mathcal{P}$ those patterns that have different assignment for v. The remaining patterns are denoted as $\mathcal{P}_{v=a}$. Suppose column B is validated to be of type country, then $\mathcal{P}_{v_B=\text{country}} = \{\varphi_1, \varphi_3, \varphi_4\}$. Since we do not know what value a variable can take, we measure the expected reduction of uncertainty of variable φ after validating variable v, formally defined as:

$$E(\Delta H(\varphi))(v) = \Sigma_a Pr(v = a) H_{\mathcal{P}_{v=a}}(\varphi) - H_{\mathcal{P}}(\varphi)$$

In each iteration, we choose the variable v (column or column pair) with the maximum uncertainty reduction, *i.e.,* $E(\Delta H(\varphi))(v)$. Each iteration has a complexity of $O(|V||\mathcal{P}|^2)$ because we need to examine all $|V|$ variables, each variable could take $|\mathcal{P}|$ values, and calculating $H_{\mathcal{P}_{v=a}}(\varphi)$ for each value also takes $O(|\mathcal{P}|)$ time. The following theorem simplifies the calculation for $E(\Delta H(v))$ with a complexity of $O(|V||\mathcal{P}|)$.

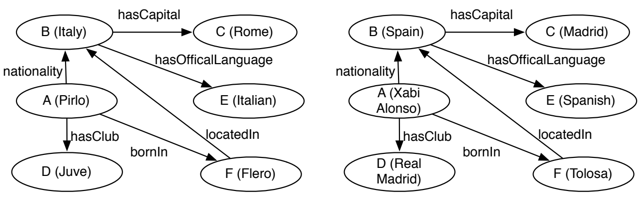
Theorem 1. *The expected uncertainty reduction after validating a column (column pair) v is the same as the entropy of the variable.* $E(\Delta H(\varphi))(v) = H(v)$*, where* $H(v) = -\Sigma_a Pr(v = a) \log_2 Pr(v = a)$.

The proof of Theorem 1 can be found in Appendix A. Algorithm 3 describes the overall procedure for pattern validation. At each iteration: (1) we choose the best variable v_{best} to validate next based on the expected reduction of uncertainty of φ (lines 4-10); (2) we remove from $\mathcal{P}_{re}$ those table patterns that have a different assignment for variable v than the validated value a (lines 11-12); and (3) we renormalize the probability distribution of the remaining table patterns in $\mathcal{P}_{re}$ (line 13). We terminate when we are left with only one table pattern (line 3).

Example 9: To validate the five patterns in Example 8, we first calculate the entropy of every variable. $H(v_B) = -0.7 \log_2 0.7 - 0.25 log_2 0.25 - 0.05 log_2 0.05 = 1.07$, $H(v_C) = 0.81$, and $H(v_{BC}) = 0.93$. Thus column B is validated first, say the answer is country. The remaining set of table patterns, and their normalized probabilities are:

	type (B)	type (C)	$P(B,C)$	prob
φ_1	country	capital	hasCapital	0.5
φ_3	country	city	locatedIn	0.35
φ_4	country	capital	locatedIn	0.15

Now $\mathcal{P}_{re} = \{\varphi_1, \varphi_3, \varphi_4\}$. The new entropies are: $H(v_B) = 0$, $H(v_C) = 0.93$ and $H(v_{BC}) = 1$. Therefore, column pair



(a) Possible repair G_1 (b) Possible repair G_2

Figure 5: Sample instance graphs

B, C is chosen, say the answer is hasCapital. We are now left with only one pattern φ_1, thus we return it. □

In Example 9, we do not need to validate v_C following our scheduling strategy. Furthermore, after validating certain variables, other variables may become less uncertain, thus requiring a smaller number of questions to validate.

6. DATA ANNOTATION

In this section, we describe how KATARA annotates data (Section 6.1). We also discuss how to generate possible repairs for identified errors (Section 6.2).

6.1 Annotating Data

KATARA annotates tuples as correct data validated by KBs, correct data jointly validated by KBs and the crowd, or data errors detected by the crowd, using the following two steps.

Step 1: Validation by KBs. For each tuple t and pattern φ, KATARA issues a SPARQL query to check whether t is fully covered by a KB $\mathcal{K}$. If it is fully covered, KATARA annotates it as a correct tuple validated by KB (case (*i*)). Otherwise, it goes to step 2.

Step 2: Validation by KBs and Crowd. For each node (*i.e.,* type) and edge (*i.e.,* relationship) that is missing from $\mathcal{K}$, KATARA asks the crowd whether the relationship holds between the given two values. If the crowd says yes, KATARA annotates it as a correct tuple, jointly validated by KB and crowd (case (*ii*)). Otherwise, it is certain that there exist errors in this tuple (case (*iii*)).

Example 10: Consider tuple t_2 (resp. t_3) in Fig. 1 and the table pattern in Fig. 2(a). The information about whether Pretoria (resp. Madrid) is a capital of S. Africa (resp. Italy) is not in KB. To verify this information, we issue a boolean question Q_{t_2} (resp. Q_{t_3}) to the crowd as:

Q_{t_2} :*Does* S. Africa hasCapital Pretoria*?*
◯ Yes ◯ No

Q_{t_3} :*Does* Italy hasCapital Madrid*?*
◯ Yes ◯ No

In such case, the crowd will answer **yes** (resp. **no**) to question Q_{t_2} (resp. Q_{t_3}). □

Knowledge base enrichment. Note that, in step 2, for each affirmative answer from the crowd (*e.g.,* Q_{t_2} above), a new fact that is not in the current KB is created. KATARA collects such facts and uses them to enrich the KB.

6.2 Generating Top-k Possible Repairs

We start by introducing two notions that are necessary to explain our approach for generating possible repairs.

Instance graphs. Given a KB $\mathcal{K}$ and a pattern $G(V, E)$, an *instance graph* $G_I(V_I, E_I)$ is a graph with nodes V_I and edges E_I, such that (i) each node $v_i \in V_I$ is a resource in $\mathcal{K}$; (ii) each edge $e_i \in E_I$ is a property in $\mathcal{K}$; (iii) there is a one-to-one correspondence f from each node $v \in V$ to a node $v_i \in V_I$, *i.e.,* $f(v) = v_i$; and (iv) for each edge $(u, v) \in E$, there is an edge $(f(u), f(v)) \in E_I$ with the same property. Intuitively, an instance graph is an instantiation of a pattern in a given KB.

Example 11: Figures 5(a) and 5(b) are two instance graphs of the table pattern of Fig. 2(a) in Yago for two players. □

Repair cost. Given an instance graph G, a tuple t, and a table pattern φ, the *repair cost* of aligning t to G *w.r.t.* φ, denoted by $\mathsf{cost}(t, \varphi, G) = \sum_{i=1}^{n} c_i$, is the cost of changing values in t to align it with G, where c_i is the cost of the i-th change and n the number of changes in t. Intuitively, the less a repair cost is, the closer the updated tuple is to the original tuple, hence more likely to be correct. By default, we set $c_i = 1$. The cost can also be weighted with confidences on data values [18]. In such case, the higher the confidence value is, the more costly the change is.

Example 12: Consider tuple t_3 in Fig. 1, the table pattern φ_s in Fig. 2(a), and two instance graphs G_1 and G_2 in Fig. 5. The repair cost to update t_3 to G_1 is 1, *i.e.,* $\mathsf{cost}(t_3, \varphi_s, G_1)$ = 1, by updating $t_3[C]$ from Madrid to Rome. Similarly, the repair cost from t_3 to G_2 is 5, *i.e.,* $\mathsf{cost}(t_3, \varphi_s, G_2) = 5$. □

Note that the possible repairs are ranked based on repair cost in ascending order. We provide top-k possible repairs and we leave it to the users (or crowd) to pick the most appropriate repair. In the following, we describe algorithms to generate top-k repairs for each identified erroneous tuple.

Given a KB $\mathcal{K}$ and a pattern φ, we compute all instance graphs $\mathcal{G}$ in $\mathcal{K}$ *w.r.t.* φ. For each tuple t, a naive solution is to compute the distance between t and each graph G in $\mathcal{G}$. The k graphs with smallest repair cost are returned as top-k possible repairs. Unfortunately, this is too slow in practice.

A natural way to improve the naive solution for top-k possible repair generation is to retrieve only instance graphs that can possibly be repairs, *i.e.,* the instance graphs whose values have an overlap with a given erroneous tuple. We leverage inverted lists to achieve this goal.

Inverted lists. Each inverted list is a mapping from a key to a posting list. A *key* is a pair (A, a) where A is an attribute and a is a constant value. A *posting* list is a set $\mathcal{G}$ of graph instances, where each $G \in \mathcal{G}$ has value a on attribute A.

For example, an inverted list *w.r.t.* G_1 in Fig. 5(a) is as:

$$\boxed{\text{country, Italy}} \rightarrow \boxed{G_1}$$

Algorithm. The optimized algorithm for a tuple t is given in Algorithm 4. All possible repairs are initialized (line 1) and instantiated by using inverted lists (lines 2-3). For each

Algorithm 4 TOP-k repairs

```
Input: a tuple $t$, a table pattern $\varphi$, and inverted lists $\mathcal{L}$
Output: top-$k$ repairs for $t$
1: $\mathcal{G}_t = \emptyset$
2: for each attribute $A$ in $\varphi$ do
3:     $\mathcal{G}_t = \mathcal{G}_t \cup \mathcal{L}(A, t[A])$
4: for each $G$ in $\mathcal{G}_t$ do
5:     compute $\mathsf{cost}(t, \varphi, G)$
6: return top-$k$ repairs in $\mathcal{G}_t$ with least cost values
```

possible repair, its repair cost *w.r.t.* t is computed (lines 4-5), and top-k repairs are returned (line 6).

Example 13: Consider t_3 in Fig. 1 and pattern φ_s in Fig. 2(a). The inverted lists retrieved are given below.

A, Pirlo	$\rightarrow \boxed{G_1}$	D, Juve	$\rightarrow \boxed{G_1}$
B, Italy	$\rightarrow \boxed{G_1}$	E, Italian	$\rightarrow \boxed{G_1}$
C, Madrid	$\rightarrow \boxed{G_2}$	F, Flero	$\rightarrow \boxed{G_1}$

It is easy to see that the occurrences of instance graphs G_1 and G_2 are 5 and 1, respectively. In other words, the cost of repairing t_3 *w.r.t.* G_1 is $6 - 5 = 1$ and *w.r.t.* G_2 is $6 - 1 = 5$. Hence, the top-1 possible repair for t_3 is G_1. □

The practicability of possible repairs of KATARA depends on the coverage of KBs, while existing automatic data repairing techniques usually require certain redundancy in the data to perform well. KATARA and existing techniques complement each other, as demonstrated in Section 7.4.

7. EXPERIMENTAL STUDY

We evaluated KATARA using real-life data along four dimensions: (*i*) the effectiveness and efficiency of table pattern discovery (Section 7.1); (*ii*) the efficiency of pattern validation via the expert crowd (Section 7.2); (*iii*) the effectiveness and efficiency of data annotation (Section 7.3); and (*iv*) the effectiveness of possible repairs (Section 7.4).

Knowledge bases. We used Yago [21] and DBpedia [27] as the underlying KBs. Both were transformed to Jena format (`jena.apache.org/`) with LARQ (a combination of ARQ and Lucene) support for string similarity. We set the threshold to 0.7 in Lucene to check whether two strings match.

Datasets. We used three datasets: WikiTables and WebTables contains tables from the Web[2] with relatively small numbers of tuples and columns, and RelationalTables contains tables with larger numbers of tuples and columns.

• WikiTables contains 28 tables from Wikipedia pages. The average number of tuples is 32.

• WebTables contains 30 tables from Web pages. The average number of tuples is 67.

• RelationalTables has three tables: Person has personal information joined on the attribute country from two sources: a biographic table extracted from wikipedia [32], and a country table obtained from a wikipedia page[3] resulting in 316K tuples. Soccer has 1625 tuples about soccer players and their clubs scraped from the Web[4]. University has 1357 tuples about US universities with their addresses[5].

All the tables were manually annotated using types and relationships in Yago as well as DBPedia, which we considered as the *ground truth*. Table 1 shows the number of columns that have types, and the number of column pairs that have relationships, using Yago and DBPedia, respectively.

All experiments were conducted on Win 7 with an Intel i7 CPU@3.4Ghz, 20GB of memory, and an SSD 500GB hard disk. All algorithms were implemented in JAVA.

[2]`http://www.it.iitb.ac.in/~sunita/wwt/`

[3]`http://tinyurl.com/qhhty3p`

[4]`www.premierleague.com/en-gb.html`, `www.legaseriea.it/en/`, `www.premierleague.com/en-gb.html`

[5]`ope.ed.gov/accreditation/GetDownLoadFile.aspx`

	Yago		DBPedia	
	#-type	#-relationship	#-type	#-relationship
WikiTables	54	15	57	18
WebTables	71	33	73	35
RelationalTables	14	7	14	16

Table 1: Datasets and KBs characteristics

7.1 Pattern Discovery

Algorithms. We compared four discovery algorithms.

(i) RankJoin - our proposed approach (Section 4).

(ii) Support - a baseline approach that ranks the candidate types and relationships solely on their support scores, *i.e.,* the number of tuples that are of the candidate's types and relationships.

(iii) MaxLike [39] - infers the type of a column and the relationship between a column pair separately using maximum likelihood estimation.

(iv) PGM [28] - infers the type of a column, the relationship between column pairs, and the entities of cells by building a probabilistic graphic model to make holistic decisions.

Evaluation Metrics. A type (relationship) gets a score of 1 if it matches the ground truth, and a partial score $\frac{1}{s+1}$ if it is the super type (relationship) of the ground truth, where s is the number of steps in the hierarchy to reach the ground truth. For example, a label Film for a column, whose actual type is IndianFilm, will get a score of 0.5, since Film is the immediate super type of IndianFilm, *i.e.,* $s = 1$. The precision P of a pattern φ is defined as the sum of scores for all types and relationships in φ over the total number of types and relationships in φ. The recall R of a pattern φ is defined as the sum of scores for all types and relationships in φ over the total number of types and relationships in the ground truth.

Effectiveness. Table 2 shows the precision and recall of the top pattern chosen by four pattern discovery algorithms for three datasets using Yago and DBPedia. We first discuss Yago. (1) Support has the lowest precision and recall in all scenarios, since it selects the types/relationships that cover the most number of tuples, which are usually the general types, such as Thing or Object. (2) MaxLike uses maximum likelihood estimation to select the best type/relationship that maximizes the probability of values given the type/relationship. It performs better than Support, but still chooses types and relationships independently. (3) PGM is a supervised learning approach that requires training and tuning of a number of weights. PGM shows mixed effectiveness results: it performs better than MaxLike on WebTables, but worse on WikiTables and RelationalTables. (4) RankJoin achieves the highest precision and recall due to its tf-idf style ranking, as well as for considering the coherence between types and relationships. For example, consider a table with two columns `actors` and `films` that have a relationship `actedIn`. If most of the values in the `films` column also happen to be `books`, MaxLike will use `books` as the type, since there are fewer instances of `books` than `films` in Yago. However, RankJoin would correctly identify `films` as the type, since it is more coherent with `actedIn` than `books`.

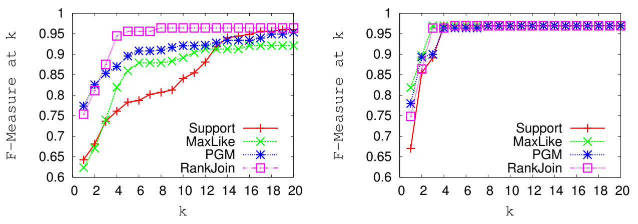
The result from DBPedia, also shown in Table 2, confirms that RankJoin performs best among the four methods. Notice that the precision and recall of all methods are consis-

	Support		MaxLike		PGM		RankJoin	
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
WikiTables	.54	.59	.62	.68	.60	.67	.78	.86
WebTables	.65	.64	.63	.62	.77	.77	.86	.84
RelationalTables	.51	.51	.71	.71	.53	.53	.77	.77
Yago								
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
WikiTables	.56	.70	.71	.89	.61	.77	.71	.89
WebTables	.65	.69	.80	.84	.76	.80	.82	.87
RelationalTables	.64	.67	.81	.86	.74	.77	.81	.86
DBPedia								

Table 2: Pattern discovery precision and recall

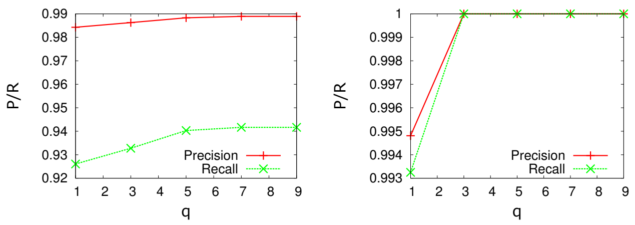
	Support	MaxLike	PGM	RankJoin
WikiTables	153	155	286	153
WebTables	160	177	1105	162
RelationalTables/Person	130	140	13842	127
Person	252	258	N.A.	257
Yago				
WikiTables	50	54	90	51
WebTables	103	104	189	107
RelationalTables/Person	400	574	11047	409
Person	368	431	N.A.	410
DBPedia				

Table 3: Pattern discovery efficiency (seconds)



(a) Yago (b) DBPedia

Figure 6: Top-k F-measure (WebTables)



(a) Yago (b) DBPedia

Figure 7: Pattern validation P/R (WebTables)

tently better using DBPedia than Yago. This is because the number of types in DBPedia (865) is much smaller than that of Yago (374K), hence, the number of candidate types for a column using DBPedia is much smaller, causing less stress for all algorithms to rank them.

To further verify the effectiveness of our ranking function, we report the F-measure F of the top-k patterns chosen by every algorithm. The F value of the top-k patterns is defined as the best value of F from one of the top-k patterns. Figure 6 shows F values of the top-k patterns varying k on WebTables. RankJoin converges faster than other methods on Yago, while all methods converge quickly on DBPedia due to its small number of types. Top-k F-measure results for the other two datasets show similar behavior, and are reported in Appendix B.

Efficiency. Table 3 shows the running time in seconds for all datasets. We ran each test 5 times and report the average time. We separate the discussion of Person from RelationalTables due to its large number of tuples. For Person, we implemented a distributed version of candidate types/relationships generation by distributing the 316K tuples over 30 machines, and all candidates are collected into one machine to complete the pattern discovery. Support, MaxLike, and RankJoin have similar performance in all datasets, because their most expensive operation is the disk I/Os for KBs lookups in generating candidate types and relationships, which is linear *w.r.t.* the number of tuples. PGM is the most expensive due to the message passing algorithms used for the inference of probabilistic graphical model. PGM takes hours on tables with around 1K tuples, and cannot finish within one day for Person.

