Foundations and Trends® in Databases
Vol. 5, No. 4 (2012) 281–393
© 2015 I. F. Ilyas and X. Chu
DOI: 10.1561/1900000045

Trends in Cleaning Relational Data: Consistency and Deduplication

Ihab F. Ilyas
University of Waterloo
ilyas@uwaterloo.ca

Xu Chu
University of Waterloo
x4chu@uwaterloo.ca

Contents

Abstract

Data quality is one of the most important problems in data management, since dirty data often leads to inaccurate data analytics results and wrong business decisions. Poor data across businesses and the government cost the U.S. economy \$3.1 trillion a year, according to a report by InsightSquared in 2012.

To detect data errors, data quality rules or integrity constraints (ICs) have been proposed as a declarative way to describe legal or correct data instances. Any subset of data that does not conform to the defined rules is considered erroneous, which is also referred to as a violation.

Various kinds of data repairing techniques with different objectives have been introduced, where algorithms are used to detect subsets of the data that violate the declared integrity constraints, and even to suggest updates to the database such that the new database instance conforms with these constraints. While some of these algorithms aim to minimally change the database, others involve human experts or knowledge bases to verify the repairs suggested by the automatic repeating algorithms.

In this paper, we discuss the main facets and directions in designing error detection and repairing techniques. We propose a taxonomy of current anomaly detection techniques, including error types, the automation of the detection process, and error propagation. We also propose a taxonomy of current data repairing techniques, including the repair target, the automation of the repair process, and the update model. We conclude by highlighting current trends in “big data” cleaning.

I. F. Ilyas and X. Chu. *Trends in Cleaning Relational Data: Consistency and Deduplication*. Foundations and Trends® in Databases, vol. 5, no. 4, pp. 281–393, 2012.
DOI: 10.1561/1900000045.

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 [36], since establishing trust in data becomes a challenge as the variety and the number of sources grow. For example, in health care domains, inaccurate or incorrect data may threaten patient safety [75].

Gartner predicted that more than 25% of critical data in the world's top companies is flawed [106]. Poor data across businesses and the government costs the U.S. economy \$3.1 trillion a year, according to a report by InsightSquared [29]. With the increasing popularity of data science, it became evident that data curation, preparation, cleaning, and other "janitorial" data tasks, are key enablers in unleashing value of data, as indicated in a 2014 article in the New York Times[1].

Even when the data is ingested in JSON, XML, or text format, many of data quality assessment and cleaning activities happen after transforming the data into relational tables. There are many notions related to relational data quality: data consistency, data accuracy, data completeness, and data currency. Data consistency refers to the valid-

[1]`http://nyti.ms/1t8IzfE`

ity and integrity of data; data accuracy refers to how accurate the data values in a database with respect to the true values; data completeness indicates whether all the data needed to meet the information needs is available; and data currency, also known as, data timeliness, gives the degree to which the data is current with respect to the world or the process it models. There are various surveys and books on relational data quality. Rahm and Do [93] give a classification of different types of errors that can happen in an Extract-Transform-Load (ETL) process, and survey the tools available for cleaning data in an ETL process; some focus on the effect of incompleteness data on query answering [61], and the use of a Chase procedure for dealing with incomplete data [62]; Hellerstein [67] focuses on cleaning quantitative data, such as integers and floating points, using mainly statistical outlier detection techniques. Bertossi [8] provides complexity results for repairing inconsistent data, and performing consistent query answering on inconsistent data; Fan and Geerts [44] discuss the use of data quality rules in data consistency, data currency, and data completeness, how different aspects of data quality issues might interact; and Dasu and Johnson [33] summarize how techniques in exploratory data mining can be integrated with data quality management.

In this paper, we focus on the data consistency aspect of relational data quality. To ensure data consistency, data quality rules are often used. We use integrity constraints (ICs) to express data quality rules. Any part of the data that does not conform to a given set of ICs is considered erroneous, also known as a violation of ICs. Data deduplication can be seen as enforcing a key constraint defined on all the attributes of a relational schema, since two duplicate tuples can be seen as a violation of the key constraint. Data cleaning, in this context, is the exercise of detecting errors, and possibly modifying the database, such that the data conforms to a set of data quality rules expressed in a variety of languages. This paper covers techniques to detect data inconsistencies, as well as techniques to repair data inconsistencies.

The following example illustrates a real world tax record database that has various data quality problems due to the violations of different data quality rules, and the existence of duplicate records.

Example 1.1. Consider the US tax records in Table 1.1. Each record describes an individual's 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).

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 Los Angeles lives in California; (4) if two persons live in the same state, the one with lower salary has a lower tax rate; (5) tax exemption is less than the salary.

A violation with respect to an IC is defined as the minimal subset of database cells, such that at least one of the cells has to be modified to satisfy the IC, where a cell is an attribute value of a tuple, *e.g.*, Cell t_1[FN] corresponds to Attribute FN of Tuple t_1 . For instance, the set of four cells $\{t_1[\text{ZIP}], t_8[\text{ZIP}], t_1[\text{ST}], t_8[\text{ST}]\}$ is a violation with respect to the second constraint. Furthermore, Record t_4 and t_9 refer to the same person, even though t_4[FN] and t_9[FN] are different, and t_9[AC] is empty. Given a relational database instance I of schema R and a set of integrity constraints Σ, we need to find another database instance I' with no violations with respect to Σ.

1.1 Notations

Let R denote a relational schema, and I be an instance of that schema. Attributes of R are denoted as $attr(R) = \{A_1, \ldots, A_m\}$. For each Attribute A in R, let $Dom(A)$ denote the domain of A. I consists of a set of tuples, each of which belongs to the domain $Dom(A_1) \times \ldots \times Dom(A_m)$. We assume that there is a unique tuple identifier associated with each tuple $t \in I$. Let $TIDs(I)$ denote the set of all tuple identifiers. We identify a cell of Attribute A of a tuple t in I by $I(t[A])$, simply referred to as $t[A]$ when the context is clear. Let $CIDs(I)$ denote the set of all cell identifiers in I.

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	WA	25813	S	Y	70000	3	2000	0	2000
t_2	Chunho	Black	M	206	154-4816	Seattle	WA	98103	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
t_9	Ann	Puerta	F		378-7304	West Crossett	AR	72045	M	N	86000	7.22	0	40	0

Table 1.1: Tax data records.

1.2 Outline

The remainder of the paper is organized as follows. Section 2 discusses different ways to detect anomalies in the data, such as data duplication, integrity constraints languages, along with algorithms for their automatic discovery, and provenance-based error propagation, based on what, how, and where to detect. Section 3 introduces the taxonomy we adopt to classify data repairing techniques, based on what, how, and where to repair, and presents the details of multiple techniques in each dimension. Section 4 discusses the techniques proposed for dealing with big data cleaning. Section 5 concludes and summarizes future research directions.

2
Taxonomy of Anomaly Detection Techniques

Given a dirty database instance, the first step toward cleaning the database is to detect and surface anomalies or errors. Figure 2.1 depicts the classification we adopt to categorize the current anomaly detection techniques. In the following, we discuss our classification dimensions. For each dimension, we give one or more examples to discuss in detail. The three adopted dimensions capture the three main questions involved in detecting errors in a database.

- *Error Type (What to Detect?)* Anomaly detection techniques can be classified according to which type of errors can be captured. While we mentioned earlier in Section 1 that duplicate records can be considered a violation of an integrity constraint, we recognize the large body of work that focused on this problem and we discuss it separately from other types of integrity constraints. We discuss example violations of various integrity constraints (ICs), including functional dependencies (FDs), conditional functional dependencies (CFDs), and denial constrains (DCs), along with methods for their automatic discovery, and the main steps involved in detecting duplicate records.

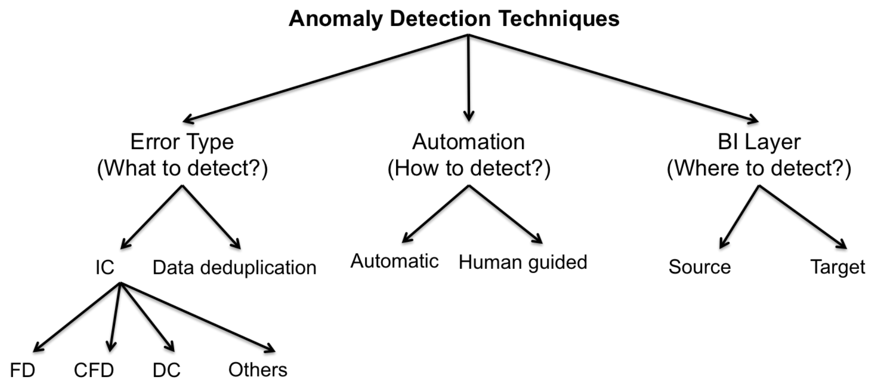


Figure 2.1: Classification of anomaly detection techniques.

- *Automation (How to Detect?)* We classify proposed approaches according to whether and how humans are involved in the anomaly detection process. Most techniques are fully automatic, for example, detecting violations of functional dependencies, while other techniques involve humans, for example, to identify duplicate records.

- *Business Intelligence Layer (Where to Detect?)* Errors can happen in all stages of a business intelligence (BI) stack, for example, errors in the source database are often propagated through the data processing pipeline. While most anomaly detection techniques detect errors in the original database, some errors can only be discovered much later in the data processing pipeline, where more semantics and business logics becomes available, for example, constraints on total budget can only be enforced after aggregating cost and expenses.

Table 2.1 shows a sample of anomaly detection techniques, which cover all categories of the proposed taxonomy.

	Error Type What		**Automation How**		**BI Layer Where**	
	IC	Data deduplication	Automatic	Human involved	Source	Target
FDs value modification [17]	✓		✓		✓	
Holistic data cleaning [26]	✓		✓		✓	
CrowdER [114]		✓		✓	✓	
Corleone [59]		✓		✓	✓	
Causality Analysis [87]	✓			✓		✓
Scorpion [123]	✓			✓		✓
DBRx [19]	✓		✓			✓

Table 2.1: A sample of anomaly detection techniques.

2.1 What to Detect

In this section, we discuss the various types of errors considered in this paper, namely, violations of integrity constraints, and presence of duplicate records. We start by discussing example formal types of integrity constraints (e.g., functional dependencies and denial constraints in Section 2.1.1, and we briefly survey de-duplication proposals in Section 2.1.2.

2.1.1 Integrity Constraints

Integrity constraints (ICs), which are usually declared as part of the database schema, have been increasingly used as data quality rules, either through checking the validity of the data upon addition or update, or by cleaning the dirty data at various points during the processing pipeline [26, 77]. Quality constraints are used to identify data violations and automatically repair them according to some cost function, such as minimizing the number of changes to the original data. Traditional types of ICs, such as key constraints, check constraints, and functional dependencies (FDs), have been proposed for data quality management [17]. Conditional functional dependencies (CFDs), an extension of FDs, have been found to be more general and expressive than FDs as they can capture many real life data quality problems that FDs fail to capture [18]. Not surprisingly, the more expressive an IC language is, the more difficult it is to exploit it in automated data cleaning algorithms, or even in consistency checking. Hence, a balance should be achieved between the expressive power of ICs in order to deal with a broader space of business rules from one side, and the development of effective cleaning and discovery algorithms from the other side [25].

Since data owners are often not data quality experts, automatic IC discovery is an extremely useful functionality in bootstrapping the cleaning exercise. In this section, we survey the most commonly used ICs in the literature for detecting data inconsistencies, as well as the techniques proposed for their automatic discovery.

Let Σ denote a set of integrity constraints (ICs). We use $I \models \Sigma$ to indicate that database Instance I conforms with the set of integrity constraints Σ. We say that I' is a repair of I with respect to Σ if $I' \models \Sigma$ and $CIDs(I) = CIDs(I')$ (i.e., no deletions or insertions are allowed to clean a database instance).

2.1.1.1 Functional Dependencies

Consider a relational schema R with attributes $attr(R)$.

Definition 2.1. A functional dependency (FD) φ is defined as $X \rightarrow Y$, where $X \subseteq attr(R)$ and $Y \subseteq attr(R)$. An instance I of R satisfies FD φ, denoted as $I \models \varphi$ if for any two tuples t_α, t_β in I, such that $t_\alpha[X] = t_\beta[X]$, then $t_\alpha[Y] = t_\beta[Y]$.

In other words, if there exist any two tuples, t_α, t_β, in any instance I, that have the same value for attributes X, but different values for Y, then there must be some errors present in t_α or t_β. We call X the left hand side (LHS), and Y the right hand side (RHS). In Example 1.1, the second data quality rule is an FD $ZIP \rightarrow ST$.