7.2 Pattern Validation

Given the top-k patterns from the pattern discovery, we need to identify the most appropriate one. We validated the patterns of all datasets using an expert crowd with 10 students. Each question contains five tuples, *i.e.,* $k_t = 5$.

We first evaluated the effect of the number of questions used to validate each variable, which is a type or a relationship, on the quality of the chosen pattern. We measure the precision and recall of the final chosen validation *w.r.t.* the ground truth in the same way as in Section 7.1. Figure 7 shows the average precision and recall of the validated pattern of WebTables while varying the number of questions q per variable. It can be seen that, even with $q = 1$, the precision and recall of the validated pattern is already high. In addition, the precision and recall converge quickly, with $q = 5$ on Yago, and $q = 3$ on DBPedia. Pattern validation results on WikiTables and RelationalTables show a similar behavior, and are reported in Appendix C.

	Yago		DBPedia	
	MUVF	AVI	MUVF	AVI
WikiTables	64	79	88	102
WebTables	81	105	90	118
RelationalTables	24	28	28	36

Table 4: #-variables to validate

To evaluate the savings in crowd pattern validation that are achieved by our scheduling algorithm, we compared our method (denoted MUVF, short for most-uncertain-variable-first) with a baseline algorithm (denoted AVI for all-variables-independent) that validates every variable independently. For each dataset, we compared the number of variables needed to be validated until there is only one table pattern left. Table 4 shows that MUVF performs consistently better than AVI in terms of the number of variables to validate, because MUVF may spare validating certain variables due to scheduling, *i.e.,* some variables become certain after validating some other variables.

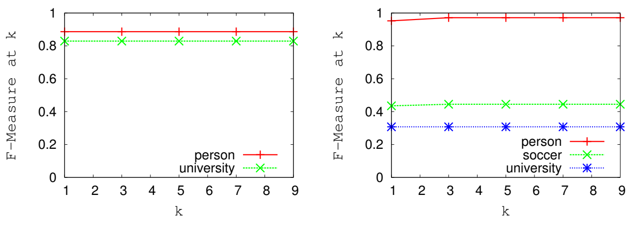
The validated table patterns of RelationalTables for both Yago and DBPedia are depicted in Fig. 10 in the Appendix. All validated patterns are also used in the following experimental study.

7.3 Data Annotation

Given the table patterns obtained from Section 7.2, data values are annotated *w.r.t.* types and relationships in the validated table patterns, using KBs and the crowd. The

	type			relationship		
	KB	crowd	error	KB	crowd	error
WikiTables	0.60	0.39	0.01	0.56	0.42	0.02
WebTables	0.69	0.28	0.03	0.56	0.39	0.05
RelationalTables	0.83	0.17	0	0.89	0.11	0
Yago						
	KB	crowd	error	KB	crowd	error
WikiTables	0.73	0.25	0.02	0.60	0.36	0.04
WebTables	0.74	0.24	0.02	0.56	0.39	0.05
RelationalTables	0.90	0.10	0	0.91	0.09	0
DBPedia						

Table 5: Data annotation by KBs and crowd



(a) Yago (b) DBPedia

Figure 8: Top-k repair F-measure (RelationalTables)

result of data annotation is shown in Table 5. Note that KATARA annotates data in three categories (cf. Section 6.1): when KB has coverage for a value, the value is said to be validated by the KB (KB column in Table 5), when the KB has no coverage, the value is either validated by the crowd (crowd column in Table 5), or the value is erroneous (error column in Table 5). Table 5 shows the breakdown of the percentage of values in each category. Data values validated by the crowd can be used to enrich the KBs. For example, a column in one of the table in WebTables is discovered to be the type `state capitals in the United States`. Surprisingly, there are only five instances of that type in Yago[6], we can add the rest of 45 state capitals using values from the table to enrich Yago. Note that the percentage of KB validated data is much higher for RelationalTables than it is for WikiTables and WebTables. This is because data in RelationalTables is more redundant (*e.g.,* Italy appears in many tuples in Person table), when a value is validated by the crowd, it will be added to the KB, thus future occurrences of the same value will be automatically validated by the KB.

7.4 Effectiveness of Possible Repairs

In these experiments, we evaluate the effectiveness of our possible repairs generation by (1) varying the number k of possible repairs; and (2) comparing with other state of the art automatic data cleaning techniques.

Metrics. We use standard precision, recall, and F-measure for the evaluation, which are defined as follows.

precision = (#-corrected changed values)/(#-all changes)
recall = (#-corrected changed values)/(#-all errors)
F-measure= $2 \times$ (precision $\times$ recall)/(precision + recall)

For comparison with automatic data cleaning approaches, we used an equivalence-class [2] (*i.e.,* EQ) based approach provided by an open-source data cleaning tool NADEEF [12], and a ML-based approach SCARE [43]. When KATARA provides nonempty top-k possible repairs for a tuple, we count it as correct if the ground truth falls in

[6]`http://tinyurl.com/q65yrba`

	KATARA (Yago)		KATARA (DBPedia)		EQ		SCARE	
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Person	1.0	0.80	1.0	0.94	1.0	0.96	0.78	0.48
Soccer	N.A.		0.97	0.29	0.66	0.29	0.66	0.37
University	0.95	0.74	1.0	0.18	0.63	0.04	0.85	0.21

Table 6: Data repairing precision and recall (RelationalTables)

the possible repairs, otherwise we count it as an incorrect repair.

Since the average number of tuples in WikiTables and WebTables is 32 and 67, respectively, both datasets are not suitable since both EQ and SCARE require reasonable data redundancy to compute repairs. Hence, we use RelationalTables for comparison. We learn from Table 5 that tables in RelationalTables are clean, and thus are treated as ground truth. Thus, for each table in RelationalTables, we injected 10% random errors into columns that are covered by the patterns to obtain a corresponding dirty instance, that is, each tuple has a 10% chance of being modified to contain errors. Moreover, in order to set up a fair comparison, we used FDs for EQ that cover the same columns as the crowd validated table patterns (see Appendix D). SCARE requires that some columns to be correct. To enable SCARE to run, we only injected errors to the right hand side attributes of the FDs, and treated the left hand side attributes as correct attributes (*a.k.a.* reliable attributes in [43]).

Effectiveness of k. We first examined the effect of using top-k repairs in terms of F-measure. The results for both Yago and DBPedia are shown in Fig. 8. The result for soccer using Yago is missing since the discovered table pattern does not contain any relationship (cf. Fig. 10 in Appendix). Thus, KATARA cannot be used to compute possible repairs *w.r.t.* Yago. We can see the F-measure stabilizes at $k = 1$ using Yago, and stabilizes at $k = 3$ using DBPedia. The result tells us that in general the correct repairs fall into the top ones, which justifies our ranking of possible repairs. Next, we report the precision and recall of possible repairs generated by KATARA, fixing $k = 3$.

Results of RelationalTables. The precision/recall of KATARA, EQ and SCARE on RelationalTables, are reported in Table 6. The result shows that KATARA always has a high precision in cases where KBs have enough coverage of the input data. It also indicates that if KATARA can provide top-k repairs, it has a good chance that the ground truth will fall in them. The recall of KATARA depends on the coverage of the KBs of the input dataset. For example, DBPedia has a lot of information for Person, but relatively less for Soccer and University. Yago cannot be used to repair Soccer because it does not have relationships for Soccer.

Both EQ and SCARE have precision that is generally lower than KATARA, because EQ targets at computing a consistent database with the minimum number of changes, which are not necessarily the correct changes, and the result of SCARE depends on many factors, such as the quality of the training data in terms of its redundancy, and a threshold ML parameter that is hard to set precisely. The recall for both EQ and SCARE is highly dependent on data redundancy, because they both require repetition of data to either detect errors.

Results of WikiTables *and* WebTables. Table 7 shows the result of data repairing for WikiTables and WebTables. Both

	KATARA (Yago)		KATARA (DBPedia)		EQ	SCARE
	P	**R**	**P**	**R**	**P/R**	**P/R**
WikiTables	1.0	0.11	1.0	0.30	N.A.	
WebTables	1.0	0.40	1.0	0.46	N.A.	

Table 7: Data repairing precision and recall (WikiTables and WebTables)

EQ and SCARE are not applicable on WikiTables and WebTables, because there is almost no redundancy in them. Since there is no ground truth available for WikiTables and WebTables, we manually examine the top-3 possible repairs returned by KATARA. As we can see, KATARA achieves high precision on WikiTables and WebTables as well. In total, KATARA fixed 60 errors out of 204 errors, which is 29%. In fact, most of remaining errors in these tables are null values whose ground truth values are not covered by given KBs.

Summary. It can be seen that KATARA complements existing automatic repairing techniques: (1) EQ and SCARE cannot be applied to WebTables and WikiTables since there is not enough redundancy, while KATARA can, given KBs and the crowd; (2) KATARA cannot be applied when there is no coverage in the KBs, such as the case of Soccer with Yago; and (3) when both KATARA and automatic techniques can be applied, KATARA usually achieves higher precision due to its use of KBs and experts, while automatic techniques usually make heuristic changes. The recall of KATARA depends on the coverage of the KBs, while the recall of automatic techniques depends on the level of redundancy in the data.

8. RELATED WORK

The traditional problems of matching relational tables and aligning ontologies have been largely studied in the database community. A matching approach where the user is also aware of the target schema has been recently proposed [34]. Given a source and a target single relation, the user populates the empty target relation with samples of the desired output until a unique mapping is identified by the system. A recent approach that looks for isomorphisms between ontologies is PARIS [37], which exploits the rich information in the ontologies in a holistic approach to the alignment. Unfortunately, our source is a relational table and our target is a non-empty labeled graph, which make these proposals hard to apply directly. On one hand, the first approach requires to project all the entities and relationships in the target KB as binary relations, which leads to a number of target relations to test that is quadratic *w.r.t.* the number of entities, and only few instances in the target would match with the source data. On the other hand, the second approach requires to test all the possible relationships among attributes in the source relation, as it does not come with this information; in the general case there are 2^n combinations of n attributes to evaluate. However, our approach can be used to alleviate this problem, since efficiently discovering top-k relationships and types over the source table is handled by KATARA.

Another line of related work is known as *Web tables semantics understanding*, which identifies the type of a column and the relationship between two columns *w.r.t.* a given KB, for the purpose of serving Web tables to search applications [13, 28, 39]. Our pattern discovery module shares the same goal. Compared with the state of the art [28, 39], our rank join algorithm shows superiority in both effectiveness and efficiency, as demonstrated in the experiments.

Several attempts have been made to do repairing based on integrity constraints (ICs) [1, 9, 11, 17, 20]; they try to find a consistent database that satisfies given ICs in a minimum cost. It is known that the above heuristic solutions do not ensure the accuracy of data repairing [19]. To improve the accuracy of data repairing, experts have been involved as first-class citizen of data cleaning systems [19, 35, 44], high quality reference data has been leveraged [19, 24, 42], and confidence values have been placed by the users [18]. KATARA differs from them in that (1) we do not require experts to give high quality data quality rules as input; and (2) we do not involve experts to guide repairs. Instead, we explore this opportunity from crowdsourcing, which is appropriate for general pay-as-you-go service.

Numerous studies have attempted to discover data quality rules, *e.g.,* for CFDs [6] and for DCs [8]. Automatically discovered rules are error-prone, thus cannot be directly fed into data cleaning systems without verification by domain experts. However, and as noted earlier, they can exploit the output of KATARA, as rules are easier to discover from clean samples of the data [8].

Another line of work studies the problem of combining ontological reasoning with databases [5, 33]. Although their operation could also be used to enforce data validation, our work differs in that we do not assume knowledge over the constraints defined on the ontology. Moreover, constraints are usually expressed with FO logic fragments that restrict the expressive power to enable polynomial complexity in the query answering. Since we limit our queries to instance-checking over RDFS, we do not face these complexity issues.

One concern with regards to the applicability of KATARA is the accuracy and coverage of the KBs and the quality of crowdsourcing: neither the KBs nor the crowdsourcing is ensured to be completely accurate. There are several efforts that aim at improving the quality and coverage of both KBs [14–16] and crowdsourcing [4, 26]. With more accurate and big KBs, KATARA can discover the semantics of more long tail tables, and further alleviate the involvement of experts. A full discussion of the above topics lies beyond the scope of this work. Nevertheless, KBs and experts are usually more reliable than the data at hand, thus can be treated as relatively trusted resources to pivot on.

9. CONCLUSION AND FUTURE WORK

We proposed KATARA, the first *end-to-end* system that bridges knowledge bases and crowdsourcing for high quality data cleaning. KATARA first establishes the correspondence between the possibly dirty database and the available KBs by discovering and validating the table patterns. Then each tuple in the database is verified using a table pattern against a KB with possible crowd involvement when the KB lacks coverage. Experimental results have demonstrated both the effectiveness and efficiency of KATARA.

One important future work is to cold-start KATARA when there is no available KBs to cover the data, *i.e.,* bootstrapping and extending the KBs at the intensional level by soliciting structural knowledge from the crowd. It would be also interesting to assess the effects of using multiple KBs together to repair one dataset. Another line of work is to extend our current definition of tables patterns, such as a person column A_1 is related to a country column A_2 via two relationships: A_1 wasBornIn city, and city isLocatedIn A_2.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[3] G. Bouma. Normalized (pointwise) mutual information in collocation extraction. *Proceedings of GSCL*, pages 31–40, 2009.

[4] S. Buchholz and J. Latorre. Crowdsourcing preference tests, and how to detect cheating. 2011.

[5] A. Calì, G. Gottlob, and A. Pieris. Advanced processing for ontological queries. *PVLDB*, 2010.

[6] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 2008.

[7] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *ICDE*, 2011.

[8] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 2013.

[9] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[10] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. *Comput. Linguist.*, 16(1):22–29, Mar. 1990.

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

[12] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.

[13] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *PVLDB*, 2013.

[14] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: a report from the trenches. In *SIGMOD Conference*, 2013.

[15] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, 2014.

[16] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *PVLDB*, 2014.

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

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

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

[20] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-Cleaning Framework. *PVLDB*, 2013.

[21] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194, 2013.

[22] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3), 2004.

[23] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[24] M. Interlandi and N. Tang. Proof positive and negative data cleaning. In *ICDE*, 2015.

[25] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek. Interpreting and unifying outlier scores. In *SDM*, pages 13–24, 2011.

[26] R. Lange and X. Lange. Quality control in crowdsourcing: An objective measurement approach to identifying and correcting rater effects in the social evaluation of products and services. In *AAAI*, 2012.

[27] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 6(2):167–195, 2015.

[28] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1), 2010.

[29] C. D. Manning, P. Raghavan, and H. Schütze. Scoring, term weighting and the vector space model. *Introduction to Information Retrieval*, 100, 2008.

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

[31] M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. Dbpedia SPARQL benchmark - performance assessment with real queries on real data. In *ISWC*, 2011.

[32] J. Pasternack and D. Roth. Knowing what to believe (when you already know something). In *COLING*, 2010.

[33] A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. Data Semantics*, 10, 2008.

[34] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.

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

[36] S. Song, H. Cheng, J. X. Yu, and L. Chen. Repairing vertex labels under neighborhood constraints. *PVLDB*, 7(11), 2014.

[37] F. M. Suchanek, S. Abiteboul, and P. Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *PVLDB*, 2011.

[38] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.

[39] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 2011.

[40] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[41] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 2012.

[42] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.

[43] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be SCAREd: use SCalable Automatic REpairing with maximal likelihood and bounded changes. In *SIGMOD*, 2013.

[44] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.

[45] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6, 2013.

APPENDIX

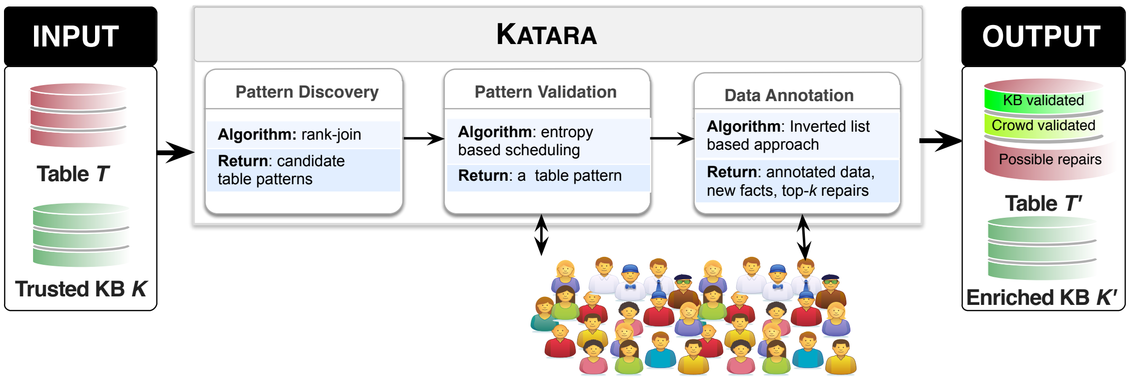
A. PROOF OF THEOREM 1

The expected uncertainty reduction is computed as the difference between the current entropy and the expected one.

$$E(\Delta H(\varphi))(v) = -H_{\mathcal{P}}(\varphi) + \textstyle\sum_a Pr(v=a) H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$$

The uncertainty of the conditional distribution of patterns given $v = a$, $H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$ can be computed as follows:

$$H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$$
$$= \textstyle\sum_{\varphi_i \in \mathcal{P}_{v=a}} \frac{Pr(\varphi_i)}{\sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)} \log_2 \frac{Pr(\varphi_i)}{\sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)}$$



INPUT
Table T
Trusted KB K
KATARA
Pattern Discovery
Algorithm: rank-join
Return: candidate table patterns
Pattern Validation
Algorithm: entropy based scheduling
Return: a table pattern
Data Annotation
Algorithm: Inverted list based approach
Return: annotated data, new facts, top-k repairs
OUTPUT
KB validated
Crowd validated
Possible repairs
Table T'
Enriched KB K'