Example 2.1. Consider two tuples t_1 and t_8 in Table 1.1; they have the same value "25813" for ZIP, but t_1 has "WA" for ST and t_8 has "WV" for ST. Clearly, at least one of the four cells

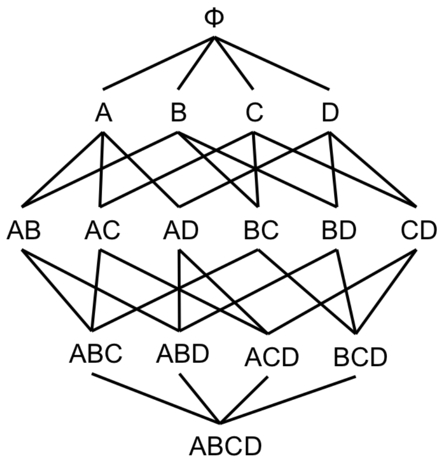
$\{t_1[ZIP], t_8[ZIP], t_1[ST], t_8[ST]\}$ has to be incorrect. In order to limit the space of possible repairs, certain data cleaning algorithms only allow changes on the RHS. In this case, changing $t_1[ST]$ to "WV" or changing $t_8[ST]$ to "WA" would fix the violation of the FD. If changes on the LHS are allowed, changing $t_1[ZIP]$ or $t_8[ZIP]$ to any value other than "25813" would also resolve the violation.

FD Discovery

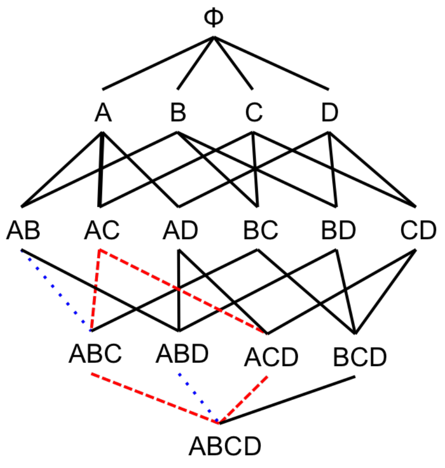
An FD $\varphi : X \rightarrow A$ is valid with respect to a database instance I if there is no violation of φ on I. FD φ is said to be minimal if removing any attribute from X would make it invalid. Moreover, an FD is trivial if its RHS is a subset of its LHS. Since FDs with multiple attributes in the RHS can be equivalently decomposed into multiple FDs with one attribute in the RHS, only FDs with one attribute in the RHS need to be considered. Thus, given a database instance I of schema R, the FD discovery problem is to find all valid minimal nontrivial FDs with one attribute in the RHS that hold on I.

Current FD discovery approaches can be divided into schema-driven and instance-driven approaches. We present an example technique for each category: TANE [71] as an example of a schema-driven approach, and FASTFD [125] as an example of an instance-driven approach. TANE adopts a level-wise candidate generation and pruning strategy and relies on a linear algorithm for checking the validity of FDs. On the other hand, FASTFD first computes difference sets from data, then adopts a heuristic-driven depth-first search algorithm to search for covers of difference sets. TANE is sensitive to the size of the schema, while FASTFD is sensitive to the size of the instance.

TANE Assume the relational schema R has m attributes; $|R| = m$. Selecting an attribute as the RHS of an FD, any subset of the remaining $m-1$ attributes could serve as the LHS. Thus, the space to be explored for FD discovery is $m \times 2^{m-1}$. Figure 2.2a shows the space of candidate FDs organized in a lattice for a table with four columns, A, B, C, and D, with every edge in the lattice represents a candidate FD. For example, edge A to AC represents the FD $A \rightarrow C$.



(a) Space of FDs.



(b) Candidate FDs pruned if $A \rightarrow C$ is valid

Figure 2.2: TANE

Algorithm 1 describes TANE [71]. TANE searches the lattice level by level. The level-by-level traversal ensures that only minimal FDs are in the output. There are three types of pruning employed by TANE: (1) If $X \rightarrow A \in \Sigma$, then all FDs of the form $XY \rightarrow A$ are implied, and hence they can be pruned; (2) If $X \rightarrow A \in \Sigma$, then all FDs of the form $XAY \rightarrow B$ can be pruned. The reason is that if $XY \rightarrow B$ is a valid FD, then $XAY \rightarrow B$ is implied by $XY \rightarrow B$, which would be discovered earlier due to level-by-level traversal, and if $XY \rightarrow B$ is not a valid FD, then $XAY \rightarrow B$ is also not valid due to $X \rightarrow A$; and (3) If X is a key, then any node containing X can be pruned.

Partitioning I by X produces a set of nonempty disjoint subsets denoted as Π_X, and each subset contains identifiers of all tuple in I sharing the same value for attributes X. An FD $X \rightarrow A$ is valid if and only if $|\Pi_X| = |\Pi_{X \cup A}|$, where $|\Pi_X|$ denotes the number of disjoint subsets in Π_X. The partitions need not be computed from scratch for every set of attributes, rather, TANE computes Π_{XY} from two previously computed partitions, Π_X and Π_Y. Note that Π_{XY} contains all subsets of tuples, where each subset is in both Π_X and Π_Y. For example, if $\Pi_X = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$ and $\Pi_Y = \{\{t_1, t_2, t_3\}, \{t_4\}\}$, then

Algorithm 1 TANE

```
Input: One relational instance $I$, schema $R$
Output: All minimal FDs $\Sigma$
 1: $L_1 \leftarrow \{\{A\}|A \in attr(R)\}$
 2: $l \leftarrow 1$
 3: while $L_l \neq \emptyset$ do
 4:    for all Node $Y \in L_l$ do
 5:       for all Parent node $X$ of $Y$ do
 6:          if $X \rightarrow Y - X$ is valid then
 7:             add $X \rightarrow Y - X$ to $\Sigma$
 8:    pruning $L_l$ based on the three pruning rules
 9:    $L_{l+1} \leftarrow$ generate next level based on $L_l$
10:    $l \leftarrow l + 1$
```

$\Pi_{XY} = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$. Therefore, TANE needs only to compute partitions for every single attribute $A \in R$, partitions for every set of attributes X can be computed from a previous level following the level-by-level traversal.

FASTFD FASTFD [125] is an instance-based FD discovery algorithm. We start by defining the difference set of two tuples t_1, t_2 as $D(t_1, t_2) = \{A \in attr(R) \mid t_1[A] \neq t_2[A]\}$. The difference sets of I are $D_I = \{D(t_1, t_2) \mid t_1, t_2 \in I,\ D(t_1, t_2) \neq \emptyset\}$. Given a fixed $A \in attr(R)$, the difference sets of I modulo A are $D_I^A = \{D - \{A\} \mid D \in D_I$, and $A \in D\}$. An FD $X \rightarrow A$ is a valid FD if and only if X covers D_I^A, *i.e.*, X intersects with every element in D_I^A. The intuition is that if X intersects with every element in D_I^A, then X distinguishes any two tuples that disagree on A.

Example 2.2. Consider a table I of R with four attributes as follows:

	A	B	C	D
t_1	a_1	b_1	c_1	d_1
t_2	a_2	b_1	c_1	d_2
t_3	a_1	b_2	c_2	d_1

We have $D(t_1, t_2) = \{AD\}$, $D(t_1, t_3) = \{BC\}$, and $D(t_2, t_3) = \{ABCD\}$. Thus, $D_I = \{AD, BC, ABCD\}$, and $D_I^A = \{D, BCD\}$. Since $\{D\}$ is a minimal cover of D_I^A, we have $D \rightarrow A$.

Therefore, the problem of finding all valid FDs is transformed to the problem of finding all minimal set covers of D_I^A for every attribute $A \in attr(R)$. Every subset of $attr(R) - A$ is a potential candidate minimal cover of D_I^A. Algorithm 2 describes FASTFD. In the following, we describe the depth-first search (Line 4) of Algorithm 2 using the table in Example 2.2.

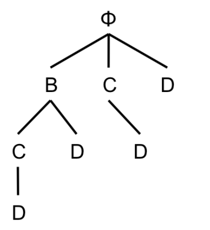
Algorithm 2 FASTFD

```
Input: One relational instance $I$, schema $R$
Output: All minimal FDs $\Sigma$
1: for all $A \in attr(R)$ do
2:     calculate $D_I^A$
3: for all $A \in attr(R)$ do
4:     Finding all minimal set covers of $D_I^A$ using a depth-first search
5:     For every cover $X$, add $X \to A$ to $\Sigma$
```

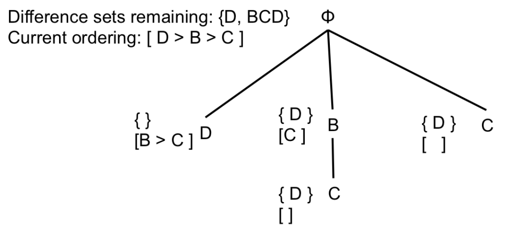
To generate all possible minimal set covers for D_I^A, that is all subsets of $\{BCD\}$, without repetition, the attributes are lexically ordered, *i.e.*, $B > C > D$, and arranged in a depth-first search tree, as shown in Figure 2.3a. An improved version of the search orders the remaining attributes dynamically according to how many difference sets they cover. Ties are broken lexically. For example, to search for minimal covers of D_I^A using $\{BCD\}$, the attributes are ordered $D > B > C$, since D covers two difference sets, while B and C cover one difference set, as shown in Figure 2.3b. If the algorithm reaches at a node where there are no remaining difference sets left, we have reached a cover X, which may not be minimal. If every immediate subset of X is not a cover, then X is minimal. If a node is reached where there are still remaining difference sets, but no attributes left, the depth-first search procedure terminates.

2.1.1.2 Conditional Functional Dependencies

FDs are not sufficient to capture certain semantics of data. Conditional functional dependencies (CFDs), an extension of FDs, are capable of capturing FDs that hold partially on the data [18].



(a) Static ordering of attributes.


Difference sets remaining: {D, BCD}
Current ordering: [D > B > C]
Φ
{ }
[B > C] D
{ D }
[C] B
{ D } C
[ ]
{ D } C
[ ]

(b) Dynamic ordering of attributes

Figure 2.3: FASTFD

Definition 2.2. A CFD φ on R is a pair $(R : X \to Y, T_p)$, where:

- $X, Y \subset R$;
- $X \to Y$ is an FD, called *embedded FD* in the context of CFD; and
- T_p is called a *pattern tableau* of φ, where for every attribute $A \in X \cup Y$ and each pattern tuple $t_p \in T_p$, either $t_p[A]$ is a constant in the domain $Dom(A)$ of A, or $t_p[A]$ is a wild card '-'.

A tuple $t_\alpha \in I$ is said to *match* a pattern tuple $t_p \in T_p$, denoted as $t_\alpha \approx t_p$, if for every attribute $A \in X \cup Y$, $t_\alpha[A] = t_p[A]$, in case $t_p[A]$ is a constant. A relation instance I of R is said to satisfy a CFD φ, denoted as $I \models \varphi$, if for every tuple $t_\alpha, t_\beta \in I$, and for each tuple $t_p \in T_p$, if $t_\alpha[X] = t_\beta[X] \approx t_p[X]$, then $t_\alpha[Y] = t_\beta[Y] \approx t_p[Y]$.

Intuitively, a CFD is a traditional FD with an added constraint of the pattern tableau. If, for two tuples $t_\alpha, t_\beta \in I$, $t_\alpha[X]$ and $t_\beta[X]$ are equal and they both match $t_p[X]$, then $t_\alpha[Y]$ and $t_\beta[Y]$ must also be equal and must both match the pattern $t_p[Y]$.

Example 2.3. Consider a table of sales records in Table 2.2 and an FD {name, type, country} $\to$ {price, tax}. The FD does not hold on the entire relation (*e.g.*, t_6 and t_7 are violating this FD), but it holds if (1) type is "clothing"; (2) country is "France" and type is "book"; or (3) country is "UK".