Figure 9: Workflow of KATARA

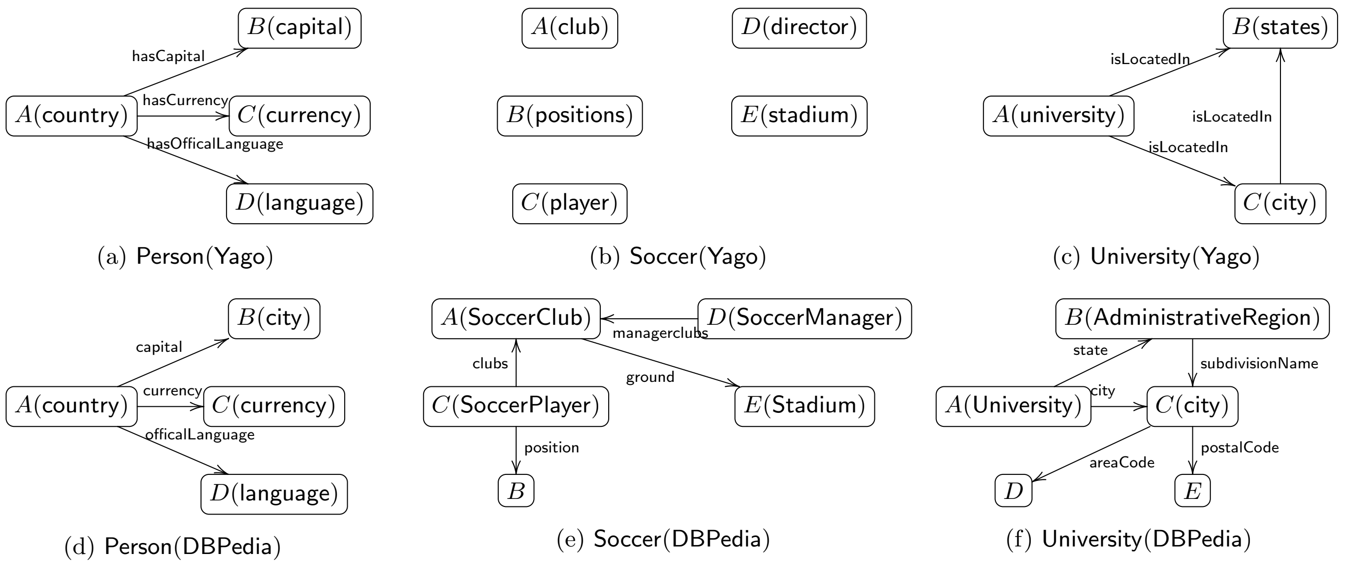


Figure 10: Validated table patterns

However, $\sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)$ is exactly $Pr(v = a)$. Thus, we can replace for $H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$.

$$
\begin{aligned}
&E(\Delta H(\varphi))(v)\\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum_a Pr(v = a) \sum_{\varphi_i \in \mathcal{P}_{v=a}} \frac{Pr(\varphi_i)}{Pr(v=a)} \log_2 \frac{Pr(\varphi_i)}{Pr(v=a)}\\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum_a \sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)(\log_2 Pr(\varphi_i) - \log_2 Pr(v = a))\\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum_a \sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i) \log_2 Pr(\varphi_i)\\
&\qquad - \textstyle\sum_a \sum_{\mathcal{P}_{v=a}} Pr(\varphi_i) \log_2 Pr(v = a)
\end{aligned}
$$

The first double summation is exactly the summation over all the current patterns, ordering them by the value of v. Thus, we have the following:

$$
\begin{aligned}
&E(\Delta H(\varphi))(v)\\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum Pr(\varphi) \log_2 Pr(\varphi)\\
&\qquad - \textstyle\sum_a \log_2 Pr(v = a) \sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)\\
&\quad = -H_{\mathcal{P}}(\varphi) + H_{\mathcal{P}}(\varphi) - \textstyle\sum_a \log_2 Pr(v = a) \times Pr(v = a)\\
&\quad = -\textstyle\sum_a Pr(v = a) \log_2 Pr(v = a)\\
&\quad = H(v)
\end{aligned}
$$

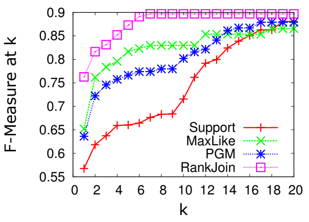
The above result proves Theorem 1.

B. TOP-K PATTERNS ANALYSIS

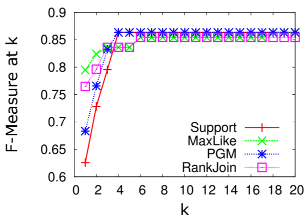
Figure 11 shows the F-measure of the top-k patterns varying k on WikiTables and RelationalTables. It tells us that RankJoin converges much quicker than other methods on Yago, while all methods converge quickly on DBPedia due to its small number of types.

C. PATTERN VALIDATION

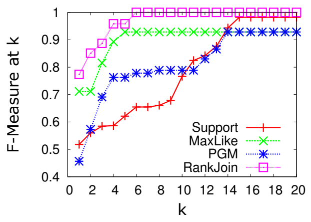
Figure 12 shows the quality of the validated pattern, varying the number of questions per variable q, on WikiTables and RelationalTables. Notice that RelationalTables only require one question per variable to achieve 1.0 precision and recall. This is because RelationalTables are less ambiguous compared with WikiTables and WebTables. Experts can correctly validate every variable with only one question.



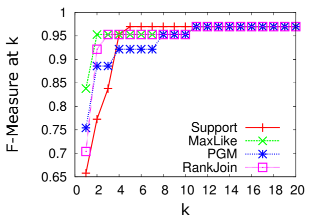
(a) Yago (WikiTables)



(b) DBPedia (WikiTables)

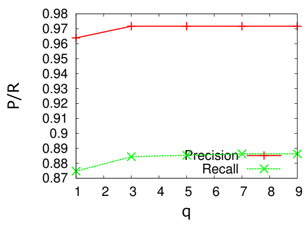


(c) Yago (RelationalTables)

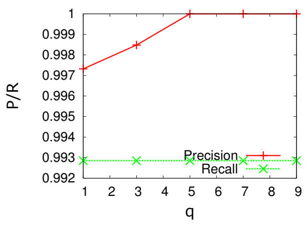


(d) DBPedia (RelationalTables)

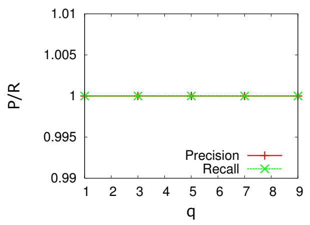
Figure 11: Top-k F-measure



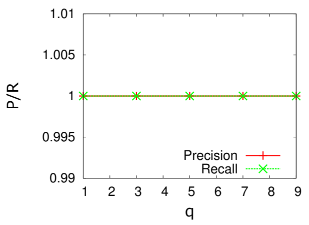
(a) Yago (WikiTables)



(b) DBPedia (WikiTables)



(c) Yago (RelationalTables)



(d) DBPedia (RelationalTables)

Figure 12: Pattern validation P/R

D. DATA REPAIRING

We use the following FDs for algorithm EQ, referring to Fig. 10.

(1) Person, we used $A \rightarrow B, C, D$.

(2) Soccer, we used $C \rightarrow A, B$, $A \rightarrow E$, and $D \rightarrow A$.

(3) University, we used $A \rightarrow B, C$ and $C \rightarrow B$.

Interactive and Deterministic Data Cleaning

A Tossed Stone Raises a Thousand Ripples

Jian He[1*] Enzo Veltri[2] Donatello Santoro[2] Guoliang Li[1]
Giansalvatore Mecca[2] Paolo Papotti[3*] Nan Tang[4]

[1]Tsinghua University, China [2]Università della Basilicata, Potenza, Italy [3]Arizona State University, USA
[4]Qatar Computing Research Institute, HBKU, Qatar

{hej13, liguoliang}@tsinghua.edu.cn, ppapotti@asu.edu, ntang@qf.org.qa
{enzo.veltri, donatello.santoro, giansalvatore.mecca}@gmail.com

ABSTRACT

We present FALCON, an *interactive, deterministic, and declarative* data cleaning system, which uses SQL update queries as the language to repair data. FALCON does not rely on the existence of a set of pre-defined data quality rules. On the contrary, it encourages users to explore the data, identify possible problems, and make updates to fix them. Bootstrapped by one user update, FALCON guesses a set of possible SQL update queries that can be used to repair the data. The main technical challenge addressed in this paper consists in finding a set of SQL update queries that is minimal in size and at the same time fixes the largest number of errors in the data. We formalize this problem as a search in a lattice-shaped space. To guarantee that the chosen updates are semantically correct, FALCON navigates the lattice by interacting with users to gradually validate the set of SQL update queries. Besides using traditional one-hop based traverse algorithms (*e.g.,* BFS or DFS), we describe novel multi-hop search algorithms such that FALCON can dive over the lattice and conduct the search efficiently. Our novel search strategy is coupled with a number of optimization techniques to further prune the search space and efficiently maintain the lattice. We have conducted extensive experiments using both real-world and synthetic datasets to show that FALCON can effectively communicate with users in data repairing.

1. INTRODUCTION

High quality data is important to all businesses, and data cleaning is an important but tedious step. In fact, removing errors in order to get high quality data takes most of data analysts' time [31], and some studies predict a shortage of people with the skills and the know-how for these tasks [33].

*Work partially done while interning/working at QCRI.

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915242

	Date	Molecule	Laboratory	Quantity
t_1	11 Nov	$C_{16}H_{16}Cl$	Austin	200
t_2	12 Nov	statin→$C_{22}H_{28}F$	Austin	200
t_3	12 Nov	$C_{24}H_{75}S_6$	N.Y.→ New York	1000→100
t_4	12 Nov	statin	Boston	200
t_5	13 Nov	statin	Austin	200
t_6	15 Nov	$C_{17}H_{20}N$	Dubai	150

Table 1: Dataset T_{drug} with drug tests.

Consequently, the number and variety of users who are getting close to the data for data quality tasks are destined to increase, and we cannot assume that only IT staff and data scientists are in charge of the data cleaning process.

The above requirement poses new and interesting research challenges. Indeed, a large body of the research has been conducted on *rule-based data repairing*, which consists of using integrity constraints to identify data errors [11,12,17,25,40], and automated algorithms to enforce these constraints over the data [7,22,23,32,43]. However, in the evolving scenario of data cleaning, these approaches show a serious limitation. Specifically, they assume that data quality rules are declared upfront by domain experts who understand the data and write logical formulas or procedural code. Despite many promising results, these systems have failed short in terms of adoption in industrial tools.

We address the problem of improving the data cleaning process by involving non-expert users as first-class citizens, and present FALCON, a novel system for *interactive* data repairing. FALCON departs from other interactive data cleaning systems [20,27,37,41,46], since it brings together a simple, user-oriented interaction paradigm with the benefits of a declarative, *deterministic*, and expressive data quality language – SQL update (SQLU) queries. In fact, the system is bootstrapped by an update to the data made by the user to rectify an error; based on that, it infers a set of SQLU queries that can be used as data quality rules to correct more errors. We illustrate by example how it works.

Example 1: Table 1 reports a sample real-world dataset T_{drug} for experiments collected from different labs. Each record represents the quantity and date of a test done in a lab over a certain molecule. Errors are highlighted. Consider the following three user updates.

Δ_1: $t_3[\mathsf{Laboratory}] \leftarrow$ "New York" (from "N.Y.")
Δ_2: $t_3[\mathsf{Quantity}] \leftarrow 100$ (from 1000)
Δ_3: $t_2[\mathsf{Molecule}] \leftarrow$ "$C_{22}H_{28}F$" (from "statin")

There exist multiple interpretations for each update. For instance, two possible semantics behind Δ_1 could be either reformatting all "N.Y." to "New York" as shown in Q_1, or changing all Laboratory values to "New York" as shown in Q'_1, regardless of their original values.

Q_1: UPDATE T_{drug} SET Laboratory = "New York" WHERE Laboratory = "N.Y.";

Q'_1: UPDATE T_{drug} SET Laboratory = "New York";

Similarly, one possible interpretation of Δ_2, as given in Q_2, is that it is specific for Molecule and Date. Hence, it is hard to generalize this update to apply it to other tuples.

Q_2: UPDATE T_{drug} SET Quantity = 100 WHERE Molecule = "$C_{24}H_{75}S_6$" AND Date = "12 Nov";

Update Δ_3 is more interesting. Consider the following three interpretations with different effects. Q_3 repairs errors in both t_2 and t_5. Q'_3 also repairs both t_2 and t_5, but additionally, it modifies t_4[Molecule] to "$C_{22}H_{28}F$", which is an erroneous update, since in Boston they test a different statin molecule. On the other hand, the tuple-specific query Q''_3 only corrects t_2 but misses the chance to repair t_5.

Q_3: UPDATE T_{drug} SET Molecule = "$C_{22}H_{28}F$" WHERE Molecule = "statin" AND Laboratory = "Austin";

Q'_3: UPDATE T_{drug} SET Molecule = "$C_{22}H_{28}F$" WHERE Molecule = "statin";

Q''_3: UPDATE T_{drug} SET Molecule = "$C_{22}H_{28}F$" WHERE Molecule = "statin" AND Laboratory = "Austin" AND Date = "12 Nov" AND Quantity = 200;

From Example 1, one may observe that there might exist a large number of SQLU queries. Indeed, this large number is not surprising, as up to thousands of precise and reliable update queries can be needed in real-world settings, such as Walmart catalog [14]. However, while an update is a perfect starting point for the process of inferring the general scripts, it comes with new challenges in terms of user interactions.

First, the search space for a new update is exponential to the number of the attributes, and domain experts cannot manually validate each of these SQLU queries. We have to assume that a *budget* (*e.g.,* #-user interactions) is given for a specific update. Second, the discovery algorithm must be fast (*e.g.,* able to react in seconds) to enable user interactions. However, each interaction may trigger the update of data, which makes the search space a *dynamic* environment. This *dynamic* behavior, together with the large search space and a budget of user capacity, prevents the use of traditional tools for interactive response, such as precomputing and caching. In order to efficiently manage all potential updates, and effectively interact with users, we propose FALCON, which works as follows.

Workflow. The workflow of FALCON is depicted in Figure 1. ❶ The user examines the data and provides a repair Δ over table T. ❷ Given Δ, FALCON generates a set of SQLU queries as rules. It then selects a query Q whose validity is yet unknown, and asks the user to verify it. ❸ Based on the user verification on Q to be either True (*i.e.,* valid) or False (*i.e.,* invalid), if Q is True, it utilizes Q to repair more data. Obviously, FALCON can prune the search space based on the validation on Q. The loop for steps ❷ and ❸ terminates when either all usable queries have been identified, or the user has no more capacity for the current Δ. Afterwards, the user may go back to step ❶ to inspect another repair.

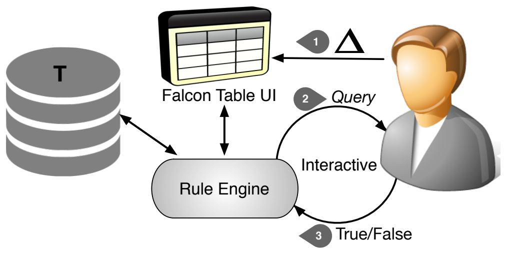


Figure 1: FALCON workflow.

Contributions. We present FALCON, a novel interactive data cleaning system, with the following contributions.

(1) To design data quality rules, we adopt the standard and deterministic language of SQL update statements (Section 2). We discuss how to organize the search space of candidate rules as a lattice, and its pruning principles, by leveraging the properties of the lattice (Section 3).

(2) We devise efficient algorithms for selecting candidate queries to effectively interact with the user (Section 4). In particular, in contrast to traditional traversal (one-hop) based approaches (Section 4.1), we present novel multiple-hop search algorithms such that FALCON can accurately discover useful queries in a small number of steps (Section 4.2).

(3) We describe optimization techniques to improve the efficiency of lattice maintenance (Section 5.1). We also propose *closed query sets* to compress the lattice so as to improve the search efficiency (Section 5.2).

(4) Implemented on top of an open-source data wrangling tool OpenRefine (http://openrefine.org), we have conducted experiments with real-world and synthetic data to show the effectiveness and efficiency of FALCON (Section 6).

Section 7 presents related work. Section 8 closes this paper, followed by our agenda for future work.

2. PROBLEM STATEMENT

We first introduce the rules used to repair data (Section 2.1). We then describe the search space of rules given one user update (Section 2.2) and formally define the problem studied in this paper (Section 2.3). Finally, we discuss its associated fundamental problems (Section 2.4).

2.1 SQL Update Queries: Mother Tongue

We adopt a simple and standard language to repair the database, the language of update statements in SQL (SQLU).

An SQLU statement updates records in a table T on attributes $A, B \ldots$, when some conditions hold. In this work, we restrict the language to the case where updates are done on one attribute A of table T with only boolean conjunctions:

UPDATE $\boxed{T}$ SET $\boxed{A = a}$ WHERE $\boxed{\text{boolean conjunctions}}$

More specifically, each boolean conjunction is of the form $B = v_B$, where B is an attribute of table T and v_B is a constant value from the domain of B, *e.g.,* Molecule = "statin". Attribute B could also be the attribute to be updated (*i.e.,* $B = A$), such as Laboratory in Q_1 of Example 1.

We shall use the terms SQLU queries and *data quality rules* (or simply *rules*) interchangeably in the following. We will also treat *updates* and *repairs* equally.

Remark. SQLU queries used in this work are quite different from the integrity constraints (ICs) that are widely adopted by other data cleaning systems, such as functional dependencies [1], conditional functional dependencies [16], conditional inclusion dependencies [7], and denial constraints [12]. ICs are used to capture errors as violations, where one violation is a set of values that is not semantically coherent when putting together. In other words, ICs do not explicitly specify how to change data values to resolve violations. In contrast, SQLU statements explicitly specify how to change data values, which are thus considered to be *deterministic*. The proposed SQLU is powerful enough to support existing deterministic cleaning languages such as fixing rules [43], constant CFDs [16], and widely used ETL rules.

Note that in this work we restrict our discussion to conjunctive SQLU queries for three reasons. (1) It is easy for users to understand, which is important for interacting with users; (2) It is efficient to reason about the relationship between different queries; and (3) It is known that queries with other formulae such as disjunctions or negations can be rewritten into an equivalent conjunctive formula [1].

2.2 Search Space for One Repair

Consider a repair $\Delta : t[A] \leftarrow a'$ that changes the value of $t[A]$ from error a to its correct value a' with $a \neq a'$. We want to generalize this action so as to repair more errors.

Naturally, there exist multiple queries to interpret this repair Δ. Implicitly, for each query, the SET clause is $A \leftarrow a'$. Hence we focus on the WHERE clause. Consider a boolean condition as $B = v_B$, where B could be any attribute in relation R. In an *open-world* assumption, the constant v_B can be assigned from an infinite set of values, which is neither reasonable nor feasible in practice. Instead, we adopt a *closed-world* assumption by only using the evidence from tuple t, the tuple that is being repaired. In other words, for a query Q *w.r.t.* the above update Δ, if an attribute B appears in the WHERE condition of Q, then the boolean conjunction is $B = t[B]$, which is to bind the constant v_B to the value $t[B]$. As a special query, we consider $\varnothing$ as *no condition* being enforced in the WHERE clause. Stating in another way, it is to update all A values in T to a'.

In summary, given a repair $t[A] \leftarrow a'$ for tuple t in table T of relation R, the set $\mathcal{Q}$ of all rules for such a repair is:

$$\text{UPDATE } \boxed{T} \text{ SET } \boxed{A = a'} \text{ WHERE } \boxed{X = t[X]}$$

where X is an arbitrary subset of R, which can range from the empty set $\varnothing$ to all attributes in R (*i.e.,* $X = R$). Hence, there are $2^{|R|}$ possibilities of X, where $|R|$ is the arity of relation R. In other words, we can infer $2^{|R|}$ queries for each update. Consider update Δ_3 in Example 1, we can infer $2^4 = 16$ queries, where three of them are shown as Q_3, Q_3' and Q_3''.

2.3 Problem Statement

Given a repair, one wants to find the queries that are semantically correct so as to repair the database.

Valid SQLU query. Given a repair, an SQLU query is *valid* if the query is semantically correct. Since we do not know which queries are valid in advance, we need to ask the user to either validate the query as semantically correct, or invalidate it otherwise. Naturally, we want to find all valid SQLU queries and use them to repair the database. A straightforward strategy is to ask the user to check every possible query. Of course, this method is rather expensive as there could be a large number of possible queries, for which we will use containment relationships among queries to improve the search of queries (Section 3).

Furthermore, the user normally has limited capacity for the number of queries he/she can verify. To this end, we want to find the *cost-effective queries* to maximize the number of repaired tuples based on the queries validated by the user, which is formally defined below.

Budget repair problem. Given a set $\mathcal{Q}$ of SQLU queries, a table T, and a budget B for the number of interactions the user can afford, the *budget repair problem* is to select B queries $\mathcal{Q}'$ from $\mathcal{Q}$, so as to maximize $|\bigcup_{Q \in \mathcal{Q}' \wedge \mathsf{valid}(Q)=\mathsf{T}} Q(T)|$.

Here, $\mathsf{valid}(Q)$ is a boolean function that is T (resp. F) if Q is a valid query (resp. not), and $Q(T)$ represents the set of repairs of applying query Q over table T.

Observe that in the above problem, given a query Q, the validity of Q (*i.e.,* $\mathsf{valid}(Q)$) is unknown, to be verified by the user. Such a problem is typically categorized under the framework of *online algorithms* [3], where one can process input piece-by-piece in a serial fashion (*i.e.,* the verification $\mathsf{valid}(Q)$ of some Q), without having the entire input (*i.e.,* the value $\mathsf{valid}(Q)$ for each Q in $\mathcal{Q}$) available from the start.

Offline problem. Its corresponding *offline variant* is the following. Given as input that whether each query Q in $\mathcal{Q}$ is valid or not is known, how to select B queries from $\mathcal{Q}$ to maximize the number of repaired tuples. The objective of designing an online algorithm is to get answers as accurate as the offline problem. It is easy to see that the offline problem of its online version (*i.e.,* the budget repair problem) is NP-hard, which can be readily proved by a reduction from the maximum-coverage problem [34].

On analogy of what is proved in [5], when the offline variant is NP-hard, there is no efficient algorithm for computing an optimal solution for its online algorithm. In other words, when the offline variant is intractable, there is no hope to find an optimal solution with the cost in a constant factor of the offline variant (*a.k.a.* a *competitive analysis* [39]).

However, not all is lost. As will be shown later, we can organize all queries in a graphical structure, such that when the user verifies a query Q as valid or invalid, we can even generate more inputs by computing the validity of queries Q' that are related to Q (Section 3). Even better, we devise efficient algorithms to search over the above graphical structure (Section 4) and empirically show the effectiveness of the presented strategies (Section 6).

2.4 Fundamental Problems

Let $\mathcal{Q}^+$ be a set of valid queries *w.r.t.* one user update.

Termination problem. The *termination problem* determines whether a rule-based process will stop, given $\mathcal{Q}^+$ and an instance T. We can readily verify that no matter in what order the queries in $\mathcal{Q}^+$ are executed, the whole process will terminate, since the execution of each query is deterministic.

Conflicting queries. Two queries Q_1 and Q_2 are *conflicting queries* if there exists a tuple t' such that the following two sequences of SQL updates will obtain different results: (1) $Q_1(Q_2(t'))$, *i.e.,* applying Q_2 first to t followed by Q_1, and (2) $Q_2(Q_1(t'))$.

Note that, the search space *w.r.t.* one repair $\Delta : t[A] \leftarrow a'$ is a set $\mathcal{Q}$ of queries (Section 2.2), where each query $Q \in \mathcal{Q}$ is a way to generalize the action of changing $t[A]$ to a specific value a', by considering different attribute combinations. In other words, no query Q will change a tuple to a value a'' that is different from a'. Hence, conflicting queries will not be generated in one lattice.

Determinism problem. The *determinism problem* asks whether all repairing processes (with different repairing orders of the SQLU queries) end up with the same repair, given $\mathcal{Q}^+$ and an instance T.

It is easy to verify that, given $\mathcal{Q}^+$ and T, regardless of the orders of the queries in $\mathcal{Q}^+$ are applied, all data repairs are $\bigcup_{Q \in \mathcal{Q}^+} Q(T)$, where T is the original instance. Hence, any set of rules is trivially deterministic.

3. A LATTICE: FALCON SEARCH SPACE

In this section, we shall present our organization of the search space, so as to enable both efficient and effective search over the candidate rules. We start by discussing the relationship between two data quality rules.

Rule containment. For two rules Q and Q', we say that Q is *contained* by Q' (or Q' *contains* Q), denoted by $Q \preceq Q'$, if for all possible database instances T over the input schema R, the result of $Q(T)$ is a subset of the result of $Q'(T)$ (*i.e.,* $Q(T) \subseteq Q'(T)$).

Intuitively, the rule containment captures the semantic relationship among rules. In other words, no matter which database T is used, Q will update a subset of T tuples that Q' will update if $Q \preceq Q'$, since Q is more specific than Q'.

Example 2: Consider queries Q_3, Q_3' and Q_3'' in Example 1. It is straightforward to see that both Q_3 and Q_3'' are contained by Q_3' (*i.e.,* $Q_3 \preceq Q_3'$ and $Q_3'' \preceq Q_3'$), and Q_3'' is contained by Q_3 (*i.e.,* $Q_3'' \preceq Q_3$).

It is readily to verify that the query containment "$\preceq$" is a partial order over the set $\mathcal{Q}$ of all possible rules, which is reflexive, antisymmetric, and transitive. More specifically:

[Reflexivity] $Q \preceq Q$, for any $Q \in \mathcal{Q}$.
[Antisymmetry] If $Q \preceq Q'$ and $Q' \preceq Q$, then $Q = Q'$.
[Transitivity] If $Q \preceq Q'$ and $Q' \preceq Q''$, then $Q \preceq Q''$.

For a query Q, we denote by $\mathsf{attr}(Q)$ the set of distinct attributes in its WHERE condition.

Note that for each user update, the SQLU queries have the same value constraint on the same attribute, and thus the rule containment verification is *equivalent* to a simpler condition: $Q \preceq Q'$ if $\mathsf{attr}(Q')$ is a subset $\mathsf{attr}(Q)$. For instance, $Q_3 \preceq Q_3'$ since $\mathsf{attr}(Q_3') = \{\mathsf{Molecule}\} \subseteq \{\mathsf{Molecule}, \mathsf{Laboratory}\} = \mathsf{attr}(Q_3)$.

Affected tuples. For each query Q and instance T, we call the tuples in $Q(T)$ *affected tuples*, *i.e.,* the tuples that Q will repair. We also call $|Q(T)|$ the *affected number* of Q, relative to T. Consider Q_3 and T_{drug} in Example 1 for instance. The affected tuples are $Q_3(T_{\mathsf{drug}}) = \{t_2, t_5\}$, and its corresponding affected number is $|Q_3(T_{\mathsf{drug}})| = 2$.

We discuss next how to organize these queries to facilitate search strategies.

A set with a partial order is a *partially ordered set*, or *poset*. Hence, $\mathcal{Q}$ is a *poset* on the partial order $\preceq$ of rule containment. Moreover, consider any two rules Q and Q'.

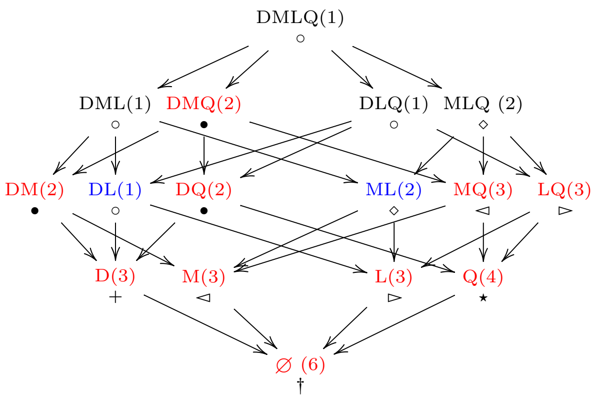


Figure 2: A sample lattice graph.

They have a *greatest lower bound*: the most specific query that is contained by both Q and Q'. This query, denoted by $Q \wedge Q'$, is the one *w.r.t.* $\mathsf{attr}(Q) \cup \mathsf{attr}(Q')$. Also, they have a *least upper bound*: the most general query that contains both Q and Q'. This query, denoted by $Q \vee Q'$, is the one *w.r.t.* $\mathsf{attr}(Q) \cap \mathsf{attr}(Q')$. Therefore, we can organize the queries in our search space as a *lattice.*

Query lattice. Given a repair Δ and a database instance T, we denote by $(\mathcal{Q}, \preceq, T)$ the corresponding lattice, or simply $(\mathcal{Q}, \preceq)$ when T is clear from the context. Each node in the lattice corresponds to a query $Q \in \mathcal{Q}$. Each directed edge from node Q to Q' indicates that $Q \preceq Q'$ (Q is contained in Q') and $|\mathsf{attr}(Q)| = |\mathsf{attr}(Q')|+1$ (with one different attribute). Moreover, the affected number associated with each query is maintained in the lattice (we will discuss how to compute the number in Section 5.1.2).

Example 3: Figure 2 depicts the lattice for dataset T_{drug} and update Δ_3 given in Example 1. Each capital letter is an abbreviation of an attribute, *e.g.,* D for Date. The node ML is for the query Q_3 on attributes Molecule and Laboratory. The edge from ML to M indicates that the query Q_3 (for ML) is contained in Q_3' (for M). The number 2 in node ML is the affected number of $|Q_3(T_{\mathsf{drug}})|$. Moreover, the greatest lower bound (resp. lowest upper bound) of ML and DL is MDL (resp. L). We postpone the discussion of the shapes in the figure, *e.g.,* "$\rhd$", "$\star$" and "$\circ$", to Section 5.2.

Valid and maximal valid nodes. Given a lattice $(\mathcal{Q}, \preceq)$, the node relative to a rule Q is *valid* if it is semantically correct, thus should be executed to repair data. In our work, if the validity of a rule is unknown, we rely on the user to verify (see more details in Section 2.3). Fortunately, if a rule Q is known to be valid, we can infer that Q' is also valid if $Q' \preceq Q$. Moreover, the node relative to a valid rule Q is *maximal valid*, if no Q'' is valid and $Q \preceq Q''$.

Example 4: Consider the lattice in Figure 2. Assume that there are two valid queries to be applied: ML (Q_3 in Example 1); and the other query DL that represents on a certain date a certain lab works on only one molecule. All red nodes are invalid queries, *i.e.,* the queries that users will semantically invalidate. The other nodes are valid nodes. Moreover, the blue nodes DL and ML are maximal valid nodes.

One nice property of using a lattice is that it provides opportunities to prune nodes to be visited during traversal.

Lattice pruning. If a node Q is valid, by inference, all nodes Q' where $Q' \preceq Q$ are valid. On the other hand, if a node Q is invalid, by inference, all nodes Q'' where $Q \preceq Q''$

are invalid. The rationality behind the above inferences is that: if one query is valid, then any query that is more specific is also valid; conversely, if it is invalid, then any query that is more general is also invalid.

We denote by $Q^{\curlyvee}$ (*i.e.,* above Q in the lattice) the queries that Q contains, and $Q_{\curlywedge}$ (*i.e.,* below Q in the lattice) the queries that contain Q. These notations naturally extend to a set of queries, $\mathcal{Q}^{\curlyvee}$ and $\mathcal{Q}_{\curlywedge}$, such that $\mathcal{Q}^{\curlyvee} = \bigcup_{Q \in \mathcal{Q}} Q^{\curlyvee}$ and $\mathcal{Q}_{\curlywedge} = \bigcup_{Q \in \mathcal{Q}} Q_{\curlywedge}$.

Example 5: Consider again the lattice in Figure 2. During interactions with the user, if DL is validated, we can then derive that $\text{DL}^{\curlyvee}$ = {DML, DLQ, DMLQ} is valid. Consider now DQ, if DQ is invalidated, we can then derive that $\text{DQ}_{\curlywedge}$ = {D, Q, ∅} is invalid.

The notation used in this paper is summarized in Table 3 in Appendix A.

4. ALGORITHMS: FALCON IN ACTION

In this section, we first describe some traversal based algorithms to solve our budget repair problem (Section 4.1). We then present advanced algorithms to efficiently navigate the search space (Section 4.2).

When discussing the algorithms, we assume that the lattice has been built given the user provided repair. The algorithms are designed for traversing the lattice and interacting with the user. Details of constructing and maintaining the lattice will be provided in Section 5.1.

4.1 One-Hop Search: Falcon Glide

In traditional traversal algorithms of a lattice $\mathcal{L}$ the search is based on some seeds, and then neighbours of the seeds (*i.e.,* one-hop) are visited by following edge connections. For example, Breadth-first search (BFS) traverses $\mathcal{L}$, by starting at the bottom and explores the neighbor nodes first, before moving to the next level neighbors. Depth-first search (DFS) differentiates in that after visiting a node, it explores as far as possible along each branch before backtracking. A recent traversal proposal, Ducc [28], bootstraps the search with a DFS-style exploration until a node of interest is found. Then it traverses the lattice alternating visits over valid and invalid nodes, in order to identify the border between them. While the algorithm was defined to find minimal unique column combination, it can be used for any lattice traversal.

To better understand how different algorithms work, we illustrate by an example below.

Example 6: Figure 3 shows how various search algorithms work, where red nodes indicate invalid nodes, blue nodes represent maximal valid nodes, and the other nodes (*i.e.,* small circles) are valid nodes. Let $B = 3$, the number of questions the user can verify. BFS search will visit the nodes in a bread-first fashion, *e.g.,* in the order $B1, B2, B3$. DFS search will visit the nodes in a depth-first fashion, *e.g.,* in the order $D1, D2, D3$. Different from BFS and DFS, Ducc [28] explores the graph in a zigzag fashion, which tries to pivot on valid nodes and explores their neighbors, *e.g.,* in the order $A1, A2, A3$. Since the above methods are edge based, the search paths are indicated on edges.

Now let us give some insight why traversal based algorithms fail for our problem. Nodes close to the top (resp. bottom) are more likely to be valid (resp. invalid). Hence, if we traverse the lattice *top-down*, we have more chances to visit a valid node Q. However, since it is close to the top, the number of inferred valid nodes $Q^{\curlyvee}$ is small. On the other hand, if we traverse the lattice *bottom-up*, we have more chances to visit an invalid node Q'. However, since it is close to the bottom, the number of inferred invalid nodes $Q'_{\curlywedge}$ is small.

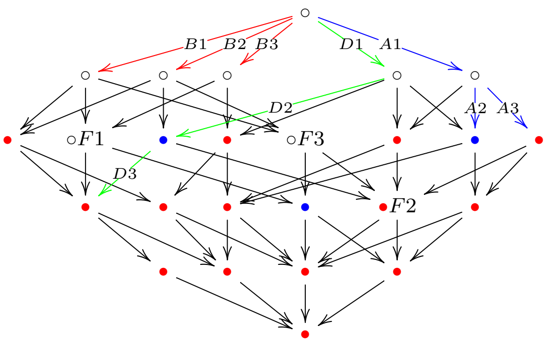


Figure 3: Lattice search algorithms. (Nodes ○/●/● represent valid nodes/maximal valid nodes/invalid nodes. Red/green/blues edges are used to explain different search strategies: BFS/DFS/Ducc.)

As shown in Example 6 and the above discussion, traversal based algorithms are locality based – they follow edge connections from visited nodes. In such a way, FALCON can only *glide* over the lattice. This is obviously not ideal when the lattice is big but the budget B is small, which is exactly the case we face. Hence, we propose new algorithms next.

4.2 Multi-Hop Search: Falcon Dive

Now that we know that traversal based algorithms are not suitable for our studied problem, we need to devise new algorithms so that FALCON can *dive* on the lattice.

4.2.1 Binary Jump

Given a budget B, our objective is to define a *divide-and-conquer* strategy that efficiently identifies nodes that are both valid and not very close to the top, so as to maximize the number of tuples to be repaired. To this purpose, we present an strategy, namely *binary jump*, inspired by classical binary search. Roughly speaking, we treat the search space as a linear space (*i.e.,* an array) by sacrificing some structural connections, and sort the nodes based on their associated affected numbers. We can then do multi-hop search to locate a candidate node to be verified with the user.

Note that conventionally, *a binary search* finds the position of a target value within a sorted array. Different from it, *binary jump* does not have a target value to be searched. In other words, binary jump is just inspired by binary search by doing half-interval style lattice traversal.

Binary jump over a path. We first discuss binary jump over a path. Consider DMLQ → DLQ → LQ → Q → ∅ in Figure 2. The ground truth of the validity of them is (T, T, F, F, F), where T means valid and F means invalid. The search algorithm does not know the ground truth, so initially we have (?, ?, ?, ?, ?). To find the truth with traversal based approaches, we need $O(N)$ questions in average, where N is the length of the path. However, using binary jump will reduce it to $O(\log N)$ questions, which is optimal, by applying inferences of finding all valid/invalid nodes.

Next we discuss the meaning of "*binary*". Straightforwardly, *binary* may refer to the offset as standard binary search. However, we need to incorporate the information of affected number. Hence, the *binary* search could refer to the median number. For instance, in Figure 2, the path DMLQ $\rightarrow$ DLQ $\rightarrow$ LQ $\rightarrow$ Q $\rightarrow \varnothing$ corresponds to the affected numbers $(1, 2, 3, 4, 6)$ and the binary jump is to find the value that is closest to $\lceil(1 + 6)/2\rceil = 4$.