TID	name	type	country	price	tax
t_1	Harry Potter	book	France	10	0
t_2	Harry Potter	book	France	10	0
t_3	Harry Potter	book	France	10	0.05
t_4	The Lord of the Rings	book	France	25	0
t_5	The Lord of the Rings	book	France	25	0
t_6	Algorithms	book	USA	30	0.04
t_7	Algorithms	book	USA	40	0.04
t_8	Armani suit	clothing	UK	500	0.05
t_9	Armani suit	clothing	UK	500	0.05
t_{10}	Armani slacks	clothing	UK	250	0.05
t_{11}	Armani slacks	clothing	UK	250	0.05
t_{12}	Prada shoes	clothing	USA	200	0.05
t_{13}	Prada shoes	clothing	USA	200	0.05
t_{14}	Prada shoes	clothing	France	500	0.05
t_{15}	Spiderman	DVD	UK	19	0
t_{16}	Star Wars	DVD	UK	29	0
t_{17}	Star Wars	DVD	UK	25	0
t_{18}	Terminator	DVD	France	25	0.08
t_{19}	Terminator	DVD	France	25	0
t_{20}	Terminator	DVD	France	20	0

Table 2.2: Sales data records [60]

The constraints, however, can be expressed by the CFD ($\{$name, type, country$\} \rightarrow \{$price, tax$\}, T_p$), with T_p as shown in Figure 2.4.

While it requires two tuples to have a violation of an FD, one tuple may also violate a CFD. A single tuple t violates a CFD if t matches the LHS of a tuple t_p in the pattern tableaux, but not the RHS, where t_p consists of all constants, *i.e.*, no wild cards, traditionally referred to as "tuple check constraint".

CFD Discovery

Similar to FD discovery, we aim at discovering nontrivial, minimal CFDs with only one attribute in the RHS that hold on a given database instance. The CFD discovery problem is challenging for two reasons:

name	type	country	price	tax
-	clothing	-	-	-
-	book	France	-	0
-	-	UK	-	-

Figure 2.4: T_p for the CFD $(\{\text{name, type, country}\} \to \{\text{price, tax}\}, T_p)$

(1) the number of all possible embedded FDs is exponential in the number of attributes in the schema, which is shared by the problem of FD discovery; and (2) the number of all possible constants in the pattern tableaux is huge, a challenge unique for CFD discovery.

Candidate CFDs can be generated according to the same lattice used in FD discovery (cf. Figure 2.2a) [22]. Unlike FD discovery, where each edge in the lattice corresponds to one candidate FD, in CFD discovery, each edge corresponds to multiple candidate CFDs, depending on which attributes are conditioned on. Specifically, an edge (X, XA) generates CFDs of the form $[Q, P] \to A$, consisting of *variable attributes* P and *conditional attributes* Q, where $X = P \cup Q$. The conditional attributes are those attributes that appear as constants in T_p. The same strategy used in TANE is employed to traverse the lattice level by level, and to reduce the computation at each level by using the results from previous levels [22]. Several interestingness measures, *e.g.*, support, χ^2 test, and conviction, are proposed for discovered CFDs to avoid returning an unnecessarily large number of CFDs.

Three CFD discovery methods that require that the number of tuples matching the pattern tableaux should be above a minimum threshold were proposed [45]. The first method, named CFDMiner, aims at discovering constant CFDs, *i.e.*, CFDs with the pattern tableaux containing only constants. CFDMiner leverages the similarity between the problem of discovering constant CFDs and the problem of mining free and closed itemsets: constant CFDs correspond to association rules with 100% confidence. The second method, named CTANE, extends TANE for FD discovery and uses a level-wise search strategy, which is similar to the search strategy used in [22]. The third method, FASTCFD, extending FASTFD for FD discovery, employs a depth-first search strategy. A novel pruning strategy used in FASTCFD is to use constant CFDs already discovered by CFDMiner. CTANE is sensitive to the number of attributes in the schema; while FASTCFD is sensitive to the number of tuples in the database.

If the embedded FD is given, the CFD discovery problem becomes that of generating a near-optimal pattern tableaux [60]. The "goodness" of the pattern tableaux is defined by the support and the confidence, where the support of a pattern tableaux is defined as the fraction of tuples in the database that match the LHS of the pattern tuples in the pattern tableaux, and the confidence of a pattern tableaux is defined as the maximum fraction of tuples in the database that do not violate the pattern tableaux. The optimal pattern tableaux generation problem will be further discussed in Section 3.1.2.

2.1.1.3 Denial Constraints

As powerful as CFDs are, they are still not capable of capturing many real life data quality rules, such as the fourth rule, that is "if two persons live in the same state, the one with lower salary has a lower tax rate", and the fifth rule, that is "tax exemption is less than the salary", in Example 1.1. Denial constraints (DCs), a universally quantified first order logic formalism, which subsume FDs and CFDs, can describe all data quality rules in Example 1.1.

Definition 2.3. A denial constraint (DC) φ on R is defined as: $\forall t_\alpha, t_\beta, t_\gamma, \ldots \in R, \neg(P_1 \wedge \ldots \wedge P_m)$, where each *predicate* P_i is of the form $v_1 \theta v_2$ or $v_1 \theta c$ with $v_1, v_2 \in t_x.A, x \in \{\alpha, \beta, \gamma, \ldots\}, A \in R$, c is a constant in the domain of A, and $\theta \in \{=, <, >, \neq, \leq, \geq\}$.

A relation instance I of R is said to satisfy a DC φ, denoted as $I \models \varphi$, if for every ordered list of tuples $\forall t_\alpha, t_\beta, t_\gamma, \ldots \in I$, at least one of P_i is false.

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

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.

Example 2.4. DCs are expressive enough to capture all data quality rules in Example 1.1 as follows:

$c_1 : \forall t_\alpha, t_\beta \in R, \urcorner(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$
$c_2 : \forall t_\alpha, t_\beta \in R, \urcorner(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$
$c_3 : \forall t_\alpha \in R, \urcorner(t_\alpha.CT = \text{‘Los Angeles’} \wedge t_\alpha.ST \neq \text{‘CA’})$
$c_4 : \forall t_\alpha, t_\beta \in R, \urcorner(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$
$c_5 : \forall t_\alpha \in R, \urcorner(t_\alpha.SAL < t_\alpha.STX)$

DC Discovery

FASTDC [27, 25] is proposed as an extension of FASTFD for DC discovery. In order to define the space of DCs, we first need to define the *space of predicates* $\mathbf{P}$, since DCs are composed of predicates. Then, the *evidence set* Evi_I is built. The *evidence set* Evi_I is a set, where each element in Evi_I is a subset of predicates in $\mathbf{P}$ that are satisfied by a tuple pair in I. The evidence set has a similar functionality to the difference set in FASTFD, in that each minimal set cover for Evi_I corresponds to a valid minimal DC.

Example 2.5. 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

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 we add cross column predicates accordingly. We build the following predicate space $\mathbf{P}$ for it:

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

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 $SAT(\langle t_x, t_y \rangle) = \{P | P \in \mathbf{P}, \langle t_x, t_y \rangle \models P\}$, where $\mathbf{P}$ 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 $\mathbf{X} \subseteq \mathbf{P}$ is a *minimal set cover* for Evi_I if $\forall E \in Evi_I, \mathbf{X} \cap E \neq \emptyset$, and $\nexists \mathbf{Y} \subset \mathbf{X}$, s.t. $\forall E \in Evi_I, \mathbf{Y} \cap E \neq \emptyset$.

Example 2.6. $Evi_{Emp} = \{\{P_2, P_3, P_5, P_8, P_{10}, P_{12}, P_{14}\}, \{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}, \{P_2, P_3, P_6, P_7, P_{10}, P_{11}, P_{13}\}\}$. 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.

$\mathbf{X}_1 = \{P_2\}$ 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.

$\mathbf{X}_2 = \{P_{10}, P_{14}\}$ 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.

Algorithm 3 FASTDC

```
Input: One relational instance $I$, schema $R$
Output: All minimal DCs $\Sigma$
1: $\mathbf{P} \leftarrow$ building the predicate space based on $R$ and $I$
2: $Evi_I \leftarrow$ building the evidence set based on $I$ and $P$
3: $\mathbf{MC} \leftarrow$ search for all minimal covers of $Evi_I$
4: for all $\mathbf{X} \in \mathbf{MC}$ do
5:     $\Sigma \leftarrow \Sigma + \neg(\overline{\mathbf{X}})$
6: rank DCs in $\Sigma$ based on their interestingness
```

Algorithm 3 describes FASTDC, which first builds the space of predicates and the evidence set, then searches for all minimal set covers for the evidence set. Every minimal set cover corresponds to a minimal DC. In the end, all discovered DCs are ranked according to their *interestingness*, which is a linear combination of *succinctness* and *coverage* [25].

2.1.1.4 Other Types of Constraints

Multiple other types of constraints have been proposed for different purposes: Inclusion Dependencies (INDs) [1] can be used for detecting inconsistencies or information incompleteness, and schema matching; Matching dependencies (MDs) [49] use similarity measures to generalize the equality condition used in FDs, to support record linkage across two tables; Metric functional dependencies (MFDs) [80] can be considered as special MDs defined on one table, to capture small variations in the data; Numeric functional dependencies (NFDs) [43] can capture interesting constraints involving numeric attributes, since NFDs allow arithmetic operations; Editing rules (eRs) [50] not only provides a way to detect errors, but also tells how to fix errors by referencing a master table; Fixing rules [117] precisely captures which attribute is wrong, and how to correct the error, when enough evidence is present; Sherlock Rules [72] annotate the correct and erroneous attributes, and precisely tell how to fix the errors by referencing master tables.

Inclusion Dependency

Inclusion Dependencies (INDs) [1], which are a generalization of referential constraints, can be used for detecting inconsistencies or information incompleteness [17] and schema matching systems (e.g., [65]).

Definition 2.4. An inclusion dependency φ for two relations (R_1, R_2) is defined as $R_1[X] \subseteq R_2[Y]$, where (X_1, X_2) are lists of attributes in (R_1, R_2), and $|X| = |Y|$. An instance pair (I_1, I_2) satisfies IND φ if for any tuple $t_\alpha \in I_1$, there exists a tuple $t_\beta \in I_2$, such that $t_\alpha[X] = t_\beta[Y]$.

In other words, for any $t_\alpha \in I_1$, if there is no tuple $t_\beta \in I_2$ that satisfies $t_\alpha[X] = t_\beta[Y]$, then either $t_\alpha \in I_1$ is incorrect, or there is a tuple missing in R_2. Inclusion dependencies may also refer to only one relation, i.e., R_1 is the same as R_2.

Example 2.7. Consider the relation about tax records in Example 1.1 as Relation R_1, and a second employee relation

$R_2(\mathsf{EID}, \mathsf{FirstName}, \mathsf{LastName})$ that keeps all the employee IDs, and their first and last names. A valid IND would be $R_1[\mathsf{FN}, \mathsf{LN}] \subseteq R_2[\mathsf{FirstName}, \mathsf{LastName}]$, which means that if there is a tax record about a person, then that person must appear in the employee table.

De Marchi et al. [34] discover unary INDs, that is, INDs with one attribute in X and Y, using a two step process by first building an inverted index pointing every value in the database to the set of all attributes containing the value, and then retrieving valid INDs using set intersections. N-ary INDs are discovered following a level-wise approach similar to TANE. BINDER [92] does not assumes that the data set fits into main memory, and aims at discovering INDs in a scalable manner based on a divide and conquer strategy.

INDs are later extended to conditional inclusion dependencies (CINDs) [84], which are INDs that hold on subset of the tuples. Just like the extension of CFDs to FDs, CINDs have more expressive power than INDs.

Matching Dependency

Matching dependencies (MDs) [49] use similarity measures to generalize the equality condition used in FDs. While FDs are defined on a single relation, MDs are defined on two relations.

Definition 2.5. A matching dependency φ for two relations (R_1, R_2) is defined as follows:

$$\wedge_{j \in [1,k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2],$$

where

- (X_1, X_2) are lists of attributes in (R_1, R_2), $X_1[j], X_2[j]$ denotes the j^{th} attribute in X_1, X_2;
- for every $j \in [1, k]$, $X_1[j]$ and $X_2[j]$ are comparable, *i.e.*, they belong to the same domain;
- for every $j \in [1, k]$, $\approx_j$ is a similarity operator, which can be any similarity metric used in record matching, *e.g.*, q-grams or edit distance;

- $\rightleftharpoons$ is a matching operator. For any two values x, y, $x \rightleftharpoons y$ indicates that x and y are changed to be identical.

Intuitively, an MD φ states that if $R_1[X_1]$ and $R_2[X_2]$ are similar with respect to some similarity metrics, then $R_1[Z_1]$ and $R_2[Z_2]$ should be changed to be identical.

Example 2.8. Consider two relational tables from a bank in the U.K.: $\mathsf{card(FN, LN, St, city, AC, zip, tel, dob, gd)}$ maintains customer information collected when credit cards are issued; and $\mathsf{tran(FN, LN, St, city, AC, post, phn, gd, item, when, where)}$ consists of transaction records of credit cards, which may be dirty. Here a card tuple specifies a U.K. credit card holder identified by first name (FN), last name (LN), address (street (St), city, zip code), area code (AC), phone (tel), date of birth (dob) and gender (gd). A tran tuple is a record of a purchased item paid by a credit card at place where and time when, by a U.K. customer who is identified by name (FN, LN), address (St, city, post code), AC, phone (phn), and gender (gd).

A possible matching rule is that for any tuple in card and any tuple in tran, if they have the same last name and address, and moreover, if their first names are *similar*, then their phone and FN attributes can be identified. This rule can be expressed by an MD ψ: $\mathsf{tran[LN, city, St, post]} = \mathsf{card[LN, city, St, zip]} \wedge \mathsf{tran[FN]} \approx \mathsf{card[FN]} \rightarrow \mathsf{tran[FN, phn]} \rightleftharpoons \mathsf{card[FN, tel]}$.

The MD discovery problem also aims at discovering interesting MDs with high support and confidence [102], which is similar to CFD discovery [22]. Only interested MDs are selected as output.

Metric Functional Dependency

While MDs are for capturing small variations on string attributes, Metric functional dependencies (MFDs) [80] are usually used to capture small variations on numerical data.

Definition 2.6. A metric functional dependency (MFD) φ is defined as $X \xrightarrow{\delta} Y$, where X and Y denote subsets of attributes of $attr(R)$. An instance I of R satisfies this FD φ, denoted as $I \models \varphi$ if for any two

Source	Title	Duration
movies.aol.com	Aliens	110
finnguide.fi	Aliens	112
amazon.com	Clockwork Orange	137

Table 2.3: Movie data records integrated from multiple data sources.

tuples t_α, t_β in I such that $t_\alpha[X] = t_\beta[X]$, then $d(t_\alpha[Y], t_\beta[Y]) \leq \delta$, where d is a metric function defined on the domain of Y.

An MFD can be seen as a special case of MD where R_1 and R_2 are the same relation and the LHS is exact matching.

Example 2.9. Consider a table of movies in Table 2.3, resulting from integrating movies from multiple websites. A plausible constraint is that the same movie should have the same duration. However, different websites may have different ways of calculating the duration, for example, depending on whether extra materials, such as the advertisement, are included. An MFD Title $\xrightarrow{5}$ Duration would be more suitable than the FD Title $\rightarrow$ Duration to capture such a constraint.

Numeric Functional Dependency

Numeric functional dependencies (NFDs) [43] are another type of constraints for capturing constraints involving numeric attributes. They are able to capture errors in numeric attributes that FDs, CFDs, and DCs cannot capture.

Definition 2.7. A numeric functional dependency (NFD) φ defined on a relational table $R(A_1, \ldots, A_m)$ is a pair of tables:

- a *pattern table* T_p of schema R that has two tuples p_1 and p_2; for $i \in [1, 2]$ and $j \in [1, m]$, $p_i[A_j]$ is a constant in $dom(A_j)$, a variable x, or a wildcard '_'; and
- a *condition table* T_c with a single *condition tuple* of the form $eopz$, where e is either a variable in T_p or a linear arithmetic expression of variables in T_p, op is one of the operations in $\{=, \neq, <, \leq, >, \geq\}$, and z is either a constant or a variable in T_p.

name	YoB	YoD	town	country
_	x	y	_	_

(a) Pattern table T_{P1}

condition
$y - x \leq 120$

(b) Condition table T_{C1}

SS#	name	cno	hw	tests	lab	proj
_	_	_	x_1	x_2	x_3	x_4

(c) Pattern table T_{P2}

condition
$x_1 + x_2 + x_3 + x_4 = 100$

(d) Condition table T_{C2}

	CC#	name	street	city	zip	when	where	amnt
p_1:	x_c	_	_	_	_	x_t	Edi	_
p_2:	x_c	_	_	_	_	y_t	NYC	_

(e) Pattern table T_{P3}

condition
$\lvert x_t - y_t \rvert \geq 2$

(f) Condition table T_{C3}

Figure 2.5: NFDs examples [43].

As we can see, NFDs are defined on at most two tuples, and it can express more constraints involving numeric attributes than FDs, CFDs, and DCs because NFDs allow arithmetic operations.

Example 2.10. Figure 2.5 shows three NFDs on three different tables. (1) The first table specifies a person with his name, year of birth (YoB), year of death (YoD), and origin (country, town). The first NFD with T_{P1} in Figure 2.5a and T_{C1} in Figure 2.5b says that no one can live more than 120 years. (2) The second table specifies the academic report of courses of a student, with the distribution of the score into homework (hw), tests, lab, and projects (proj). The second NFD with T_{P2} in Figure 2.5c and T_{C2} in Figure 2.5d says that the total percentage has to be equal to 100. (3) The third table is about credit card transactions with each tuple specifying the card number (CC#), the hard holder information (name, street, city, zip), when and where the card was used, and the amount charged to the card (amnt). The third NFD, with T_{P3} in Figure 2.5e and T_{C3} in Figure 2.5f , asserts that two transactions involving the same credit card, one happening in Edinburgh (Edi) and one happening in New York city (NYC), has to be at least two hours apart.

Editing Rules

Editing rules (eRs) [50] not only provides a way to detect errors, but also tells how to fix errors by referencing a master table.

Definition 2.8. An editing rule (eR) φ defined on a relation R and a master relation R_m is a pair $\big((X, X_m) \to (B, B_m), t_p[X_p]\big)$, where:

- X and X_m are two lists of distinct attributes in R and R_m respectively, with the same number of attributes;
- B is an attribute in $attr(R) - X$, and B_m is an attribute in $attr(R_m) - X_m$; and
- t_p is a pattern tuple over a set of distinct attributes X_p in R, such that for each $A \in X_p$, $t_p[A]$ is one of $\_$, a or $\overline{a}$, where a is a constant from the domain of A, $\overline{a}$ is any constant other than a, and $\_$ is an unnamed variable.

An eR φ is said to be applicable to a tuple $t \in I$ and a tuple $t_m \in I_m$ to update t to t', denoted as $t \to t'$ if:

- t and t_m matches on the LHS of φ, *i.e.*, $t[X] = t_m[X_m]$;
- t matches t_p, *i.e.*, $t[X_p] \approx t_p[X_p]$; and
- t' is obtained from t by updating $t[B]$ to be $t_m[B_m]$.

CFDs and eRs are both based on pattern tuples. CFDs are defined on a single relation, while eRs are defined on an input tuple and a master relation. In addition, while CFDs have *static semantics*, *i.e.*, they only tell whether two tuples are in violation or not, eRs have *dynamic semantics*, *i.e.*, they update t to t' if an eR is applicable.

eRs are also similar to MDs in that they both share dynamic semantics. Although both MDs and eRs are defined on two relations, MDs neither have pattern tuples nor master data relation. For two tuples t_1 and t_2, an MD $\wedge_{j\in[1,k]}(R_1[X_1[j]] \approx_j R_2[X_2[j]]) \to R_1[Z_1] \rightleftharpoons R_2[Z_2]$ only states that $t_1[Z_1]$ and $t_2[Z_2]$ should be identified, but it does not tell what values are to be taken; however, eRs directly dictate that values from the master relation should be taken.

Example 2.11. Consider the same two relational tables as defined in Example 2.8, namely, $\mathsf{card}(\mathsf{FN}, \mathsf{LN}, \mathsf{St}, \mathsf{city}, \mathsf{AC}, \mathsf{zip}, \mathsf{tel}, \mathsf{dob}, \mathsf{gd})$ and $\mathsf{tran}(\mathsf{FN}, \mathsf{LN}, \mathsf{St}, \mathsf{city}, \mathsf{AC}, \mathsf{post}, \mathsf{phn}, \mathsf{gd}, \mathsf{item}, \mathsf{when}, \mathsf{where})$. Assume that card is a *clean master relation*, that is, tuples in card are correct.

An plausible eR φ is that for any tuple t in tran, if there exists a master tuple s in card with $t[\mathsf{LN}, \mathsf{FN}, \mathsf{city}, \mathsf{St}, \mathsf{post}] = s[\mathsf{LN}, \mathsf{FN}, \mathsf{city}, \mathsf{St}, \mathsf{zip}]$, the $t[\mathsf{phn}]$ should be updated to be $s[\mathsf{phn}]$, which can be written as φ: $((X, X_m) \rightarrow (\mathsf{phn}, \mathsf{phn}), t_p[X_p] = ())$, where X ranges over $\mathsf{LN}, \mathsf{FN}, \mathsf{city}, \mathsf{St}, \mathsf{post}$, and X_m ranges over $\mathsf{LN}, \mathsf{FN}, \mathsf{city}, \mathsf{St}, \mathsf{zip}$.

Editing rules discovery is studied in [37], which adapts techniques from CFD discovery.

Fixing Rules

Similar to eRs, Fixing rules [117] not only precisely capture which attribute is wrong, but also indicate how to correct the error, when enough evidence is present.

Definition 2.9. A fixing rule φ defined on a schema R is formalized as $((X, t_p[X]), (B, T_p^-[B])) \rightarrow t_p^+[B]$ where

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

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

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

Figure 2.6: Database D: an instance of schema Travel

value (*i.e.*the fact) is different from known wrong values (*i.e.*negative patterns) relative to a specific evidence pattern.

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

Example 2.12. Consider a table of travel records, shown in Figure 2.6, for a research institute, specified by the following schema: Travel (name, country, capital, city, conf). A Travel tuple specifies a person, identified by name, who has traveled to conference (conf), held at the city of the country with capital. All errors are highlighted and their correct values are given between brackets. For instance, r_2[capital] = `Shanghai` is wrong, and its correct value is `Beijing`.

Consider the following two fixing rules defined on Travel: φ_1 : (([country], [`China`]), (capital, {`Shanghai`,`Hongkong`})) $\to$ `Beijing`, and φ_2 : (([country], [`Canada`]), (capital, {`Toronto`})) $\to$ `Ottawa`.

In both φ_1 and φ_2, X consists of country and B is capital. Here, φ_1 states that, if the country of a tuple is `China` and its capital value is in {`Shanghai, Hongkong`}, its capital value is wrong and should be updated to `Beijing`. Similarly for φ_2. Tuple r_1 does not match rule φ_1, since r_1[country] = `China` but r_1[capital] $\notin$ {`Shanghai, Hongkong`}. As another example, tuple r_2 matches rule φ_1, since r_2[country] = `China`, and r_2[capital] $\in${`Shanghai, Hongkong`}. Similarly, we have r_4 matches φ_2. After applying φ_1 and φ_2, two errors, r_2[capital] and r_4[capital], can be repaired.

To the best of our knowledge, the automatic discovery for fixing rules has not been studied.

Sherlock Rules

Sherlock Rules [72] annotate the correct and erroneous attributes, and precisely tell how to fix the errors by referencing master tables. Let I be a table over schema R, and M a reference table with schema R_m. The relational schema R is often different from R_m.

Definition 2.10. A Sherlock Rule (sR) φ defined on schemas (R, R_m) is formalized as $\varphi : ((X, X_m), (B, B^-_m, B^+_m), \vec{\approx})$ where:

- X and X_m are lists of distinct attributes in schemas R and R_m respectively, where $|X| = |X_m|$;
- B is an attribute such that $B \in R \setminus X$, and B^-_m, B^+_m are two distinct attributes in $R_m \setminus X_m$; and
- $\vec{\approx}$ is a vector of similarity operators over comparable attributes, (A, A_m), (B, B^-_m) and (B, B^+_m), where $A \in X$, and A_m is the corresponding attribute in X_m.

Rule φ says that for a pair of tuples t in I and t_m in M, if both $(t[X], t_m[X_m])$ and $(t[B], t_m[B^-_m])$ are similar with respect to some similarity metrics, φ validates that $t[X]$ is correct, $t[B]$ is wrong, and moreover, the correct value of $t[B]$ is $t_m[B^+_m]$. Intuitively, given that $t[X]$ and $t_m[X_m]$ are similar, $t[B]$ should take value from $t[B^+_m]$, rather than $t[B^-_m]$. In other words, the rule explicitly captures the possible errors $t[B]$ can make, for example, attributes B and B^+_m might be office phone number, and B^-_m might be mobile phone number.