For binary jump, we introduce a parameter d to bound the search depth, which is the number of iterations one can do binary jump before termination. Given a path $Q_1, Q_2, \cdots, Q_x$, we first ask the middle node $Q_{x/2}$. If the node is valid, we ask the next middle node between $Q_{x/2}$ and Q_x; otherwise, we ask the next middle node between Q_1 and $Q_{x/2}$. After d wrong searches, the process terminates. We refer to this search strategy as BinaryJump(). The rationale behind using the parameter d is that if we are following the wrong direction, we should be aware and go back to the right track, as *a fault confessed is half redressed.* We will discuss how d is set in practice in Section 6.

Note that the number of the most general query (*i.e.,* the empty set at the bottom of the lattice) will change the whole column, which makes the median number an optimistic estimation. To make it more realistic, instead, we set the binary jump using log scale to find the value that is closest to *e.g.,* $\lceil \log_2^{(1+6)} \rceil = 3$.

From a path to a lattice. In order to take the advantage of binary jump for lattice traversal, the broad intuition is to do dimension reduction from a lattice to a one-dimensional structure. That is, if we treat all nodes in the lattice uniformly, by sorting them in ascending order on their associated affected numbers, we get a sorted array similar to the one discussed above for the path.

Let $\mathcal{Q}^{?}$ denote a set of unvalidated nodes, $\mathcal{Q}^{+}$ represent a set of valid nodes, and $\mathcal{Q}^{-}$ indicate a set of invalid nodes. Next we present the algorithm.

Algorithm. Given a lattice $(\mathcal{Q}, \preceq)$ *w.r.t.* a repair Δ over table T, a budget B for the number of questions the user can answer, and a depth d to bound the search depth, the algorithm for binary jump is given below.

D1. [Initialization.] Let $\mathcal{Q}^{-} = \varnothing$, $\mathcal{Q}^{+} = \{\mathsf{top}\}$ (the top node of the lattice), and $\mathcal{Q}^{?} = \mathcal{Q}\backslash(\mathcal{Q}^{-} \cup \mathcal{Q}^{+})$. Also, let $\mathcal{Q}^{\checkmark}$ be the set of nodes verified by users, initially empty.

D2. [Sort.] Sort unvalidated nodes $\mathcal{Q}^{?}$ based on their affected numbers in ascending order.

D3. [Binary jump] Do the binary jump over $\mathcal{Q}^{?}$ and select one node Q, which is referred to as BinaryJump(). If the user still has capacity (the total number of interactions is below B), it interacts with user to verify Q, and updates $\mathcal{Q}^{\checkmark} = \mathcal{Q}^{\checkmark} \cup \{Q\}$. Otherwise, the whole process terminates. If Q is valid, it goes to step **D4**; otherwise, it goes to **D5** below, if Q is invalid.

D4. [Q is valid.] Apply Q over table T and update the affected numbers of nodes in $\mathcal{Q}$. Set $\mathcal{Q}^{+} = \mathcal{Q}^{+} \cup Q^{\curlyvee}$ (infer and enlarge valid nodes). Let $\mathcal{Q}^{?} = Q_{\curlywedge}$ and go to step **D2**.

D5. [Q is invalid.] Set $\mathcal{Q}^{-} = \mathcal{Q}^{-} \cup Q_{\curlywedge}$ (infer and enlarge invalid nodes). If the current depth is d, it goes to step **D6**. Otherwise, $\mathcal{Q}^{?} = Q^{\curlyvee}$ and goes to step **D2**.

D6. [New search space.] Let $\mathcal{Q}^{?} = \mathcal{Q}\backslash(\mathcal{Q}^{\checkmark\curlyvee} \cup \mathcal{Q}^{\checkmark}{}_{\curlywedge})$, *i.e.,* search on the nodes that are not linked to any verified node. It then goes to step **D2**.

Complexity. It is easy to see that there are up to B iterations, and the sort (**D2**) dominates the cost. Hence, the total time complexity is $O(B \cdot |\mathcal{Q}| \cdot log|\mathcal{Q}|)$. Here, budget B is typically small. Although the size of $\mathcal{Q}$ could be large for a big relation, we will discuss an optimization in Section 5.1.1 about how to ensure that the size of $\mathcal{Q}$ is easily manageable.

4.2.2 Attribute Correlation: A Good Bait

Intuitively, we want to greedily select at each step the node Q that is more likely to repair a large number of tuples. However, since we do not know what are the correct nodes until we verify them with the user, we need to estimate this information. To define the score of a node, we augment the existing information on the affected number of each query Q, *i.e.,* the number of A values Q can repair (Section 3), with the likelihood of a certain node to be related to the current attribute A.

Attribute correlations. The *attribute correlation* between two attributes A and B, denoted by $\mathsf{cor}(A, B)$, is to measure how close they are to each other.

The intuition of using attribute correlations is that, if node Q is correlated to attribute A that is being updated, then it is more likely to be semantically relevant and useful for the repair process. In general, we may get such information from data profiling tools that measure attributes correlation.

We adopt the techniques proposed in CORDS [29] to profile a database T of relation R. Specifically, CORDS computes for each attribute pair a score in $[0, 1]$. Note that (A, B) and (B, A) are different pairs. The score of an attribute pair (A, B) equals to 1 means that it is a soft FD, indicating that A approximately uniquely determines B. Otherwise, it is a score computed using χ^2 statistics by examining the attribute values in attributes A and B.

In our lattice, oftentimes, we want to estimate the correlation between the attributes in a query Q (*i.e.,* $\mathsf{attr}(Q)$) and the attribute A being updated. In other words, we need to compute the correlation between a set of attributes to a single attribute.

Using attribute correlations. We modified the algorithm presented in CORDS to compute the correlation between a set X of attributes and an attribute A, denoted by $\mathsf{cor}(X, A)$. In CORDS, an attribute pair (A, B) is a soft FD if the support value $\mathsf{sup}(A, B)$ is above a given threshold τ (see [29] for more details). Similarly, we output (X, B) as a soft FD if the support value $\mathsf{sup}(X, B) > \tau$. Otherwise, we compute the correlation score in $[0, 1]$ for (X, B) as follows.

$$\mathsf{cor}(X, B) = \frac{\chi^2}{nq} \tag{1}$$

$$\chi^2 = \sum_{v_1=1}^{m_1} \sum_{v_2=1}^{m_2} \dots \sum_{v_k=1}^{m_k} \frac{(n_{v_1,v_2,\dots,v_k} - e_{v_1,v_2,\dots,v_k})^2}{e_{v_1,v_2,\dots,v_k}} \tag{2}$$

$$e_{v_1,v_2,\dots,v_k} = n\prod_{j=1}^{k} Pr(v_j) = n\prod_{j=1}^{k} \frac{n_{v_j}^{j}}{n} = \frac{n_{i_1}^{j} n_{v_2}^{j} \dots n_{v_k}^{j}}{n^{k-1}} \tag{3}$$

$$q = \prod_{i=1}^{k} m_i - \sum_{i=1}^{k} m_i + k - 1 \tag{4}$$

Here, k is the number of attributes in X and m_i is the number of distinct values in the i-th attribute. Moreover, $(v_1, v_2, \dots, v_k)$ is a tuple where the value of the j-th attribute is v_j. Also, $n_{v_1,v_2,\dots,v_k}$ is the frequency of tuple $(v_1, v_2, \dots, v_k)$, and $e_{v_1,v_2,\dots,v_k}$ is the estimated frequency

	Austin	N.Y.	Boston	Dubai	
$C_{16}H_{16}Cl$	1	0	0	0	1
statin	2	0	1	0	3
$C_{24}H_{75}S_6$	0	1	0	0	1
$C_{17}H_{20}N$	0	0	0	1	1
	3	1	1	1	

Table 2: A 2-way contingency table.

based on the probability of v_j appearing in the j-th attribute, *i.e.,* $n^j_{v_j}/n$, where $n^j_{v_j}$ is the frequency of v_j in the j-th attribute and n is the number of tuples.

Example 7: Consider Table 1, and a given soft FD in the traditional form: $\{\mathsf{Molecule}, \mathsf{Laboratory}\} \rightarrow \mathsf{Quantity}$. Naturally, we have that the correlation value for $\mathsf{cor}(\{\mathsf{Molecule}, \mathsf{Laboratory}\}, \mathsf{Quantity}) = 1$, since they can be verified from the soft FD given above.

Consider now $X = \{\mathsf{Molecule}\}$ and $B = \mathsf{Laboratory}$. Since there is no corresponding soft FD as $\{\mathsf{Molecule}\} \rightarrow \mathsf{Laboratory}$, we compute its correlation value by normalizing χ^2 statistics.

To do so, we first compute contingency table (see Table 2). We then compute expected count of each symbol tuple. Consider tuple $\{\mathsf{statin},\mathsf{Austin}\}$. The expected count $e_{\mathsf{statin},\mathsf{Boston}} = (n^{\mathsf{Molecule}}_{\mathsf{stain}} \cdot n^{\mathsf{Laboratory}}_{\mathsf{Boston}})/n = 0.5$, and the real count $n_{\mathsf{statin},\mathsf{Austin}} = 1$. Thus the difference is $(n_{\mathsf{statin},\mathsf{Austin}} - e_{\mathsf{statin},\mathsf{Boston}})^2/e_{\mathsf{statin},\mathsf{Boston}} = 0.5$. By summing up all differences we have $\chi^2 = 12.67$, the degrees of freedom $q = 4 \cdot 4 - (4+4) + 2 - 1 = 9$, thus $\mathsf{cor}(\{\mathsf{Molecule}\}, \mathsf{Laboratory}) = 12.67/(6 * 9) = 0.235$.

We now give our greedy algorithm for multi-hop search driven by correlation and affected number.

Correlation aware binary jump (CoDive). We revise binary jump by using the correlation information, affecting **D3** in Section 4.2.1. Note that the function BinaryJump() will locate a node Q in the sorted list $\mathcal{Q}^?$. Instead of asking the user to verify Q, we revise it with the following methodology. (1) We pick more nodes around Q in the sorted list, with w on its left and the other w on its right. (2) For the above $2w + 1$ nodes, we compute their scores (affected number multiplies correlation score) and select the one with the largest score, which will then be verified by the user. We will discuss how w is set in practice in Section 6.

5. OPTIMIZATIONS

In this section, we first discuss optimizations for maintaining the lattice (Section 5.1). We then describe a technique to compress the search space, which can be applied to all algorithms (Section 5.2). We also discuss an extension when external sources are available (Appendix B).

5.1 Lattice Maintenance

There are two main challenges when maintaining the lattice: its potential large size, and the updates of affected numbers of lattice nodes during each interaction. We address these two issues below.

5.1.1 Partial Lattice Materialization

For some dataset, the number of attributes in R can be large, such that a full materialization of the lattice is prohibitively expensive with $2^{|R|}$ nodes.

Fortunately, in our framework, the update provided by the user is a strong indicator to guide which attributes should be used. The intuition is that, given an update $t[A] \leftarrow a'$, not all attributes are relevant. Consequently, constructing a lattice by incorporating irrelevant attributes will decrease both efficiency and effectiveness. Hence, we propose to pick top-k attributes that are related to the attribute A being updated, based on the attribute correlation score discussed in Section 4.2.2. We refer to such a strategy as *partial lattice materalization*, which performs much faster than a full materialization of the entire lattice, without losing accuracy. This reduces the time complexity from $O(2^{|R|} \cdot |T|)$ to $O(2^k \cdot |T|)$ where k could be much smaller than $|R|$ in practice.

Practically, attribute correlation plays an important role in devising effective search strategies. We combine functional dependencies (FDs) and highly related attribute sets (rules) to improve the search strategy. Please see the experiment in Appendix D.1 for more details on this point.

5.1.2 Initialize and Maintain Affected Numbers

Initialization. Given an update $t[A] \leftarrow a'$, we need to compute the affected number of each query Q in the lattice. The straightforward way of executing an SQLU query for each node is very costly.

We approach the problem of *initializing affected numbers* by leveraging the containment relationships between nodes. Consider two queries Q and Q', if $Q \preceq Q'$, then given any database T, we have $Q(T) \subseteq Q'(T)$. Clearly, we can compute the result of $Q(T)$ from $Q'(T)$. This is exactly the problem of answering queries using materialized views [26]. Given the simplicity of the SQLU queries adopted in this work, the query rewriting is simply to apply a selection using a constant value.

Example 8: Consider two queries Q_3, Q'_3, and the dataset T_{drug} in Example 1. If we compute Q'_3 over T_{drug} first as $Q'_3(T_{\mathsf{drug}})$, the result of $Q_3(T_{\mathsf{drug}})$ is simply to select all tuples from $Q'_3(T_{\mathsf{drug}})$ whose $\mathsf{Laboratory}$ values are Austin.

The above example suggests a simple way of computing affected numbers of lattice nodes in a bottom-up fashion. Indeed, only one SQLU query is needed for the bottom node of the lattice. Afterwards, in the bottom-up procedure, for each query Q, it applies the aforementioned query rewriting technique on $Q'(T)$ to compute $Q(T)$, where $Q \preceq Q'$ indicates that Q' is one level below Q.

Maintenance. Given the lattice $(\mathcal{Q}, \preceq)$ for table T and update Δ, when some rule Q is validated by the user, the tuples affected by Q will be repaired, *i.e.,* $Q(T)$ will result in a repaired database T' where $T' = T \oplus Q(T)$, *i.e.,* applying Q to T. For each yet unvalidated rule Q', the above changes should be reflected, *i.e.,* the number of affected tuples should be changed correspondingly, from $|Q'(T)|$ to $|Q'(T')|$.

The straightforward way is to execute $Q'(T')$ to refresh $|Q'(T')|$, or an optimized way of using the query rewriting technique discussed above. However, in such incremental scenarios, incremental algorithms have been developed for various applications (see [36] for a survey). For incremental algorithms, the updates are typically computed from *affected areas*, not the entire dataset. In our case, the affect area is exactly the affected tuples $Q(T)$. Next, we discuss how to compute, for each unvalidated rule Q', the new $|Q'(T')|$.

Case 1 [$Q' \preceq Q$]**:** $|Q'(T')| = 0$.

Case 2 [$Q \preceq Q''$]**:** $|Q''(T')| = |Q''(T)| - |Q(T)|$.

Case 3 [Q **and** Q''' **are disjoint**]**:** Neither $Q \preceq Q'''$ nor $Q''' \preceq Q$ holds. We have $|Q'''(T')| = |Q'''(T)| - |Q'''(Q(T))|$.

The above case 1 says that, if a valid rule Q is executed, then the tuples that can be affected by the queries Q' it contains have already been repaired. It is safe to set their affected numbers to 0 directly. The above case 2 tells that, for all the queries Q'' that contains Q, the set of tuples $Q(T)$ that Q'' can affect has been repaired. Hence, it is simple to reduce their affected numbers by $|Q(T)|$. In case 3, since neither $Q''' \preceq Q$ nor $Q \preceq Q'''$ holds, it first checks the number of tuples that Q''' can affect *w.r.t.* Q by executing $Q'''(Q(T))$, and then deducts its cardinality $|Q'''(Q(T))|$ from its maintained value $|Q'''(T)|$.

Time complexity. Cases 1 and 2 are clearly in constant time. For case 3, the cost is reduced from computing $Q'''(T')$ (*i.e.,* the entire table) to $Q'''(Q(T))$ (the tuples affected by Q) where $|Q(T)|$ is typically much smaller than $|T|$.

Example 9: Consider Fig. 2. Assume that during one interaction, the users validate ML (*i.e.,* query Q_3 in Example 1). The affected tuples are $Q_3(T_{\mathsf{drug}}) = \{t_2, t_5\}$ and $|Q_3(T_{\mathsf{drug}})| = 2$. One can directly set the numbers associated with DML, DLQ, and DMLQ to 0 (case 1). Moreover, it is safe to change the number with node M as $3 - 2 = 1$. Similarly, we change the number with L (resp. $\varnothing$) to 1 (resp. 4) (case 2). Consider DL and tuples $Q_3(T_{\mathsf{drug}}) = \{t_2, t_5\}$, it is easy to verify that DL can update t_2 but not t_5, hence the number with DL will be changed as $1 - 1 = 0$ (case 3).

5.2 Closed Rule Sets

A natural question, when searching a lattice, is whether there is any redundancy in the behavior of the rules, so we turn our attention now on how to identify such redundancy.

Closure operator f. Given a lattice $(\mathcal{Q}, \preceq)$ for update Δ and table T, we define a closure operator f. For any $Q \in \mathcal{Q}$, let $f(Q) = \{Q'\}$ and the following properties hold: (1) $Q \preceq Q'$; (2) $|Q(T)| = |Q'(T)|$; and (3) $\nexists Q'' \in \mathcal{Q}$ where $Q' \neq Q''$, $Q' \preceq Q''$, and $|Q'(T)| = |Q''(T)|$.

Intuitively, the closure operator f is to locate the *maximal* ancestor of a query Q that has the same effect on the number tuples they can change. Consider Fig. 2 for example, we have $f(\text{DMLQ}) = \{\text{DL}\}$, and $f(\text{DMQ}) = \{\text{DM, DQ}\}$.

Closed rule sets. Given a lattice $(\mathcal{Q}, \preceq)$, two rules Q and Q' belong to the same *closed rule set*, *iff* $f(Q) = f(Q')$. The smallest (minimal) closed rule set contains one rule Q, *i.e.,* $f(Q) = \{Q\}$ and no other rule Q' where $f(Q') = \{Q\}$.

Example 10: Consider Fig. 2. The shapes identify distinct closed rule sets. For example, the closed rule set for "○" is {DMLQ, DML, DLQ, DL}, since they are connected and have the same affected numbers. Also, the closed rule set for "•" is {DMQ, DM, DQ}, similar for other shapes.

It deserves to note that the concept of closed rule sets is in the instance level, *i.e.,* queries in the same closed rule set will change the same set of tuples for the given dataset. However, they are not the same in the semantic level, *i.e.,* some of them might be valid while the others might be invalid. In order to better understand the above discussion, consider an extreme case that each lattice node can change only one tuple, which makes all candidate queries in one closed rule set. Apparently, they contain both valid and invalid rules. In other words, the closed rule set ignores the factor that whether a rule is valid or not.