Example 2.13. Consider the employee table in Figure 2.7, and two reference tables: the capital table in Figure 2.8 and the phone table in Figure 2.9. Three sRs are defined as follows: where φ_1 and φ_2 are defined on (EMP, PHN), and φ_3 is defined on (EMP, CAP):
φ_1: ((name, name), (officePhn, mobile, officePhn), $(=, =, =)$)
φ_2: ((name, name), (officePhn, mobile, $\perp$), $(=, =, \not\approx)$)
φ_3: ((nation, country), (capital, $\perp$, capital), $(=, \not\approx, =)$)

	name	dept	nation	capital	bornat	officePhn
t_1	Si	DA	China	Beijing	ChenYang	28098001
t_2	Yan	DA	China	Shanghai	Chengdu	24038698
t_3	Ian	ALT	Chine	Beijing	Hangzhou	33668323

Figure 2.7: I_{EMP}: An instance of the schema EMP

	country	capital
s_1	China	Beijing
s_2	Japan	Tokyo
s_3	Chile	Santiago

Figure 2.8: M_{CAP}: An instance of the schema CAP

	name	officePhn	mobile
r_1	Si	28098001	66700541
r_2	Yan	24038698	66706563
r_3	Ian	27364928	33668323

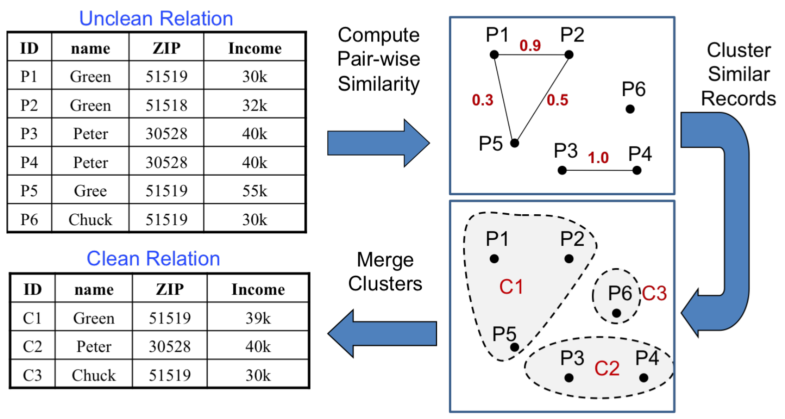
Figure 2.9: M_{PHN}: An instance of the schema PHN

where "$\perp$" indicates that a field is missing, and "$\not\approx$" that the two corresponding attributes are not comparable, *e.g.*, when some attribute is missing from reference tables.

(1) Rule φ_1 states that for a tuple t in I_{EMP}, if its name matches the name of a r tuple in M_{PHN}, and t[officePhn] matches r[mobile], then φ_1 validates that t[name] is correct, and t[officePhn] is wrong. Moreover, it will rectify t[officePhn] to r[officePhn]. Consider t_3 in I_{EMP} and r_3 in M_{PHN}, φ_1 works as follows. Firstly, t_3[name] is matched with r_3[name], and t_3[officePhn] with r_3[mobile]. It then detects that t_3 is about Ian, but someone messed up his office number with his mobile number. Consequently, t_3[name] is marked as correct and t_3[officePhn] as wrong. Since the office number of Ian is available in r_3[officePhn], φ_1 will update t_3[officePhn] to r_3[officePhn], which is 27364928.

(2) Often times, not all evidences are available. Assume that the column officePhn is missing in PHN, namely, consider a revised schema PHN$'$ (name, mobile). Rule φ_2 states that given a tuple t in I_{EMP}, if its name matches the name of a tuple r in $M_{\text{PHN}'}$, and t[officePhn] matches r[mobile], then φ_2 validates that t[name] is correct and t[officePhn] is wrong. Again, consider t_3 in I_{EMP} and r_3 in $M_{\text{PHN}'}$, φ_2 works similar to φ_1, which validates that t_3[name] is correct and t_3[officePhn] is wrong. However, due to the missing column officePhn in PHN$'$, φ_2 cannot update t_3[officePhn].

(3) Rule φ_3 states that for a tuple t in I_{EMP}, if t[country, capital] matches s[country, capital] of an s tuple in M_{CAP}, it will mark t[country, capital] as correct. Consider t_1 in I_{EMP} and s_1 in M_{CAP}. Since



ID	name	ZIP	Income
P1	Green	51519	30k
P2	Green	51518	32k
P3	Peter	30528	40k
P4	Peter	30528	40k
P5	Gree	51519	55k
P6	Chuck	51519	30k

ID	name	ZIP	Income
C1	Green	51519	39k
C2	Peter	30528	40k
C3	Chuck	51519	30k

Figure 2.10: A typical deduplication task.

both country (*i.e.* China) and capital (*i.e.* Beijing) match, φ_3 will mark t_1[country, capital] as correct.

2.1.2 Duplicate Detection

Data deduplication, also known as duplicate detection, record linkage, record matching, or entity resolution, refers to the process of identifying tuples in one or more relations that refer to the same real world entity. The topic has been extensively covered in many surveys [81, 40, 39, 90, 58, 70]: some aim at providing an extensive overview of all the steps involved in data deduplication [40, 58, 70], some focus on the design of similarity metrics [90, 81], some discuss the efficiency aspect of data deduplication [90], and some focus on how to consolidate multiple records [39].

Example 2.14. Figure 2.10 illustrates a typical example of data deduplication. The similarities between pairs of records are computed, and are shown in the similarity graph (upper right graph in Figure 2.10). The missing edges between any two records indicate that they are non-duplicates. Records are then clustered together based on the similarity graph. Suppose the user sets the threshold to be 0.5, *i.e.*, any record pairs having similarity greater than 0.5 are considered duplicates. Although Record P_1 and P_5 have similarity less than 0.5, they are clustered together due to transitivity; that is, they are both considered duplicates to Record P_2. All records in the same cluster are consolidated into one record in the final clean relation.

In this section, we discuss techniques used for detecting duplicate records, including designing similarity metrics, training classifiers to determine whether two records are duplicates, and clustering to identify a set of records that refer to the same entity. We then briefly discuss collective entity resolution. The processing of consolidating multiple records referring to the same entity into a single record is called data fusion, and will be discussed in Section 3.1.1 when we talk about data repairing.

Similarity Metrics

Measuring the similarity between two values in the same column is an essential component in determining whether two records are duplicates, and a variety of similarity metrics have been proposed to handle different types of errors.

Typographical errors are one of the most common type of errors, and multiple character-based similarity metrics, including edit distance, affine gap distance [118], Jaro distance [73], Jaro-Winkler distance [121], can be used to handle typographical errors. The edit distance between two strings s_1 and s_2 is defined as the minimum number of edit operations needed to transform s_1 to s_2; three types of edit operations are usually considered: inserting a character, deleting a character, and replacing one character with another. Levenshtein distance [82] is the simplest type of edit distance, where the cost of each edit operation is equal. The edit distance falls short when comparing strings that have been truncated or expanded (*e.g.*, *"Chris R. Lang"* versus *"Christopher Richard Lang"*), the affine gap distance [118] addresses this problem by introducing two additional edit operations: opening a gap and extending a gap. The cost of opening a gap is usually larger than the cost of extending a gap, which results in a smaller cost for gap mismatches than the cost under the edit distance. Jaro distance [73] is usually used for comparing person names. There are three steps in computing Jaro distance between two strings s_1 and s_2: (1) computing the length of two strings, denoted as $|s_1|$ and $|s_2|$, respectively; (2) finding the number of matching characters m in s_1 and s_2, where two characters $s_1[i]$ and $s_2[j]$ from s_1 and s_2, respectively, are considered matching if $s_1[i]$ is the same as $s_2[j]$, and $|i-j| \leq \frac{1}{2}\min\{|s_1|, |s_2|\}$; and (3) finding the number of transpositions t as follows: the l^{th} matching character in s_1 is compared with the l^{th} matching character in s_2, if they are not the same, the number of transpositions is increased by one. The Jaro distance of s_1 and s_2 is computed as $Jaro(s_1, s_2) = \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m})$. Jaro-Winkler distance [121] uses a prefix scale p to favor two strings that share a common prefix of length l since matching prefixes are generally more important for matching person family names. Formally, the Jaro-Winkler distance of s_1 and s_2 is computed as $Jaro\_Winkler(s_1, s_2) = Jaro(s_1, s_2) + (lp(1 - Jaro(s_1, s_2)))$, where l is the length of the prefix up to a maximum of four characters, and p is a constant scaling factor that does not exceeding 0.25. A standard value for p is 0.1.

While character-based similarity metrics are suitable for handling typographical errors, they often fail to capture the similarity between two strings that use the same set of tokens, but with different ordering (*e.g.*, "James Smith" versus "Smith James"). Multiple token-based similarity metrics can be used to handle such errors, including cosine similarity and Jaccard similarity. WHIRL [30] computes the cosine similarity of two strings s_1 and s_2 that are in the same column A of a database instance I, using *tf-idf* weighting scheme. Each string s is separated into words and each word w has a weighting $v_s(w) = \log(tf_w + 1)\log(idf_w)$, where tf_w is the number of times w appears in s, and $idf_w = \frac{|I|}{n_w}$ with $|I|$ being the number of records in I and n_w being the number of records in I that contain w in Column A. Given two strings s_1 and s_2, and let W be the set of words s_1 and s_2 contain, the cosine similarity is defined as $cosine(s_1, s_2) = \frac{\Sigma_{w \in W} v_{s_1}(w) \times v_{s_2}(w)}{\sqrt{\Sigma_{w \in W} v_{s_1}(w)^2} \times \sqrt{\Sigma_{w \in W} v_{s_2}(w)^2}}$. Let W_1 be the set of words s_1 contain, and W_2 be the set of words s_2 contain, the Jaccard similarity of s_1 and s_2 is calculated as $Jaccard(s_1, s_2) = \frac{|W_1 \cap W_2|}{|W_1 \cup W_2|}$.

Phonetics-based similarity metrics are used to detect the similarity of two strings that are phonetically similar, even if they are not similar according to character-based or token-based similarity metrics (*e.g.*, "Clair" versus "Clare"). Soundex [95] is developed to encode surnames in English for use in censuses. The Soundex code for a surname consists of a letter followed by three numerical digits, where the letter is the first letter of the name and the three digits encode the remaining consonants. Daitch-Mokotoff Soundex (D-M Soundex) [104] is a refinement of Soundex to better match surnames of Slavic and Germanic origin. The New York State Identification and Intelligence System Phonetic Code, commonly known as NYSIIS [107], differs from Soundex in that it does not use numerical digits to replace letters, rather, it replaces consonants with other, phonetically similar letters.

Classifiers

Given two records t_1, t_2, a classifier is invoked to determine whether they are duplicates or not. The classifier is usually a combination of the similarity metrics on various attributes. For example, a classifier for records in Table 1.1 could be *"if* $Jaro(t_1[FN], t_2[FN]) > 0.8$ *and* $Jaccard(t_1[CT], t_2[CT]) > 0.5$*, then* t_1 *and* t_2 *are duplicates"*. The techniques for designing a classifier can be broadly categorized into supervised learning techniques, semi-supervised or active learning techniques, and unsupervised techniques [40].

Supervised learning techniques rely on a training data set in the form of record pairs labeled as matching or non-matching. The similarity scores between record pairs serve as features for training a classifier to be applied to the rest of the data. Examples of classifier models include Naïve Bayes [122], decision tree [20], and support vector machine (SVM) [15].

Active learning techniques alleviate one major drawback of the supervised learning techniques, which may require a large training data set, by actively soliciting user feedback for unlabeled record pairs, which, when labeled, will provide the highest utility to the learner [108, 96, 5]. An active learner starts with a limited training data, and learns a preliminary classifier, which is *certain* about some unlabeled record pairs, and *uncertain* about most others, also known as, the confusion region of a classifier. The active learner will reduce the classifier's confusion by seeking human labels on those most uncertain record pairs. There are two main approaches to evaluate the uncertainty of the prediction on a record pair, namely, classifier specific approach and classifier independent approach. Examples of classifier specific approach include using posterior probabilities of predications for Bayesian classifiers [109], and using the inverse of the distance between an instance and the separator for SVM [99]. A classifier independent approach to derive the uncertainty is done by measuring the disagreement among the predications of a set of classifiers, also known as a *committee* [100]. The classifiers in the committee are different from each other, but have similar accuracy on the training data. A certain record pair would get same predications from almost all committee members, while an uncertain record pair would result in disagreement, which can be quantified in various ways, such as entropy on the fraction of committee members that predicate either duplicates or non-duplicates.