Representative rule. One natural question, given a closed rule set, is which query to be verified by the user. The intuition behind our choice is that, the more specific the query is, the easier it is for the user to verify. Hence, we define the most *representative rule* in a closed rule set to be the query Q with the largest number of predicates *w.r.t.* $|\mathsf{attr}(Q)|$.

For instance, in Example 10, the representative rule for the rule set of "○", {DMLQ, DML, DLQ, DL}, is DMLQ.

Benefits of the closed rules set. Any search algorithm over the lattice can benefit from the closed rules set. Given a node in a set, there are consequences that favor the search both if the rule is judged valid or invalid. Remember that we expose and test the representative rule. If it is true, we do not need to compute the updates for any query in the same closed rule set any more. If the answer is no, we also have a benefit in terms of pruning of the nodes, since all the nodes in the set can be safely discharged.

Example 11: Consider Figure 3 and the case that an algorithm has to test node F1. By computing the closed rule set (nodes marked with ○), the rule at the top is tested. If the rule is valid, and therefore being executed, all the nodes marked with ○ will have empty updates now, so we can avoid their computation. But if the rule is invalid, we can prune all the nodes in the set, which is a big benefit compared with the failed test of F1. In the latter case, we would still have to validate the remaining nodes marked with ○, even if we can already derive that they are not valid.

The major difference of our lattice, in contrast to traditional closed item set lattice used for data mining [42], is that our lattice is dynamically changed. More specifically, for each node Q, its associated information $|Q(T)|$ might change during each interaction, such that the closed rule sets will change correspondingly.

6. EXPERIMENTAL STUDY

We implemented Falcon in Java and used PostgreSQL 9.3 as the underlying DBMS. All experiments were conducted on a MacBook Pro with an Intel i7 CPU@2.3Ghz and 16GB of memory. Our frontend extends OpenRefine.

Datasets. We used four real-world datasets and one synthetic dataset, described as follows.

❶ Soccer is a real dataset with 7 attributes and 1625 tuples about soccer players and their clubs scraped from the Web (www.premierleague.com/en-gb.html, www.legaseriea.it/en/, www.bundesliga.com/en/).

❷ Hospital is based on a dataset from US Department of Health & Human Services (http://www.medicare.gov/hospitalcompare/). It has 12 attributes and 100k tuples.

❸ BUS is one of the UK government public datasets available at http://data.gov.uk/data and deals with bus schedules and routes. It contains 15 attributes and 250K tuples.

❹ DBLP is based on the popular collection of authors, publications and venues from http://dblp.uni-trier.de/xml/. We downloaded the whole XML dataset, and translated it into a single relational table with 15 attributes. We considered instances of 1M and 5M tuples, for quality and scalability tests, respectively.

❺ Synth is a dataset we designed starting from the original Soccer dataset in order to study the scalability over the number of tuples and a larger number of attributes. The dataset has 10 attributes and we used a generator from http://www.cs.toronto.edu/tox/toxgene/ to create instances of different sizes.

Algorithms. We implemented several algorithms for the exploration of the lattice. First, we study our own proposals for multi-hop search. Dive is the binary jump algorithm presented in Section 4.2.1. CoDive is its extention to make use of the attributes correlation information, when this is available, as described in Section 4.2.2.

These are compared with one-hop search strategies (Section 4.1). Beside BFS and DFS, we have also implemented Ducc [28], which was designed to reduce the number of tests during the discovery of all the minimal unique column combinations in a given dataset. As we will show in the results, Ducc is better than BFS and DFS for extensive searches of maximal rules in the lattice, but it was not designed to deal with small values of budget B for user interactions.

In addition to these, we also compared our results with the greedy search algorithm for the off-line version of our problem. This algorithm, OffLine, is aware of the valid nodes in the lattice. Given this information, it greedily picks the node that maximizes the error coverage at each step, with the number of steps equals to the budget B.

Baselines. We compared Falcon with four baselines.

❶ *Refine*: Our proposal generalizes the transformation language of existing tools such as OpenRefine (http://openrefine.org/) and Trifacta Wrangler (https://www.trifacta.com/trifacta-wrangler/) [27]. These tools enable users to define transformations by examples exactly as in our setting. Users modify values in a cell for attribute A and the systems suggests possible transformations over the remaining tuples for A. While we do not focus on string manipulation as some of these tools, our language supports rules (*i.e.,* transformations) with look-up over any combination of columns in the relation. In fact, given an update, these tools enable the inference of only two transformations that are comparable to our language: either the single cell is updated (the top of the lattice) or the erroneous value e is replaced with the new value v for all the occurrences in the attribute. The latter corresponds to one of the nodes in our lattice. More precisely, the standardization rule is: UPDATE T SET $A = v$ WHERE $A = e$.

Given this context, a natural baseline algorithm models these transformation tools. This algorithm, namely Refine, checks for every user update the node that generalizes it to a standardization rule, or picks the rule at the top of the lattice if the validation fails.

❷ *Rule-Learning Approaches:* Many previous approaches have concentrated on learning data-quality rules (*e.g.,* [12, 17]). Therefore, we compared our algorithm with one of these methods. More specifically:

(*i*) starting from a dirty database, we asked users to clean a sample of tuples (part of the budget was used to do this);

(*ii*) based on the sample tuples, we used a CFD-miner to learn a number of SQL-updates; since it is known that rule-mining algorithms may discover semantically invalid rules (due to *"overfitting"*) we asked users to select a subset of semantically valid rules (the second part of the budget was used for this purpose); and

(*iii*) we used the set of SQL-updates to repair the dirty instance, and measured the *benefit* score (see below).

❸ *Guided Data Repairs:* To explore the impact of *active learning*, we used GDR [46]. GDR (*"Guided Data Repairs"*) is a recently proposed algorithm that relies on active learning in order to improve the quality of repairs. Given a set of rules, it will incrementally ask users to solicit the right repairs suggested by the rules. We tested an incremental variant of algorithm ❷ above, by using GDR to suggest repairs (*i.e.,* cell updates) to users. In this case, additional budget is used to answer GDR user queries.

❹ *Active Learning in Lattice Traversal:* Finally, we compared our methods to an active learning variant of our lattice-based approach that was designed *ad-hoc* for this purpose. In the active learning algorithm, we first generated some features for each node, including attribute indicator, attribute values, original value, and updated value. We then trained a support vector machine (SVM) model with labeled nodes. Finally we used active learning to select the best node to ask users in each iteration.

Errors and Metrics. Since the considered datasets are clean, we introduce noise to verify the algorithms behaviour in the cleaning process. To start, we manually defined a set of CFDs [16] and fixing rules [43] for each scenario. We used 8 rules for Soccer, 124 rules for Hospital, 8 rules for BUS, 69 rules for DBLP, and 12 rules for Synth.

Afterwards, we used an error-generation tool to inject errors into the clean instances. To make our error-generation more systematic, we relied on an open-source error generation tool by Arocena and others [6]. The tool allows users to inject various kinds of errors within a clean database, both rule-based, random and statistics-based. Being based on an open-source tool, our error generation configuration can be easily shared and reused.

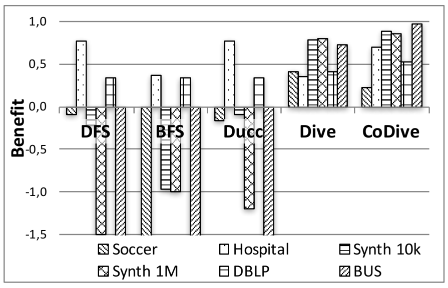
We keep running an algorithm until all the introduced errors are fixed either by a rule or by the user updates. Then, we focus our attention on the *interaction cost.* We adopt natural metrics: the number of user-provided updates U, the number of users' answers for nodes validation A, and we simply add them up to get the total interaction cost T_C. Notice that the latest metric is treating both kinds of interaction with the same weight, *i.e.,* they are considered equally difficult for the user. Despite more sophisticated combinations are possible, we found that the simple sum gives a global overview of the algorithms behaviour that is close to the real overall experience of the users.

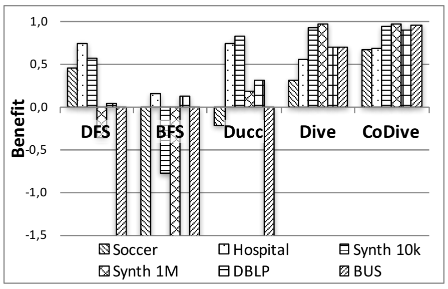
In order to have an indicator of the advantage of using interactive cleaning, we also measure the *benefit* of an algorithm in comparison to the manual update of all the errors. We first define the *cost ratio* as the number of actions divided by the number of errors. Manually updating 100 errors requires 100 user actions (updates) for a cost ratio of 1. However, by using our tool, it may be the case that 25 actions can fix 100 errors, therefore the cost ratio would be 0.25. Given an algorithm α, a dataset D, and the interaction cost T_C to obtain a set of queries $\mathcal{Q}$ covering all introduced errors, we define the *benefit* of the algorithm as $\text{BNF}_\alpha = 1 - T_C/|\mathcal{Q}(D)|$.

Finally, we measure the execution times for the algorithms in the generation of the lattice and in its maintenance.

Notice that we do not assume that users always provide correct inputs. On the contrary, the impact of user mistakes is studied in one of our experiments.

Experiments. We conducted five experiments. (*i*) Exp-1 compares benefits of the various lattice-traversal algorithms with different budget values, and show that CoDive maximizes the benefit. (*ii*) Exp-2 studies the impact of different


(a) Budget=2


(b) Budget=3

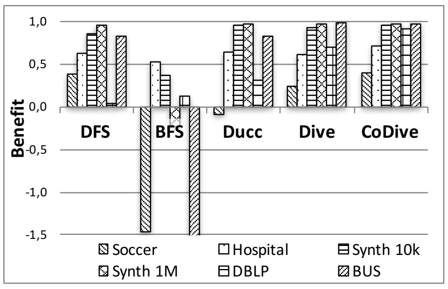

(c) Budget=5

Figure 4: Benefit for the various algorithms for the five datasets.

parameters of the models. In particular, we show that closed rule sets, an optimization technique discussed in Section 5.2, always reduces the cost. (*iii*) Exp-3 compares CoDive with closed rule sets to the four baselines. Interestingly, our algorithm outperforms all of the baselines. (*iv*) Exp-4 studies scalability. (*v*) Finally, Exp-5 investigates the robustness of FALCON *w.r.t.* user mistakes.

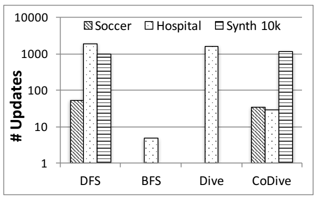
Exp-1: Lattice search algorithms. We now turn our attention to the comparison of the different search algorithms. Figure 4 reports the benefit of each algorithm for the six datasets over increasing budget B (*i.e.,* maximum number of questions after an update).

We start with the setting where the user is willing to answer only two questions (B=2) in Figure 4(a). The proposed algorithms, Dive and CoDive, consistently report a positive gain, which, for CoDive, can be interpreted as a reduction of the total user interaction cost between 22% (Soccer) and 97% (BUS). The plot also reveals that one-hop algorithms fail for the budget exploration of the lattice, with the notable exception of the Hospital dataset. This results is not surprising if we look more closely at this scenario. Hospital schema has a large number of FDs with always one or two attributes in the left hand side (LHS) of the rules. This is reflected in the CFDs that we used to introduce the errors. Rules with one or two LHS attributes are at the bottom of the lattice, and this is the most favourable setting for one-hop based algorithms, since they all start from the bottom. On the other hand, when rules start to have more attributes in the LHS, more nodes must be checked to take a decision, these algorithms fail and Dive and CoDive greatly outperform them. Similar results can be observed with $B = 3$ in Figure 4(b). More details are provided in Appendix D.2.

By increasing the budget to five questions, as reported in Figure 4(c), all algorithms can explore the lattice further at each update and the performance improve accordingly. This improvement is bigger for one-hop based algorithms as they are now able to get closer to the maximal rules in the traversal with a smaller number of updates.

Finally, since algorithm OffLine does not need to perform the search, for each update it is able to identify immediately the maximal rule. Therefore as expected OffLine is always able to completely fix the data with a number of steps that is equal to the number of rules used to introduce noise.

Exp-2: Closed rule sets and parameters. The results for the previous experiments have been conducted with the closed rule sets computed in the lattice. In fact, this optimization enables a reduction both in the number of updates and in the number of questions. To illustrate the impact of the closed rule sets, we executed the lattice search algorithms with and without this optimization, and we measure the difference in the number of required user updates and user answers to cover all errors on three scenarios (Soccer, Hospital and Synth 10k) (see Figure 5). All methods benefit from the optimization, with the exception of Ducc, which does not show any difference, and thus is not reported.


(a) Number of User Updates

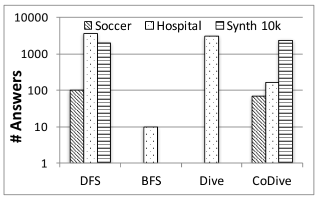
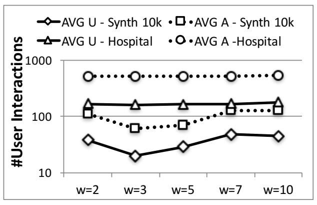

(b) Number of User Answers

Figure 5: Impact of closed rule sets for $B = 2$.


(a) Avg costs over $B = 2, 3, 5$ for algorithm CoDive wrt w

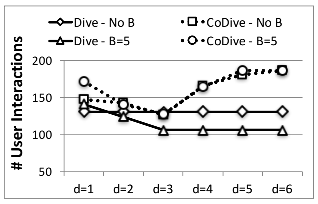

(b) Costs for Synth 1k wrt d for B=5 and without budget

Figure 6: Effects over the interaction cost. U and A are the number of user updates and user answers.

The method that gains most benefit from this optimization is DFS. The explanation is that with low budgets, such as $B = 2$, DFS always reaches the level in the lattice with two attributes. While the rule corresponding to the node may be too general and therefore invalidated by the user, it may be part of a closed rule set. Therefore, the user is offered the representative rule, which is more specific and, in some cases, true. This happens also for rules with only one attribute in the LHS for Hospital, as discussed above. In fact, even BFS, which never goes beyond nodes with only one attribute in the LHS with low B values, benefits from the closed rules set for this dataset.

As shown in Exp-1, on average CoDive has higher benefit values than Dive. However, the quality of CoDive depends on the value for parameter w (Section 4.2.2). We report in Figure 6(a) the experimental results with different w values. Each reported value is the number of user updates U (user answers A) averaged over the results for B equals to 2, 3, and 5. Both for Hospital and Synth 10k the best results are observed with $w = 3$. The parameters does not impact the results for Soccer.

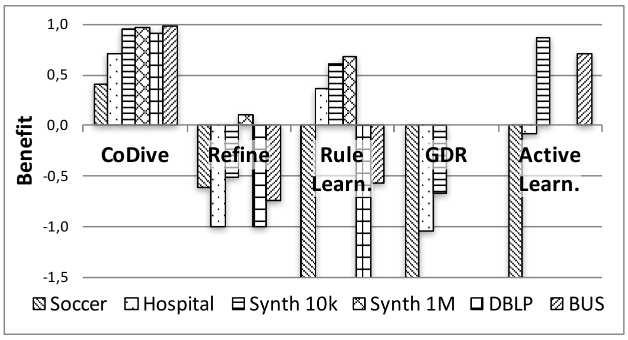


Figure 7: Benefit compared with the baselines.

We also report in Figure 6(b) the experimental results for the synthetic datasets with different values of the parameter d, as introduced for the binary jump algorithm in Section 4.2.1. Experiments over different B values and datasets also confirm that $d = 3$ leads to the best results in terms of optimization of the interaction cost.

Exp-3: Comparison to the baselines. Figure 7 reports a comparison of our CoDive algorithm to the four baselines discussed in Section 6. We fixed a timeout of two hours for all tests. Notice that not all algorithm terminated within the timeout. This accounts for the missing bars in the chart.

Our approach significantly outperforms all baselines. First, CoDive results are significantly better than those based on rule discovery. This suggests that our novel paradigm for data repairing is an improvement *w.r.t.* previous approaches in which quality rules are established upfront. Interestingly, this is confirmed also in the case in which rule discovery is coupled with an interactive algorithm, like GDR. In fact, the additional number of user interactions needed to run GDR brings to even lower benefit.

Results confirm our intuition that using user updates to lead the discovery of rules in an incremental way yields more complete and effective repairs than state-of-the-art rule-learning algorithms, which can return incomplete or redundant sets of constraints. In fact, in our experiments neither RuleLearning, nor GDR was able to repair all of errors in the data. Detailed comparison is reported in Appendix D.2.

CoDive algorithm also outperforms its active learning variant. Since ActiveLearning shares the same infrastructure as CoDive, here results are better *w.r.t.* RuleLearning and GDR. In fact, as for CoDive, whenever it terminated also ActiveLearning was able to repair all errors. ActiveLearning worked well in datasets with few rules, such as BUS and Synth 10k, while performed poorly in datasets with many rules like Hospital. Appendix C reports further details on active learning algorithm. Overall, however, benefit levels are lower. Hence, the active learning variant pays the price in terms of user-interactions of the additional training phase, which does not bring benefits *w.r.t.* CoDive.

Finally, CoDive outperforms Refine because of the less expressive language in the latter. While we discover rules using any combination of columns, Refine either generates rules for the entire column, which is unlikely to hold for data errors, or rules that update a single tuple. Single tuples updates are always correct and promptly validated, but their very small coverage leads to no benefit in using this tool.

Exp-4: Scalability. We report the performance of the lattice construction and maintenance in Figure 8. Times are reported in *ms* and the y axis is in log scale.

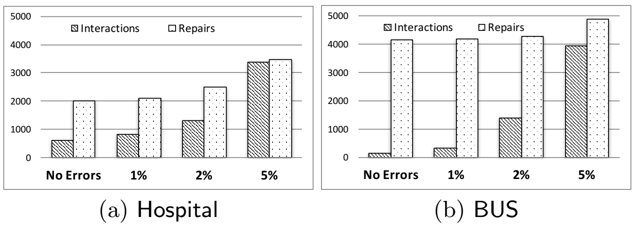


Figure 9: Impact of user mistakes.

We start by analyzing the impact of the techniques discussed in Section 5.1 for lattice maintenance. Figure 8(a) shows the total execution time for an update, defined as the time to create the lattice plus the time to update it with rules validated by the user in the interaction. We find it interesting to show that different updates can lead to very different execution times, because of the size of the queries involved in the lattice. Therefore, for the same scenario, we report both the execution times for the first user update, and corresponding interaction, and the times for the 4^{th} user update. For all combinations of updates and scenarios, the incremental maintenance is 3–5 times faster than the naive solution that rebuilds the lattice for every rule validated by the user (4 times faster on average for the first five updates).