Unsupervised techniques decide on matching pairs without the need for a training data set. Examples include using a pre-specified threshold on a distance function [89, 21] and employing domain specific rules [69, 119, 38], such as: *If the first name and last name of two persons are similar, then the two records are duplicates.*

Clustering

The result of pairwise comparison process, which takes $O(n^2)$ comparisons for a database of n records, can be represented as a graph, where nodes represent records, and edges between nodes exist if they are considered duplicates by the classifiers. Each edge may also have a weight, reflecting the confidence of the two nodes connected by the edge being duplicates, *e.g.*, the similarity between the two records. The graph is thus referred to as *similarity graph.* The goal of clustering is to partition all records into disjoint clusters of records, where each cluster corresponds to one real-world entity, and records in a cluster are different representations of the same entity [111, 63].

One simple way to obtain such clustering of records is to leverage the transitivity of duplicate records, that is, if Record A is a duplicate of Record B, and Record B is a duplicate of Record C, then Record A is a duplicate of Record C. Then the clustering problem becomes the problem of finding all connected components in the similarity graph. Each connected component is one cluster that represents one real-world entity [68]. One major drawback of such approach is that it may mistakenly consider two records as duplicates since they are in the same connected component, even though they are very dissimilar.

Correlation clustering provides a method for clustering all records into the optimum number of clusters without specifying that number in advance [42]. The objective of correlation clustering is to maximize the sum of similarities between nodes within the same cluster, and the dissimilarities between nodes in different clusters. Correlation clustering can be viewed as an integer linear programming (ILP) problem. Let $e_{xy} \in \{0, 1\}$ denote if two records x and y are in the same cluster, let $w^{+}_{xy} \in [0, 1]$ denote the cost of clustering x and y together, and let $w^{-}_{xy} \in [0, 1]$ denote the cost of placing x and y in two different clusters. Thus correlation clustering is to minimize the objective function $\Sigma_{xy}(e_{xy}w^{+}_{xy} + (1 - e_{xy})w^{-}_{xy})$, subject to the constraint $\forall x, y, z, e_{xy} + e_{xz} + e_{yz} \neq 2$ that ensures the transitivity property. Since solving the ILP problem is NP-hard [3], a number of greedy approaches have been proposed [42, 3] that generally work in the following steps: (1) all records are randomly permuted; (2) each record x is either assigned to an existing cluster or a new cluster, according to a certain criterion, such as assigning x to a cluster that contain the closest match to x [91], assigning x to a cluster that contains the most recent record y with $w^{+}_{xy} > 0$ [103], and assigning x to a cluster that minimizes the objective function [41]; and (3) run the greedy approach for multiple times, and choose the run that results in the best objective value.

Collective Entity Resolution

In the techniques discussed so far, the similarity between two records is computed based on the attributes of those two records. Sometimes, there is additional relational information in the database that can be exploited to determine whether two records are duplicates. Techniques that exploit these additional relational information are referred to as *collective entity resolution.*

Example 2.15. Consider the task of identifying duplicates in two tables: Authors and Papers. Suppose we have the additional Table CoAuthors, modeling the coauthorship of two authors in Authors, which can be used to determine whether two tuples in Authors are duplicates, for example, two tuples in Authors are more likely to be duplicates if they have a common coauthor. Similarly, Table Citations that models the citation information among papers could be useful for detecting duplicate tuples in Papers.

Bhattacharya and Getoor [13] model collective entity resolution as a clustering problem, and encodes the additional relational information in the similarity computation between two clusters. Deduplog [6] proposes a declarative way of specifying the additional hard constraint, such as "two authors of the same paper have to be different", as well as soft constraint, such as "two authors are more likely to be duplicates if they have a common coauthor", and aims at minimizing the number of soft constraints violated, while ensuring no hard constraints are violated. Multiple proposals use probabilistic models for solving collective entity resolution, such as conditional random field [86] and markov logic network [101]. The tutorial[1] by Getoor and Machanavajjhala [58] gives a more comprehensive treatment on collective entity resolution.

2.2 How to Detect

Since the notion of violation with respect to an IC is well defined, namely, the minimal subset of database cells that cannot coexist, violation detection for ICs can be achieved automatically [18, 26]. In contrast, deciding whether two records are duplicates usually requires fuzzy matching, for which humans sometimes can achieve better accuracy [114, 59, 116, 112].

We discuss three examples techniques: holistic data cleaning [26] is an example technique for automatically detecting violations of ICs; CrowdER [114] uses a hybrid machine-human approach to improve data deduplication accuracy; and Corleone [59] is a data deduplication

[1]`http://goo.gl/f5eym`

TID	FN	LN	ROLE	ZIP	ST	SAL
t_1	Anne	Nash	E	85376	NY	110
t_2	Mark	White	M	90012	NY	80
t_3	Mark	Lee	E	85376	AZ	75

Table 2.4: Employee data records.

system that is completely crowdsourced, *i.e.*, no developers need to be involved.

Holistic Data Cleaning

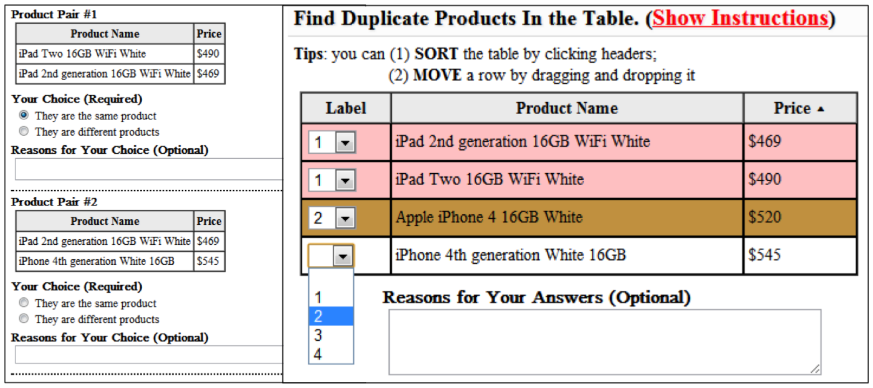
Holistic data cleaning [26] automatically detects violations of multiple ICs, and captures the cells that are more likely to be erroneous through the overlapping violations. Two violations are overlapping if they involve at least one common cell.

Example 2.16. Consider Table 2.4, every tuple specifies an employee in a company with her identification(TID), name (FN, LN), role (ROLE), ZIP (ZIP), state (ST), and salary (SAL).

Consider two data quality rules. The first is a functional dependency (FD) stating that ZIP determines ST. We can see that a set e_1 of four cells $\{t_1[ZIP], t_1[ST], t_3[ZIP], t_3[ST]\}$ in t_1 and t_3 present a violation for this FD: they have the same value for the city, but different states.

The second rule states that for two employees in the same state, the one whose role is manager cannot earn less than the one whose role is employee. In this case, a set e_2 of six cells $\{t_1[ROLE], t_1[ZIP], t_1[ST], t_2[ROLE], t_2[ZIP], t_2[ST]\}$ in t_1 and t_2 are violating the rule, since employee White, who is a manager, is earning less than Nash, who is an employee.

The two rules detect that $t_1[ST]$ is likely to be wrong since it participates in the two sets e_1 and e_2. Looking at e_1 individually, one cannot tell which one of four cells in e_1 is wrong; similarly, looking at e_2 alone is also not sufficient.


Product Pair #1

Product Name	Price
iPad Two 16GB WiFi White	$490
iPad 2nd generation 16GB WiFi White	$469

Your Choice (Required)

[✓ They are the same product] They are different products

Reasons for Your Choice (Optional)

Product Pair #2

Product Name	Price
iPad 2nd generation 16GB WiFi White	$469
iPhone 4th generation White 16GB	$545

Your Choice (Required)

They are the same product They are different products

Reasons for Your Choice (Optional)

Find Duplicate Products In the Table. (Show Instructions)

Tips: you can (1) **SORT** the table by clicking headers;
(2) **MOVE** a row by dragging and dropping it

Label	Product Name	Price ▲
1	iPad 2nd generation 16GB WiFi White	$469
1	iPad Two 16GB WiFi White	$490
2	Apple iPhone 4 16GB White	$520
	iPhone 4th generation White 16GB	$545

Reasons for Your Answers (Optional)

(a) Pair-based HIT **(b)** Cluster-based HIT

Figure 2.11: HITs for data deduplication

CrowdER

CrowdER [114] uses crowd workers to aid in the data deduplication task. The motivation for CrowdER is that while automatic techniques for data deduplication have been improving, the quality remains far from perfect; meanwhile, crowdsourcing platforms offer a more accurate, but expensive (and slow) way to bring human insight into the process. Crowdsourcing platforms, such as Amazon Mechanic Turk, support crowdsourced execution of "microtasks" or Human Intelligence Tasks (HITs), where people do simple jobs requiring little or no domain expertise, and are paid per job. Figure 2.11 shows two types for HITs used by CrowdER. The pair-based HIT in Figure 2.11a asks a human to check each pair of records individually; the cluster-based HIT in Figure 2.11b asks a human to cluster multiple records at the same time.

CrowdER proposes a human-machine workflow as shown in Figure 2.12. The workflow first uses machine-based techniques to compute, for each pair, the likelihood that they refer to the same entity. For example, the likelihood could be the similarity value given by a similarity-based technique. Then, only those pairs whose likelihood exceeds a specified threshold are sent to the crowd. It is shown that by specifying a relatively low threshold the number of pairs that need to

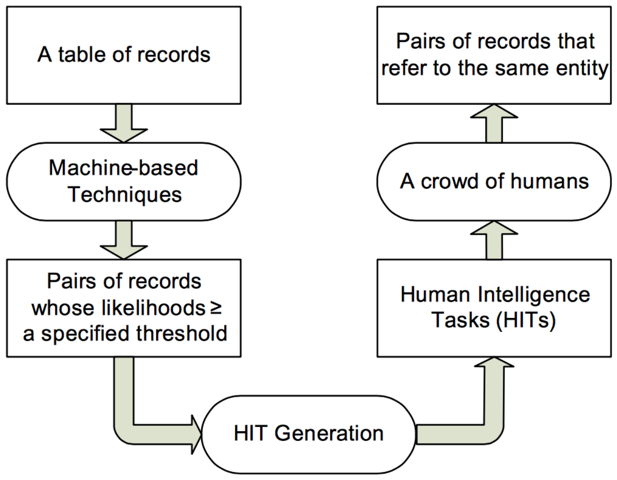

A table of records
Machine-based Techniques
Pairs of records whose likelihoods ≥ a specified threshold
HIT Generation
Human Intelligence Tasks (HITs)
A crowd of humans
Pairs of records that refer to the same entity

Figure 2.12: Hybrid human-machine workflow

be verified can be dramatically reduced with only a minor loss of quality. Given the set of pairs to be sent to the crowd, the next step is to generate HITs so that people can check them for matches. HIT Generation is a key component of the workflow. Finally, generated HITs are sent to the crowd for processing and the answers are collected.

Example 2.17. Figure 2.13 shows a workflow of CrowdER for deduplicating a table consisting of nine records $r_1, \ldots, r_9$ by using pair-based HIT. Instead of asking humans to check all pairs of records, that is $\frac{9*8}{2} = 36$ pairs, CrowdER first employs a machine based approach to calculate the similarities between pairs of records, such as the classifiers discussed in Section 2.1.2. Those record pairs whose similarities are lower than a threshold are pruned, such as (r_3, r_6). The remaining ten pairs can fit into five pair-based HITS, where each HIT contains two questions. The final matching pairs are collected based on user answers.