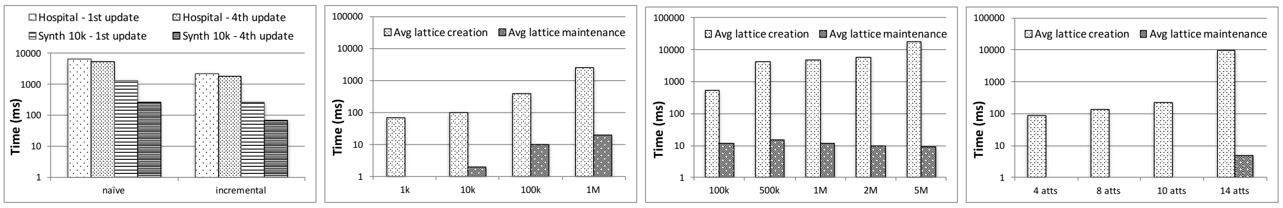
Creating the lattice requires to run queries to collect the data, and intersection over the sets of tuples to find the corresponding number of affected tuples for each node. When the dataset is large, the creation of the lattice can require a couple of seconds, as reported in Figure 8(b-c) for the average of the first ten updates. However, the creation is required only when a new user update is given, and the maintenance of the lattice in the rule validation always requires less than 20ms with our technique.

Finally, we study how the number of attributes in the dataset influences the performance. For this experiment we selected subsets of attributes of Hospital and also extended it with two more attributes by joining another table. Figure 8(d) shows the average times over the first five updates for the creation of the lattice and its maintenance with our technique. While the response time is always below 10ms, the creation of the lattice takes on average about 10 seconds, with a maximum of 30 seconds for the first user update. As discussed in Section 5.1.1, it is important to be able to identify the attributes of interest for the mining to limit the exponential explosion of the number of nodes in the lattice.

Exp-5: User Mistakes. We also tested the robustness of our approach *w.r.t.* to user-errors. That is, we do not assume that users always provide correct answers. On the contrary, assume users may sporadically make mistakes. These may be of two kinds, as follows. We notice that in both cases, our algorithm is essentially self-healing:

(i) The user performs a wrong update. This is the easier case, since we can expect that from a wrong update, only invalid rules are generated; these will be rejected by the user, and the error is fixed.

(ii) Following a valid update, the user wrongfully validates an invalid rule. This case in more delicate, since, following the wrong rule, the algorithm will indeed perform some incorrect updates. Overall, however, this will simply generate more dirtiness in the database, and the user still has a chance to correct this new dirtiness that s/he has introduced in the database in subsequent iterations.



(a) Total execution time (ms) for ith update: lattice creation + maintenance (b) Avg execution time (ms) with increasing # of tuples for Synth (first 10 updates) (c) Avg execution time (ms) with increasing # of tuples for DBLP (first 10 updates) (d) Avg exec. time (ms) with increasing # of attributes for Hospital (first 10 updates)

Figure 8: Efficiency for the lattice creation and maintenance: Dive algorithm, unbounded B.

A key property is that the rules we discover at each step are applied only once – *i.e.,* during the step they were generated in – and therefore they can be fixed by further interaction. This requires that the user is requested to reconsider some previously updated cells. As a consequence, repair ratios decrease in case of errors. In addition, we need to prevent cyclic behaviors. To do this, the system checks updates and notifies users whenever it is updating a cell that has been repaired in previous iterations. This helps users to identify previous mistakes, and prevents cycles.

Figure 9 shows the impact of user mistakes. Assume that users made mistakes with a given probability – ranging from 1% to 5% – and compare results to the case without mistake. Experiments confirm that the system is able to recover from these errors, at the price of more user interactions.

7. RELATED WORK

Data transformation. Interactive systems for data transformation [27,37,44] also reason about the updated attribute to learn transformation rules. They mainly focus on string manipulation and reformatting at the text level. In contrast, we use more expressive SQL scripts. Consequently, we discover not only rules that contain one attribute that is being updated syntactically, but also rules that combine multiple attributes to semantically determine new repairs. Our language and algorithms can lead to smaller interaction cost, as discussed in Section 6 Exp-3.

Machine learning for cleaning. Given a set of user updates, they can be used as training data to train machine learning models, which in turn can be used to predict other repairs [41, 45]. However, ML models are typically *black-boxes* that identify updates without explanations, which are hard to be trusted by users, especially for critical applications that need repairs with guaranteed correctness. Instead, SQLU queries are declarative and are preferred for human validation. Moreover, to train a machine learning model with updates, they must be semantically consistent, *i.e.,* they refer to the same type of errors. In practice, however, this assumption does not always hold since multiple updates may refer to different types of errors. This heterogeneity may hinder the usability of the trained machine learning model for prediction. Different from them, FALCON is bootstrapped by a single update, and ensures the following interactions are related to the queries with consistent semantics.

Query by examples. Several proposals have exploited the opportunity of using examples to discover queries [2,8,9,38, 49,50], schema matchings [35,47], and schema mappings [4]. They mainly focus on finding how to join multiple tables. In contrast to them, we study how to discover SQLU queries on one table, with the main challenge of understanding the update semantics that is not considered by other approaches. From an algorithmic perspective, most of these approaches exploit active learning to validate with users informative examples; we show in Section 6 Exp-3 how other signals, such as correlation, can better guide the search in our setting.

Rule-based data cleaning. Rule-based approaches for data cleaning are divided between methods to discover the rules from clean data [11,12,17,25,40], and algorithms and systems to apply the rules over dirty data to automatically fix the detected errors [7,13,15,18,19,21,23,24,30,32,43,46]. Our proposal overcomes some of the shortcomings in these methods. In terms of rules discovery, mining on dirty data leads to a lot of useless rules, therefore most of the methods report effective results assuming a clean sample. On the contrary, we naturally start from dirty data. In terms of cleaning, we restrict our language to deterministic updates, which do no need variables or placeholders that the users ultimately have to manually verify. In terms of learning from user repairs, the closest approach to our solution is the use of previous repairs to model "repair preferences" [41]. However, this approach needs a set of rules to be given as input and it only refines them, without discovering new ones.

Search on lattice. Besides traditional search strategies such as DFS and BFS, we also looked at recent lattice traversal proposals, such as the Ducc algorithm [28]. Unfortunately, these algorithms fail for our problem as they were not designed to handle the budget constraint.

Closed frequent itemset. The concept of closed frequent itemset is widely used in data mining (see [48] for a survey), where it refers to a set of itemsets that are both frequent (*i.e.,* the support value is above a given threshold) and closed (*i.e.,* there is no superset that is closed). In fact, our closed rule set is inspired from closed frequent itemset, with the major difference that our data structure (*i.e.,* the lattice) keeps changing during interactions. Traditionally, the search space for closed frequent itemset in data ming is static.

8. CONCLUSION AND FUTURE WORK

We have presented FALCON, an interactive, declarative and deterministic data cleaning system. We have demonstrated that FALCON can effectively interact with users to generalize user-solicited updates, and clean-up data with a significant benefit *w.r.t.* the number of required interactions.

A number of possible future studies using FALCON are apparent. First of all, we plan to extend it by using external sources, as remarked in Appendix B. Secondly, we will leverage the information obtained from previous interactions with the user *w.r.t.* multiple data updates. Finally, extending this approach to a crowd of noisy users would enable larger applicability and scalability, but also raises new challenges, such as the resolution of conflicts over the updates.

Acknowledgement. This work was partly supported by the 973 Program of China (2015CB358700), NSF of China (61422205, 61472198), Huawei, Shenzhou, Tencent, FDCT/116/2013/A3, MYRG105(Y1-L3)-FST13-GZ, National High-Tech R&D (863) Program of China (2012AA012600), and the Chinese Special Project of Science and Technology (2013zx01039-002-002).

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with dataplay. *PVLDB*, 5(12), 2012.

[3] S. Albers. Online algorithms: a survey. *Math. Program.*, 97(1-2), 2003.

[4] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19, 2008.

[5] C. Ambühl. Offline list update is np-hard. In *Algorithms - ESA 2000, 8th Annual European Symposium*, 2000.

[6] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with BART: error generation for evaluating data-cleaning algorithms. *PVLDB*, 9(2), 2015.

[7] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[8] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, 2014.

[9] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13), 2014.

[10] C. Chang and C. Lin. LIBSVM: A library for support vector machines. *ACMTIST*, 2(3):27, 2011.

[11] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1), 2008.

[12] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13), 2013.

[13] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.

[14] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: A report from the trenches. In *SIGMOD*, 2013.

[15] A. Ebaid, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J. Quiané-Ruiz, N. Tang, and S. Yin. NADEEF: A generalized data cleaning system. *PVLDB*, 6(12):1218–1221, 2013.

[16] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), 2008.

[17] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5), 2011.

[18] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.

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

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

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

[22] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9), 2013.

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

[24] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks! LLUNATIC goes open source. *PVLDB*, 7(13):1565–1568, 2014.

[25] L. Golab, H. J. Karloff, F. Korn, B. Saha, and D. Srivastava. Discovering conservation rules. In *ICDE*, 2012.

[26] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[27] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.

[28] A. Heise, J. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4), 2013.

[29] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[30] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, 2015.

[31] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12), 2012.

[32] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, J.-A. Quiane-Ruiz, P. Papotti, N. Tang, and S. Yin. BigDansing: a system for big data cleansing. In *SIGMOD*, 2015.

[33] J. Manyika. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.

[34] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[35] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.

[36] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

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

[38] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, pages 493–504, 2014.

[39] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), 1985.

[40] S. Song and L. Chen. Efficient discovery of similarity constraints for matching dependencies. *Data Knowl. Eng.*, 87, 2013.

[41] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[42] J. Wang, J. Han, and J. Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In *SIGKDD*, 2003.

[43] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.

[44] B. Wu and C. A. Knoblock. An iterative approach to synthesize data transformation programs. In *IJCAI*, pages 1726–1732, 2015.

[45] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD*, 2013.

[46] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.

[47] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *PVLDB*, 6(3):205–216, 2013.

[48] M. J. Zaki and W. Meira. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014.

[49] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.

[50] M. M. Zloof. Query by example. In *AFIPS*. ACM, 1975.

APPENDIX

A. SUMMARY OF NOTATION

We summarize the notations used in the paper in Table 3.

Symbol	Description
Q	a SQLU query, or a data quality rule
$Q(T)$	affected tuples of Q over table T
$Q_1 \preceq Q_2$	Q_1 is contained by Q_2
$\mathsf{attr}(Q)$	attributes in the WHERE condition of Q
Δ: $t[A] \leftarrow a'$	an update of $t[A]$ to a'
$(\mathcal{Q}, \preceq)$ *w.r.t.* Δ	a lattice of queries $\mathcal{Q}$ on partial order $\preceq$
$Q^{\curlyvee}$ $(\mathcal{Q}^{\curlyvee})$	the set of queries that Q (queries $\mathcal{Q}$) contains
$Q_{\curlywedge}$ $(\mathcal{Q}_{\curlywedge})$	the set of queries that contains Q (queries $\mathcal{Q}$)

Table 3: Notations used in the paper.

B. USING EXTERNAL SOURCES

In this section, we discuss an extension of our system when external sources are present. Often times, external sources (*e.g.,* master data) are available and contain high quality data. Next, we shall discuss how to leverage such information by FALCON.

Consider a dirty table T with schema R and master data M with schema R_m. Assume without loss of generality that $|R| = |R_m|$, and the alignment of attribute from each attribute $A \in R$ to $A \in R'$ is given. For all other attributes in either relation that are not aligned will be ignored. Moreover, given the update $\Delta : t[A] \leftarrow a'$, we assume that $A \in R$.

UPDATE $\boxed{T}$ SET $\boxed{T.A = M.A'}$

FROM $\boxed{T, M}$ WHERE $\boxed{T[X] = M[X']}$

Note that, differently from the SQLU queries defined in Section 2.1, we enforce the condition that $A \notin X$. The reason is that we assume that master data contains only correct values, but not errors. In such case, the number of potential queries is $2^{|R|-1}$, which is the number of all combinations of attributes in $R \backslash \{A\}$.

From the extension, we can repair errors from instance level to schema level by using the same lattice and the same algorithms.

C. ACTIVE LEARNING APPROACH

Active learning is a special case of semi-supervised machine learning with the goal of substantially reducing the number of labelling when training a model. All the labels for learning are obtained without reference to the learning algorithm, while in active learning the learner interactively chooses which data points to label. The hope of active learning is that interaction can substantially reduce the number of labels required. The method relies on interactively querying the user for labelling the data trying to maximize the benefit for the actual learning algorithm.

We adopt a similar idea in our setting, as described below.

In order to use active learning in our problem, *i.e.,* to predict that which node (or query) in the lattice is valid or invalid, we face two main issues to be addressed.

(1) We need to generate features for nodes in the lattice, which are used to capture their characteristics.

Indicators				AttributeValues				Original	Updated
D	M	L	Q	D	M	L	Q	M	M
1	2	1	0	11 Nov	statin	Austin	null	statin	$C_{22}H_{28}F$

Table 4: Features of node DML.

Rank	Attributes Set	Correlation
1	{Stadium, Club Country}	1
2	{Soccer Manager, Soccer Club }	1
3	{Stadium, Soccer Club}	0.822
4	{Stadium, Soccer Manager}	0.789
5	{Stadium, Soccer Club, Soccer Manager}	0.654
...	...	...
99	{Stadium, Playercountry, Soccer Club}	0.303
100	{Stadium, Position}	0.006

Table 5: Correlation of attributes in Soccer dataset when Stadium is updated.

(2) We need to use these features to select the node with the maximum benefit to be labelled, which is then interact with users to verify the selected node.

We explain in more detail about the implementation of the above two steps below.

Feature Selection. We generate a number of features for each node Q and train a Support Vector Machine (SVM) model with LIBSVM [10]. For a node Q, the features include attribute indicator, attribute value, the original value before the update, and the updated value. Attribute indicator indicates whether the attribute is included in the node (rule): if included, the indicator is 1; 0 otherwise. While if the attribute is being updated, the indicator value is 2.

Question Generation. There are two phases in question generation. First, in the initial 20 user updates, we use Ducc to explore the lattice to label the nodes, taking the nodes (and the corresponding features) from the user labelling as the training data to train a SVM model that bootstraps the active learning. Second, in each iteration, we apply the SVM model to predict the label and corresponding probability of each node, and select the node with the highest probability of being valid (reported by SVM) to ask users. After obtaining a label from user, we use **lattice pruning** technique (discussed in Section 3) to label other nodes in the lattice and add them to existing training data to re-train the SVM model.

We illustrate by an example for the active learning method.

Example 12: Consider node DML in Figure 2. Firstly, we generate features as illustrated in Table 4. Attribute indicator of D is 1 because node DML includes attribute D, indicator of M is 2 since it is the updated attribute. Attribute values are the corresponding values taken from the update Δ_3 in Example 1. The original value and updated value for the update are shown in the table.

Secondly, in each iteration, we apply SVM model to predict the probability of being valid of each node in the lattice. Suppose node ML has the highest probability to be valid that is 0.78. We then ask the user to label node ML and, since in our example the node is valid, we label all nodes above ML, *i.e.,* {ML, DML, MLQ, DMLQ} to be valid and we add them to re-train the SVM model.

	Soccer		Hospital		Synth 10k		Synth 1M		DBLP		BUS	
	U	A	U	A	U	A	U	A	U	A	U	A
DFS	11	33	**129**	**387**	177	531	5094	15282	1462	4386	3646	10938
BFS	82	246	423	1269	729	2187	14035	42105	1338	4014	4172	12516
Ducc	25	75	**129**	**387**	70	210	3083	9249	1122	3036	3646	10938
Dive	15	41	219	657	29	87	**74**	**222**	462	1386	312	936
CoDive	**8**	**19**	206	412	**24**	**72**	**74**	**222**	**140**	**420**	**48**	**144**
$\lvert Q(T)\rvert$	82		2000		1640		15000		6086		4172	

Table 6: Comparison of the lattice search algorithms with $B = 3$: U is the number of user updates, A is the number of user answers, and $|Q(T)|$ is the total number of errors.

	Soccer		Hospital		Synth 10k		Synth 1M		DBLP		BUS	
	T_C	*Rep*	T_C	*Rep*	T_C	*Rep*	T_C	*Rep*	T_C	*Rep*	T_C	*Rep*
CoDive B=5	**49**	82	**567**	2000	**70**	1640	**394**	15000	**560**	6086	**96**	4172
Refine	132	82	4000	2000	2470	1640	13326	15000	12172	6086	7258	4172
Rule Learning	194	27	315	500	474	1212	4800	15000	502	0	1191	757
GDR	225	30	1025	500	1578	943	-	-	-	-	-	-
Active Learning	217	82	2157	2000	214	1640	-	-	-	-	1220	4172
$\lvert Q(T)\rvert$	82		2000		1640		15000		6086		4172	

Table 7: Comparison of the baselines. Here T_C is the total interaction cost for the user, *Rep* is the number of repaired cells, and $|Q(T)|$ is the number of errors.

D. ADDITIONAL EXPERIMENTS

D.1 Correlation Score Results

Correlation guides the search to nodes that are likely to have a semantic connection. How to compute correlation is discussed in Section 4.2.2 to improve the binary jump strategy. This is crucial in order to discover set of attributes that form rules that are worth validating with the user. Note that if many null values are present, we only count non-null values, and the attributes with many null values will have low correlation score. To clarify the role of the correlation score, consider the following example.

Δ_1: t_1[Stadium] ← "Volkswagen Arena" (from "Weserstadion")

The user updates attribute Stadium. Table 5 shows a summary of the correlation scores, relative to attribute Stadium, for attributes in the Soccer dataset. Every set of attributes in the table can be seen as the left hand side attributes to form a FD. In fact, we know the conclusion of the rule for the given update (the Stadium attribute), but we do not know which attributes to use in the premise of the rule. From the correlation scores, we can deduce that each Stadium usually belongs to one Soccer Club and has one Soccer Manager, while each Stadium could have many Positions. Thus the correlation score of Stadium, SoccerClub, SoccerManager (row with rank 5) is much larger than that of Stadium and Position (row at rank 100). Our algorithm uses this intuition to guide the search strategy, and the experiment results also verify that the correlations are effective in avoiding rules that are unlikely to be validated by the user, such as the one deriving from row at rank 100.

D.2 More Details on Exp-1 and Exp-3

We now discuss in more details the different search algorithms and baselines for all datasets.