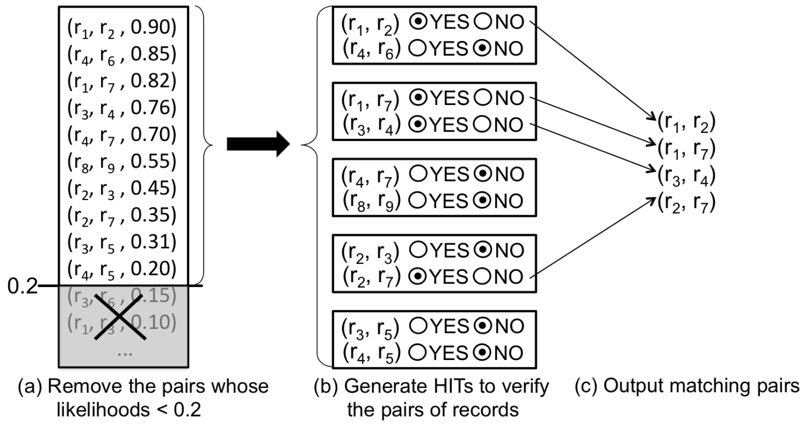


Figure 2.13: CrowdER: an example of using the hybrid human-machine workflow

Corleone

In contrast to CrowdER, which is a hybrid human-machine approach for data deduplication, Corleone [59] is a data deduplication system that is completely crowdsourced, *i.e.*, no developers need to be involved. Corleone is desirable because enterprises routinely need to solve tens to hundreds of data deduplication tasks. To involve a developer to write the blocking rules and matching function for every deduplication task is time consuming and costly. Figure 2.14 shows the Corleone architecture, which consists of four main components: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs' Locator. Blocking often uses heuristic rules, *e.g.*, "if the prices of two products differ by more than $100, then they do not match", to reduce the number of pairs of records to be matched. Corleone takes a small sample from all pairs of records and asks the crowd to label a small set of informative pairs to learn a random forest, from which potential blocking rules are extracted. The crowd is involved again to validate the quality of obtained blocking rules. After blocking, the next step is to build and apply a matcher to match surviving pairs of records. Corleone employs active learning to minimize crowdsourcing costs, taking into account possible noisy crowd input. The next step, *i.e.*, estimating the matching accuracy, is

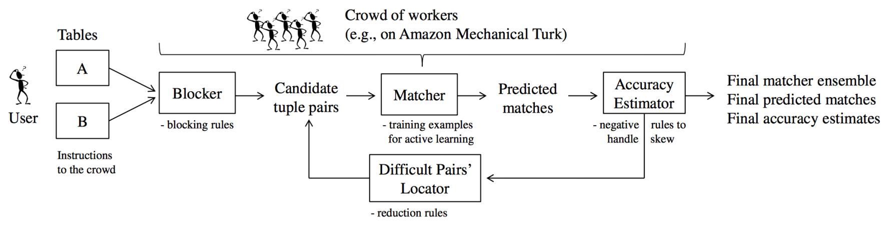


Figure 2.14: The Corleone architecture

vital for real world data deduplication tasks. To do this, Corleone considers constructing a minimal labeled set, given a maximum allowable error bound. The difficult pairs' locator finds pairs that the current matcher has matched incorrectly according to the accuracy estimator. The whole process is iterated until the estimated matching accuracy no longer improves.

2.3 Where to Detect

The problem of error detection is further complicated by the fact that errors are usually discovered much later in the data processing pipeline, where more business logics becomes available. Consider a simple example of two source tables, Employees and Departments. Detecting that the sum of employee salaries in a department exceeds the budget allocated for that department cannot be done before joining the two tables and aggregating the salaries of each group of employees.

In many applications, errors are detected in a target database (or a report) that is the result of data transformations applied on a source database. Figure 2.15 shows a typical data Extract-Transform-Load (ETL) processing stack. In each of the layers, various integrity constraints are defined as more semantics are added to the data. For example, while Constraint (4) (S1.NAME IS NOT NULL) can be defined directly on the sources, Constraints (1) and (2) can only be defined at the application and reporting layer after the necessary aggregation and joins have been performed.

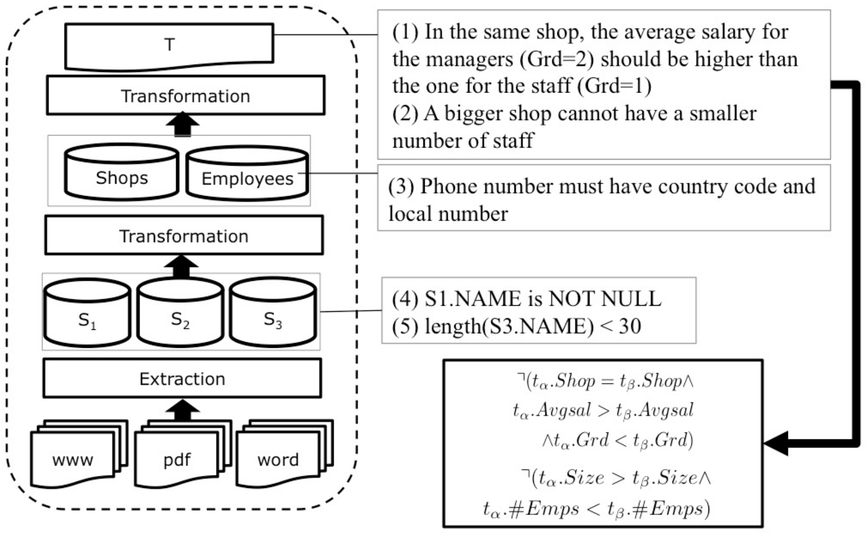

T
Transformation
Shops
Employees
Transformation
S1
S2
S3
Extraction
www
pdf
word
(1) In the same shop, the average salary for the managers (Grd=2) should be higher than the one for the staff (Grd=1)
(2) A bigger shop cannot have a smaller number of staff
(3) Phone number must have country code and local number
(4) S1.NAME is NOT NULL
(5) length(S3.NAME) < 30
¬(tα.Shop = tβ.Shop∧ tα.Avgsal > tβ.Avgsal ∧tα.Grd < tβ.Grd)
¬(tα.Size > tβ.Size∧ tα.#Emps < tβ.#Emps)

Figure 2.15: The ETL stack.

Propagating errors detected in transformation results to the data sources is essential for both repairing these errors and preventing them from reoccurring in the future. Techniques for error propagation vary according to the type of data transformations assumed, such as Boolean expressions [87], aggregation on numerical attributes [124, 123], and more general SPJA queries [19].

Causality Analysis

Causality analysis that tries to reason about the responsibility of a source in causing errors in query results is an intuitive way to handle the aforementioned problem. Work in this area [87] models the source database as a set of variables $\mathbf{X} = \{X_1, X_2, \ldots, X_n\}$ and the target database as another set of variables $\mathbf{Z} = \{Z_1, Z_2, \ldots, Z_m\}$. Each input variable X_i takes value from a discrete or continuous domain. Each output variable Z_j is a Boolean variable. The data transformation Φ_j for Z_j is a Boolean expression over threshold predicates in the form $X_i \ op \ c$, where $X_i \in \mathbf{X}$, $op \in \{<, \leq, =, \neq, \geq, >\}$, and c is a constant

$\boldsymbol{X}$	$\boldsymbol{x}$
X_1	5
X_2	T
X_3	2

(a)

$\boldsymbol{Z}$	$\boldsymbol{\Phi}$	$\boldsymbol{z}$
Z_1	$(X_1 < 10) \wedge X_2$	T
Z_2	$(X_3 > 0) \wedge X_2$	T
Z_3	$(X_3 > 3)$	F

(b)

Figure 2.16: (a): Value assignment $\mathbf{x}$ for $\mathbf{X}$. (b): The transformations $\mathbf{\Phi}$ and values $\mathbf{z}$ for $\mathbf{Z}$.

value in the domain of X_i. For example, a simple transformation Φ_1 is $Z_1 = (X_1 > 10) \wedge (X_2 < 3)$. Let $\mathbf{\Phi} = \{\Phi_1, \ldots, \Phi_m\}$ denote the m transformations for m output variables. See that $\mathbf{\Phi}$ takes input vector $\mathbf{x}$ of values for $\mathbf{X}$, and computes the output vector $\mathbf{z}$ of values for $\mathbf{Z}$. Let $\hat{\mathbf{z}}$ be the ground truth values for $\mathbf{Z}$. Given $\mathbf{X}, \mathbf{Z}, \mathbf{\Phi}, \mathbf{x}, \mathbf{z}, \hat{\mathbf{z}}$, the sources of errors are detected by ranking the input variables $\mathbf{X}$ according to how much each variable contributes to the error in $\mathbf{z}$, also referred to as the *responsibility of* X_i.

A *view-conditioned counterfactual cause* (VCC cause) of $\mathbf{z}|\hat{\mathbf{z}}$ is a minimal subset of input variables for which there exists a changed assignment that can change the output from $\mathbf{z}$ to $\hat{\mathbf{z}}$. One variable X_i is a *view-conditioned cause* (VC cause) of $\mathbf{z}|\hat{\mathbf{z}}$ if there exists a set $\Gamma \subset X$, called the *contingency set* of X_i, such that $X_i \cup \Gamma$ is a VCC cause of $\mathbf{z}|\hat{\mathbf{z}}$. The responsibility of X_i is defined as $\rho_{X_i} = \frac{1}{1+\min_\Gamma |\Gamma|}$, where Γ is a contingency for X_i. Intuitively, the responsibility of X_i is the minimal number of input variables that need to be changed together with X_i in order to change the output from $\mathbf{z}$ to $\hat{\mathbf{z}}$.

Example 2.18. Consider $\mathbf{\Phi}$ given in Figure 2.16. Assume the ground truth is $\hat{\mathbf{z}} = \{F, T, T\}$, which indicates that Z_1 and Z_3 are errors in the output. $\{X_1, X_3\}$ is a VCC cause of $\mathbf{z}|\hat{\mathbf{z}}$, since changing X_1 from 5 to 11 and X_3 from 2 to 4 will change the output values from $\mathbf{z}$ to $\hat{\mathbf{z}}$ and changing only X_1 or X_3 will not flip both Z_1 and Z_3. The minimal sized contingency sets for X_1, X_2, and X_3 are $\{X_3\}, \{X_1, X_3\}$, and $\{X_1\}$, respectively. Therefore, the responsibilities of X_1, X_2, and X_3 are $\rho_{X_1} = \frac{1}{2}, \rho_{X_2} = \frac{1}{3}$ and $\rho_{X_3} = \frac{1}{2}$, which show that X_1 and X_3 contribute more than X_2 to the errors in the output variables.

Since computing the causality and responsibility are NP-hard [87], the problem of computing causality is reduced to the SAT problem, and the problem of computing responsibility is reduced to a partial weighted MaxSAT problem. There exist several highly optimized tools to solve both SAT and weighted MaxSAT, which allow for efficient execution.

Scorpion

Another approach for computing the responsibility of sources in target errors is the Scorpion system [123, 124]. Scorpion assumes a single relational table as the source database and an SQL aggregate query as the data transformation. The target database is then a group of aggregate values, one for each group according to the aggregate query. The errors in the target database are those aggregate values that are considered outliers by users. Scorpion finds common properties of the set of tuples in the source database that cause the outliers in the target database. The common properties of a set of tuples are described by *predicates* over the attributes of the source database.

Scorpion uses *sensitivity analysis* to identify predicates that are most influential over the aggregate values. For example, given a function, $y = f(x_1, \ldots, x_n)$, the influence of x_i is defined by the partial derivative, $\frac{\Delta y}{\Delta x_i}$. Similarly, the *influence* of a predicate p on one group α, denoted as $inf(p, \alpha)$, is defined as the ratio between the change in the output if the tuples satisfying the predicates are deleted from the input and the number of tuples satisfying the predicate. Thus, the influence of predicate p, denoted as $inf(p)$, is the average influence of p on all groups.

Example 2.19. Figure 2.17a shows some readings from an Intel Sensor Dataset, with each row corresponding to readings from a certain sensor at a given time. Figure 2.17b is generated by the following aggregate query, which groups the readings by the hour and computes the mean temperature.

```
Q: SELECT AVE(temp), time
   FROM sensors
   GROUP BY time
```

TIDs	Time	SensorID	Voltage	Humidity	Temp
t_1	11AM	1	2.64	0.4	34
t_2	11AM	2	2.65	0.5	35
t_3	11AM	3	2.65	0.4	35
t_4	12AM	1	2.7	0.3	35
t_5	12AM	2	2.7	0.5	35
t_6	12AM	3	2.3	0.4	100
t_7	1PM	1	2.7	0.3	35
t_8	1PM	2	2.7	0.5	35
t_9	1PM	3	2.3	0.5	80

(a) Example reading from sensors

ResultIDs	Time	AVG(temp)	Label
α_1	11AM	34.6	Normal
α_2	12AM	56.6	Outlier
α_3	1PM	50	Outlier

(b) Query results (left three columns) and user annotations (right column)

The rightmost column in Figure 2.17b represents user annotations. The user thinks two groups α_2 and α_3 have unusual results and group α_1 is a normal result.

Consider a predicate $p : Voltage < 2.4$. The average temperature of group α_2 after deleting tuples satisfying the predicate that is t_6 is 35. Thus $inf(p, \alpha_2) = \frac{56.6-35}{1} = 21.6$. Similarly, $inf(p, \alpha_3) = \frac{50-35}{1} = 15$. Therefore, $inf(p) = 18.3$. Scorpion searches all possible predicates over attributes that are not involved in the query, *e.g.*, Voltage and Humidity in this example, and returns the predicate with the largest influence.