As reported in Table 6, all search algorithms, with the exception of BFS, lead to a clean dataset with a number of user updates U that is smaller than the number of errors in the data ($|Q(T)|$, reported at the bottom). When considering the user answers (A), their number is from 4 to 68 times smaller than the cost of manually fixing the errors (without any rule nor tool), when considering the best performing algorithm (numbers in bold). In particular, both for number of required updates and for number of required answers, CoDive is always the method with the lowest effort, with the exception of the Hospital dataset. In this case, DFS and Ducc perform better because of the simple rules that have been used to model the injection of the errors in the data. All rules for this dataset have only one or two attributes in the left hand side of the rules, such as Zip → State or Address, City → State. In these cases, the correct rules are at the bottom of the lattice, which is the level that DFS and Ducc explore first. On the contrary, if our algorithms miss the correct node at the bottom, they would start exploring the rest of the lattice and converge to the bottom again slowly, thus with a larger number of questions. Notice that, as discussed in Exp-2, the closed rule sets optimization shows a significant improvement on the Hospital scenario for the DFS algorithm.

We also remark that the Hospital dataset has a number of rather specific features. This dataset was created by joining several tables, originally in normal form, in order to obtain a large number of functional dependencies and redundancy in the data for testing rule-based data repair algorithms [20,22]. Given the highly denormalized table resulting from these joins, the dataset should not be considered representative of a standard data cleaning task.

Table 7 reports the results for CoDive compared with the baselines. Missing numbers denote cases for which the tool was stopped after the fixed timeout (two hours) for all tests. We observe that Rule Learning and GDR were not able to cover all errors because of the limited scope of the discovered rules. This is due to the limited size of the sample used in mining. A larger sample would lead to better results in terms of recall, but with a higher cost for the collection of the clean tuples. Refine is always able to detect all errors, but with a much larger number of interaction because of its less expressive language, compared to CoDive. Finally, active learning has worse performance *w.r.t.* CoDive because of required training data, and in two cases with large dataset it was not able to terminate before the timeout.

Chapter 4

Conclusions

The topic of data cleaning has gained considerable attention in the database community since the development of the first methods exploiting a declarative approach [47, 19] . After that, a huge number of contributions have been proposed to increase the coverage, with new rule languages [31, 41, 44, 49], and the effectiveness of the cleaning, with automatic algorithms [46, 65, 17] and frameworks [110, 24, 36, 64, 9] for the rule enforcement. While these approaches have proven successful to some extent, recent years have seen the rise of more interactive methods as the preferred way to tackle data preparation problems [68, 35, 69]. This direction is also confirmed by the increasing expansion of start-ups focused on the user experience in the data wrangling step, such as Tamr[1] and Trifacta[2].

The contributions gathered in this thesis aim to render the cleaning based on rules more practical and scalable and set solid foundations to explore more in the interactive cleaning direction. By developing these techniques further, the goal is to enable users to get the benefit of declarative rules without the existing obstacles. My research agenda will therefore explore this idea by moving the user interaction closer to the mining and cleaning algorithms, as demonstrated in some of the work presented in the thesis [52].

My current ongoing research is investigating a number of extensions, including two main topics that are most related to this thesis:

- **Specifications for Cleaning from Data Examples**: One line of work is to develop a unified framework for the automatic synthesis of heterogeneous cleaning specifications from data examples. Given a dataset D with errors, and a new version of the dataset D' for which at least one error has been fixed, we want to identify at least one cleaning specification that transforms D to D'. The instance pair is our *data example*. If multiple specifications exist for an example, then we want to characterize the "most general" one,

[1] `https://www.tamr.com/`
[2] `https://www.trifacta.com/`

so that its cleaning impact can be maximized over the data. In order to reach this goal, we need to tackle several challenges:

(i) Determining an appropriate formal framework in which cleaning specifications can be expressed and that allows one to numerically assess the quality of a specification. Specifications for cleaning programs are numerous and heterogeneous [3, 26]. The different kinds of specifications range from logical rules [28, 48] to procedural transformations [61], from parameters for quantitative models [84, 55], to training data for learning models [93, 4]. The framework should be general enough to express *all kinds* of specification, while at the same time being precise enough to enable rigorous formal studies. Building on solutions developed in my previous work, we will adopt for the framework the logic formalism of denial constraints [27, 28] extended with generic user defined functions [11, 64].

(ii) Understanding the computational fundamentals of the synthesis of cleaning specifications, complexity issues, and algorithms. Each example is a partial description of the semantics for a candidate specification. Given a set of examples, a decision problem states if there exists a set of specification that "fit" them. Previous efforts have shown in specific settings that the complexity varies from quadratic over the number of records for regular expressions [58] to exponential for logical formulas [7]. To handle complexity, for every specification kind we will develop a new algorithm that does not try to uniquely characterize the specification. Since any specification that holds is useful for cleaning, synthesizing the most general one(s) is the sensible approach to obtain significant cleaning effects.

(iii) Synthesizing heterogeneous specifications from data examples. Systems that infer queries and logical expressions from data examples rely on the assumption that the specification is expressed in a fixed target formalism [101, 93]. This is not the case for cleaning, as all kinds of specifications are needed to achieve effective results [3]. This new setting leads to a rethinking of the previous results, motivating algorithms that discover different kinds of specifications from a given set of examples. While every synthesis algorithm exploits the single formalism assumption for optimization, a new, "polyglot" synthesis mechanism will be built on top of the generic formal framework from the first challenge to enable a unified discovery.

- **Automatic Quality Measures for Noisy Data**: The second direction is to automate the cleaning process. Since a major goal in this context is to avoid the upfront specification and to involve the domain expert with a limited set of examples, the ability to automatically identify errors and their fixes is of utmost importance. In this context, we are looking at three problems:

  (i) Reusing existing cleaning programs. Existing mining systems discover a limited number of specifications for cleaning, are often inaccurate, and cannot identify rich semantic validations [78]. The problem is that complex specifications require background knowledge, that it is not in the data. We therefore aim at synthesizing specifications from existing programs, in which engineers and experts have already encoded their knowledge. We plan to leverage internal and external code repositories, such as GitHub. Given that specifications were not created for the dataset at hand, the challenges are in discovering the useful ones and in their reuse. We will design algorithms to automatically adapt cleaning specifications from one context to another, drawing inspiration from the literature in transfer learning [12] and recent related proposals [53]. We recently showed that this approach can be successful for declarative transformations [10], in a setting with a corpus of (generic) schema mappings [82].

(ii) Combining all available signals that lead to error identification. Examples of quality signals are statistical properties of the data, such as outlying values or correlations among attributes [27, 58], mappings to reference data [4], and rules mined from user updates [52]. Some of these methods allow the discovery of approximate specifications from noisy data, but since they are uncertain, they need to be manually validated before execution. Instead of exposing the uncertain programs for validation, we will combine their signals to identify the records that are most likely to be erroneous. We will consider the input data as a noisy version of a hidden clean dataset and treat each signal as evidence on the correctness of different records in it. To combine different signals and accumulate evidence for the detection of errors and the imputation of correct values, we will employ ensemble algorithms [90] and probability theory, which enables reasoning about inconsistencies across signals, for example with a probabilistic model whose random variables capture the uncertainty over records in the dataset [66]. Previous attempts to solve the data cleaning problem holistically considered the initial signals as certain, as the rules and mappings were manually defined by the users in a top-down fashion [36, 28, 88, 48].

(iii) Assessing the quality of a noisy dataset. Given a noisy dataset, an open challenge is ensuring that all data errors have been identified. This problem applies both for specifications, as they can be incomplete and miss rare problems, and for domain experts, as they might miss subtle issues. The ability to estimate the number of errors undetected by cleaning programs would enable the quantitative measure of the current data quality state. However, it is challenging to define data quality without knowing the ground truth (i.e., the hidden clean dataset) [83]. One direction is to exploit a statistical approach to estimate the number of remaining errors by extrapolating the number of errors from a “perfectly clean” sample [105], e.g., 10 errors in a sample of 1000 records out of 1M records leads to estimate about 10000 additional errors. This method clearly depends on the quality and size of the data sample, which is problematic to guarantee. A poor sample will give inaccurate extrapolation and therefore useless estimates. One idea is to design a statistical estimator based on the principle that every additional error is more difficult to detect than the previous ones [102], e.g., the first execution of the detection ensemble in the previous challenge should find more errors than every subsequent execution, and so on. If we estimate this diminishing return rate, we can then use it to estimate the number of remaining errors.

I envisage immediate impact of the results of my research in data cleaning systems. I have recently been awarded an ANR JCJC national grant for the duration of 3.5 years to investigate these topics in greater detail. We aim at the release of an open-source cleaning system as a product of this project. The system will be the first of its kind, as it will provide data cleaning as a service to any domain expert in the form of questions expressed over data examples. To provide the right context for these new techniques and show the impact of the project in different domains and scenarios, we plan to deploy and test the system with real case studies within ongoing collaborations, such as a leading auditing company and an European central bank.

Bibliography

[1] Data Engineering Bullettin. Special Issue on Data Transformations, 1999.

[2] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *PVLDB*, 9(4):336–347, 2015.

[3] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12), 2016.

[4] Z. Abedjan, J. Morcos, M. Gubanov, I. Ilyas, M. Stonebraker, P. Papotti, and M. Ouzzani. DataXFormer: Leveraging the web for semantic transformations. In *CIDR*, 2015.

[5] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with dataplay. *PVLDB*, 5(12), 2012.

[6] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19, 2008.

[7] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. Designing and refining schema mappings via data examples. In *SIGMOD*, 2011.

[8] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.

[9] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with BART: error generation for evaluating data-cleaning algorithms. *PVLDB*, 9(2), 2015.

[10] P. Atzeni, L. Bellomarini, P. Papotti, and R. Torlone. Meta-mappings for schema mapping reuse. *PVLDB*, 2019.

[11] P. Bailis, J. M. Hellerstein, and M. Stonebraker. *Readings in Database Systems*. 2015.

[12] S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. Pereira, and J. W. Vaughan. A theory of learning from different domains. *Mach. Learn.*, 79(1-2):151–175, 2010.

[13] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, and E. Tsamoura. Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–52, 2017.

[14] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[15] L. E. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko. The complexity and approximation of fixing numerical attributes in databases under integrity constraints. *Inf. Syst.*, 33(4-5):407–434, 2008.

[16] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 2010.

[17] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *ICDE*, 2013.

[18] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

[19] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[20] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13), 2014.

[21] A. Bonifati, U. Comignani, E. Coquery, and R. Thion. Interactive mapping specification with exemplar tuples. In *SIGMOD*, pages 667–682, 2017.

[22] D. Braga, A. Campi, and S. Ceri. XQBE (XQuery By Example): A visual interface to the standard XML query language. *ACM Trans. on Database Systems*, 30(2):398–443, 2005.

[23] M. Bronzi, V. Crescenzi, P. Merialdo, and P. Papotti. Extraction and integration of partially overlapping web sources. *PVLDB*, 6(10):805–816, 2013.

[24] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, 2014.

[25] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1), 2008.

[26] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2201–2206, 2016.

[27] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13), 2013.

[28] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[29] X. Chu, I. F. Ilyas, P. Papotti, and Y. Ye. Ruleminer: Data quality rules discovery. In *ICDE*, 2014.

[30] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. KATARA: a data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*, 2015.

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

[32] CrowdFlower. Data Science Report. Report, `http://visit.crowdflower.com/data-science-report.html`, 2016.

[33] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *PVLDB*, 2013.

[34] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: A report from the trenches. In *SIGMOD*, 2013.

[35] A. Doan, A. Ardalan, J. R. Ballard, S. Das, Y. Govind, P. Konda, H. Li, S. Mudgal, E. Paulson, P. S. G. C., and H. Zhang. Human-in-the-loop challenges for entity matching: A midterm report. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*, pages 12:1–12:6, 2017.

[36] A. Ebaid, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J. Quiané-Ruiz, N. Tang, and S. Yin. NADEEF: A generalized data cleaning system. *PVLDB*, 6(12):1218–1221, 2013.

[37] J. Euzenat and P. Shvaiko. *Ontology matching.* Springer-Verlag, Heidelberg (DE), 2nd edition, 2013.

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

[39] W. Fan and F. Geerts. *Foundations of Data Quality Management.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[40] W. Fan, F. Geerts, and X. Jia. A revival of integrity constraints for data cleaning. *PVLDB*, 1(2):1522–1523, 2008.

[41] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), 2008.

[42] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5), 2011.

[43] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.

[44] W. Fan, F. Geerts, and J. Wijsen. Determining the currency of data. In *PODS*, 2011.

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

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

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

[48] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9), 2013.

[49] L. Golab, H. J. Karloff, F. Korn, B. Saha, and D. Srivastava. Discovering conservation rules. In *ICDE*, 2012.

[50] L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.

[51] G. Gottlob and P. Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2):6:1–6:37, Feb. 2010.

[52] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning. In *SIGMOD*, pages 893–907, 2016.

[53] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. R. Narasayya, and S. Chaudhuri. Transform-data-by-example (TDE): an extensible search engine for data transformations. *PVLDB*, 11(10):1165–1177, 2018.

[54] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.

[55] J. Hellerstein. Quantitative data cleaning for large databases. UNECE White Paper, 2008.

[56] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[57] IBM. Data-driven healthcare organizations use big data analytics for big gains. White paper, `http://www.ibmbigdatahub.com/whitepaper/data-driven-healthcare-organizations-use-big-data-analytics-big-gains`.

[58] A. Ilyas, J. M. F. da Trindade, R. C. Fernandez, and S. Madden. Extracting syntactic patterns from databases. *CoRR*, abs/1710.11528, 2017.

[59] B. Insider. How the London whale debacle is partly the result of an error using excel. `http://www.businessinsider.fr/us/excel-partly-to-blame-for-trading-loss-2013-2/`, 2013.

[60] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, 2015.

[61] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[62] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12), 2012.

[63] J. Kang and J. F. Naughton. On schema matching with opaque column names and data values. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 205–216, San Diego, California, 2003.

[64] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, J.-A. Quiane-Ruiz, P. Papotti, N. Tang, and S. Yin. BigDansing: a system for big data cleansing. In *SIGMOD*, 2015.

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

[66] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning.* The MIT Press, 2009.

[67] N. Koudas, A. Saha, D. Srivastava, and S. Venkatasubramanian. Metric functional dependencies. In *ICDE*, 2009.

[68] S. Krishnan, D. Haas, M. J. Franklin, and E. Wu. Towards reliable interactive data cleaning: a user survey and recommendations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, page 9, 2016.

[69] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *PVLDB*, 9(12):948–959, 2016.

[70] M. Lichman. UCI machine learning repository, 2013.

[71] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1), 2010.

[72] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - a review. *IEEE TKDE*, 24(2):251–264, 2012.

[73] S. Lohr. For big-data scientists, janitor work is key hurdle to insights, Aug. 2014. `http://nyti.ms/1t8IzfE`.

[74] A. Lopatenko and L. Bravo. Efficient approximation algorithms for repairing inconsistent databases. In *ICDE*, 2007.

[75] J. Madhavan, P. A. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *Proc. of IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 57–68, Tokyo, Japan, 2005.

[76] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proc. of Int'l Conf. on Very Large Databases (VLDB)*, Rome, Italy, 2001.

[77] D. Mail. Tourists drive into lake champlain after waze directed them down a boat ramp. `http://www.dailymail.co.uk/news/article-5308303/Waze-directed-tourists-drive-lake.html`, 2018.

[78] V. Meduri and P. Papotti. Towards user-aware rule discovery. In *International Workshop on Information Search, Integration, and Personlization (ISIP)*, 2017.

[79] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A versatile graph matching algorithm. In *Proc. of IEEE Int'l Conf. on Data Engineering (ICDE)*, 2002.

[80] S. Ortona, V. Meduri, and P. Papotti. Robust discovery of positive and negative rules in knowledge-bases. In *ICDE*, pages 1168–1179, 2018.

[81] S. Ortona, V. V. Meduri, and P. Papotti. Rudik: Rule discovery in knowledge bases. *PVLDB*, 11(12):1946–1949, 2018.

[82] P. Papotti and R. Torlone. Schema exchange: Generic mappings for transforming data and metadata. *Data Knowl. Eng.*, 68(7):665–682, 2009.

[83] L. L. Pipino, Y. W. Lee, and R. Y. Wang. Data quality assessment. *Commun. ACM*, 45(4):211–218, Apr. 2002.

[84] N. Prokoshyna, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava. Combining quantitative and logical data cleaning. *PVLDB*, 9(4):300–311, 2015.

[85] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.

[86] Quartz. When artificial intelligence botches your medical diagnosis, whos to blame? `https://qz.com/989137/when-a-robot-ai-doctor-misdiagnoses-you-whos-to-blame/`, 2017.

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

[88] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, Aug. 2017.

[89] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[90] G. Seni and J. F. Elder. Ensemble methods in data mining: improving accuracy through combining predictions. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 2(1):1–126, 2010.

[91] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, pages 493–504, 2014.

[92] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 2012.

[93] R. Singh, V. V. Meduri, A. K. Elmagarmid, S. Madden, P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[94] S. Song and L. Chen. Efficient discovery of similarity constraints for matching dependencies. *Data Knowl. Eng.*, 87, 2013.

[95] S. Song, H. Cheng, J. X. Yu, and L. Chen. Repairing vertex labels under neighborhood constraints. *PVLDB*, 7(11):987–998, 2014.

[96] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[97] F. M. Suchanek, S. Abiteboul, and P. Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *PVLDB*, 2011.

[98] P. Suganthan, C. Sun, K. Gaytri, H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, and A. Doan. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, 2015.

[99] N. Swartz. Gartner warns firms of ‘dirty data’. *Information Management Journal*, 41(3), 2007.

[100] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of order dependencies. *Proc. VLDB Endow.*, 5(11):1220–1231, July 2012.

[101] B. ten Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *ACM Trans. Database Syst.*, 38(4):28:1–28:31, 2013.

[102] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.

[103] V. Verroios and H. Garcia-Molina. Entity resolution with crowd errors. In *IEEE ICDE*, pages 219–230, 2015.

[104] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 2012.

[105] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD*, 2014.

[106] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.

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

[108] B. Wu and C. A. Knoblock. An iterative approach to synthesize data transformation programs. In *IJCAI*, pages 1726–1732, 2015.

[109] C. M. Wyss, C. Giannella, and E. L. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK*, pages 101–110, 2001.

[110] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.

[111] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: Entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, pages 97–108, 2012.

[112] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *PVLDB*, 6(3):205–216, 2013.

[113] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6, 2013.

[114] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.

[115] M. M. Zloof. Query by example. In *AFIPS*. ACM, 1975.