To efficiently compute the influence of a predicate and to avoid testing the exponential number of all possible predicates, Scorpion identifies several properties of aggregate operators, *i.e.*, incrementally removable, independent, and anti-monotonic influence, that enables the algorithms to find the most influential predicate efficiently.

DBRx

To handle a more general class of transformations, the DBRx system [19] considers SPJA (select, project, join, and aggregate) queries as the reporting and transformation language. Figure 2.18 shows the

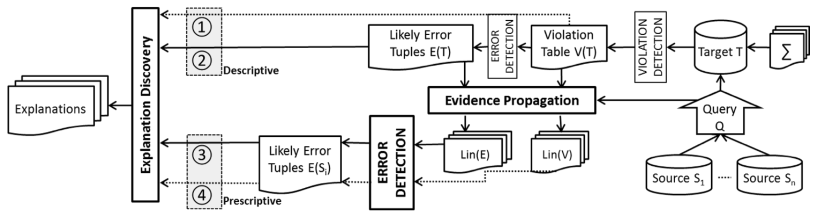


Figure 2.18: DBRx architecture.

T	Shop	Size	Grd	AvgSal	#Emps	Region
t_a	**NY1**	46 ft^2	**2**	**99** \$	1	US
t_b	**NY1**	*46* ft^2	***1***	**100** \$	*3*	US
t_c	NY2	62 ft^2	2	96 \$	2	US
t_d	NY2	*62* ft^2	*1*	90 \$	*2*	US
t_e	LA1	35 ft^2	2	105 \$	2	US
t_f	LND	38 ft^2	1	65 £	2	EU

(a) Target database T (Dirty)

Emps	EId	Name	Dept	Sal	Grd	SId	JoinYr
t_1	e4	John	S	91	1	NY1	2012
t_2	e5	Anne	D	99	2	NY1	2012
t_3	e7	Mark	S	93	1	NY1	2012
t_4	e8	Claire	S	116	**1**	NY1	2012
t_5	e11	Ian	R	89	1	NY2	2012
t_6	e13	Laure	R	94	2	NY2	2012
t_7	e14	Mary	E	91	1	NY2	2012
t_8	e18	Bill	D	98	2	NY2	2012
t_9	e14	Mike	R	94	2	LA1	2011
t_{10}	e18	Claire	E	116	2	LA1	2011

(b) Source Relation *Emps*

Shops	SId	City	State	Size	Started
t_{11}	NY1	New York	NY	46	2011
t_{12}	NY2	New York	NY	62	2012
t_{13}	LA1	Los Angeles	CA	35	2011

(c) Source Relation *Shops*

Figure 2.19: A view T on data sources *Emps* & *Shops*.

Possible Explanations

Explanation	Coverage	Preciseness	Conciseness
$Dept = s$ **?**	No	Yes	Yes
$eid = e_4 \lor eid = e_7 \lor$ $eid = e_8 \lor eid = e_{14}$ **?**	Yes	Yes	No
$Grd = 1$ **?**	Yes	No	Yes

Figure 2.20: Explanation discovery.

architecture of DBRx that takes quality rules defined over the output of a transformation and computes explanations of the errors. Given a transformation scenario (sources S_i, $1 < i < n$, and query Q) and a set of quality rules Σ, DBRx computes a violation table VT of tuples not complying with Σ. VT is mined to discover a descriptive explanation of the violations ①. The description explanation should cover the most likely erroneous tuples, while minimizing the clean tuples being covered. The lineage of the violation table over the sources enables the computation of a prescriptive explanation on the source tables ④. When applicable, a repair is computed over the target, thus allowing the possibility of a more precise description ②, and a more precise prescriptive explanation ③, based on propagating errors to the sources.

Example 2.20. Consider a target database T in Figure 2.19 which lists shops in an international franchise and information about employees working in those shops. T is generated by the following query:

```
Q: SELECT SId as Shop, Size, Grd, AVG(Sal) as
   AvgSal, COUNT(EId) as #Emps,‘US’ as Region
   FROM US.Emps JOIN US.Shops ON SId
   GROUP BY SId, Size, Grd
```

The HR department has a set of policies (ICs) pertaining to the franchise workforce and these ICs are enforced on T. The first rule states that, in the same shop, the average salary of the managers (GRD=2) should be higher than that of the staff (GRD=1). Cells $Shop, Grd, AvgSal$ of tuples t_a and t_b, labeled in bold, violate this rule. The second rule states that a bigger shop cannot have a smaller staff (GRD=1), which induces a violation between cells of t_b and t_d, labeled in italic.

Tuples t_a and t_b are in violation and their lineage is $\{t_1 - t_4\}$ and $\{t_{11}\}$ in sources *Emps* and *Shops*, respectively. Similarly, another violation between t_b and t_d has the combined lineage $\{t_5 - t_8\}$ and $\{t_{12}\}$. For each cell and tuple in the lineage, the cell contribution score (CSV) and tuple removal score (RSV) is computed. The CSV (resp. RSV) is a m-length vector to represent the contribution (resp. removal) scores of a cell (resp. tuple), where m is the number of violations in T. For instance, the RSV of t_4 is [1,1] since the removal of t_4 would resolve both violations.

m Based on the CSV and RSV, DBRx identifies t_1, t_3, t_4, t_7 to be the most likely erroneous tuples. To explain these tuples, DBRx summarizes the tuples in terms of source attribute predicates with three desirable properties, namely, *coverage*, *preciseness*, and *conciseness*. Coverage requires an explanation to cover erroneous tuples, preciseness requires an explanation to cover mostly erroneous tuples, not correct tuples, and conciseness requires an explanation to have a small number of predicates. Figure 2.20 lists three different explanations. The first one lacks coverage, since t_7 is not covered; the second one lacks conciseness, since it required four predicates to describe the four tuples; and the third one lacks preciseness, since it also incorrectly covers the correct tuple t_5. These explanations can then be inspected by users, and similar errors can be prevented from happening in the future.

3

Taxonomy of Data Repairing Techniques

Given a relational database instance I of schema R and a set of data quality requirements expressed in a variety of ways, data repairing refers to the process of finding another database instance I' that conforms to the set of data quality requirements. We assume that the data quality requirements are enforcement of ICs and duplicate detection in this paper. A plethora of data repairing techniques have been proposed. Figure 3.1 depicts the classification we adopt to categorize the proposed data repairing techniques. In the following, we discuss our classification dimensions, and their impact on the design of underlying data repairing techniques. For each dimension, we give one or more examples to discuss in detail. The three adopted dimensions capture the three main questions involved in repairing an erroneous databases:

- *Repair Target (What to Repair?)* Repairing algorithms make different assumptions about the data and the quality rules: (1) trusting the declared integrity constraints, and hence, only data can be updated to remove errors; (2) trusting the data completely and allowing the relaxation of the constraints, for example, to address schema evolution and obsolete business rules; and finally (3) exploring the possibility of changing both the data and the

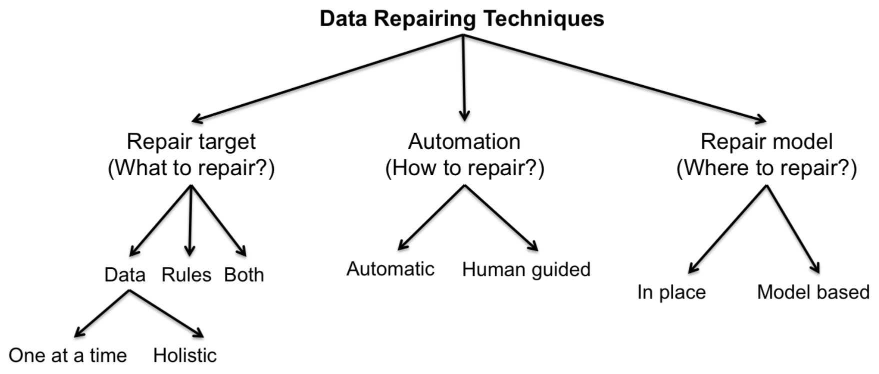


Figure 3.1: Classification of data repairing techniques.

constraints. For techniques that trust the rules, and change only the data, they can be further divided according to the driver to the repairing exercise, that is, what types of errors they are targeting. A majority of techniques repair the data with respect to one type of errors only (one at a time), while other emerging techniques consider the interactions among multiple types of errors and provide a holistic repair of the data (holistic).

- *Automation (How to Repair?)* We classify proposed approaches with respect to the tools used in the repairing process. More specifically, we classify current repairing approaches according to whether and how humans are involved. Some techniques are fully automatic, for example, by modifying the database, such that the distance between the original database I and the modified database I' is minimized according to some cost function. Other techniques involve humans in the repairing process either to verify the fixes, to suggest fixes, or to train machine learning models to carry out automatic repairing decisions.

- *Repair Model (Where to Repair?)* We classify proposed approaches based on whether they change the database in-situ, or build a model to describe the repair. Most proposed techniques repair the database in place, thus destructing the original database.

	Repair target What				**Automation How**		**Repair model Where**	
	Data - One at a time	Data - Holistic	Rules	Both	Automatic	Human involved	In place	Model based
FDs value modification [17]	✓				✓		✓	
FDs hypergraph [77]	✓				✓		✓	
CFDs value modification [31]	✓				✓		✓	
Holistic data cleaning [26]		✓			✓		✓	
LLUNATIC [55]		✓			✓		✓	
Record matching and data repairing [51]		✓			✓		✓	
NADEEF [32]		✓			✓		✓	
Generate optimal tablaux [60]			✓		✓		✓	
Unified repair [23]				✓	✓		✓	
Relative trust [10]				✓	✓		✓	
Continuous data cleaning [113]				✓	✓		✓	
AJAX [53, 54]	✓					✓	✓	
Potter's Wheel [94]	✓					✓	✓	
GDR [126]	✓					✓	✓	
KATARA [28]	✓					✓	✓	
DataTamer [105]	✓					✓	✓	
Editing rules [50]	✓					✓	✓	
Sampling FDs repairs [9]	✓				✓			✓
Sampling CFDs repairs [11]	✓				✓			✓
Sampling Duplicates [12]	✓				✓			✓

Table 3.1: A sample of data repairing techniques.

For none in-situ repairs, a model is often built to describe the different ways to repair the underlying database. Queries are answered against these repairing models using, for example, sampling from all possible repairs and other probabilistic query answering mechanisms.

Table 3.1 shows a sample of data repairing techniques using the taxonomy.

3.1 What to Repair

Business logic is not static; it often evolves over time. Previously correct integrity constraints may become obsolete quickly. Practical data repairing techniques must consider possible errors in the data as well as possible errors in the specified constraints. Thus, the repair targets include data only, rules only, and a combination of both. In data only repairing, data is modified to conform to a set of ICs; in rules only repairing, rules are modified, such that they hold on the data; and in data and rules repairing, data and rules are simultaneously modified, such that the modified data conforms to the modified rules.

3.1.1 Data Only Repairing

Data repairing techniques in this category assume there is a set of ICs Σ defined on the database schema R, and any database instance I of R should conform to these constraints.

Multiple proposed approaches aim at changing a minimum number of cells in the database, such that a set of FDs are satisfied, either by changing the data directly [17, 77] or by generating samples from the space of all possible minimal repairs [9]. We give multiple definitions of minimal repairs in Section 3.2 when we discuss automatic data repairing techniques. Other approaches that generate minimal repairs were also proposed to address other types of ICs, such as CFDs [31], and the more general Denial Constraints (DCs) [26]. Approaches that avoid the minimal repair heuristic generate verified fixes, for example, by using reference master data and editing rules [50], or by heavily involving human experts [126, 105].

In this section, we will discuss one data repairing algorithm with respect to a given set of FDs, and we will also discuss different record fusion techniques for dealing with duplicate records.

Algorithm 4 gives the details of a data repairing algorithm with respect to a given set of FDs [17] as an example. The details of other examples will be described in other sections when we discuss other dimensions and design options. Algorithm 4 takes as input a set of FDs, and a dirty database instance I, and produces a repaired database instance I'. At a high level, Algorithm 4 initially puts every database cell in its own equivalence class. An equivalence class is a set of cells that should have the same value. Then it greedily merges the equivalence classes in $\mathcal{E}$ until all FDs in Σ are satisfied. The unResolved sets keeps tracks of all tuples that participate in violations of FDs in Σ. At each step, a tuple t and an FD $X \rightarrow A$ is picked, such that the cost of merging all equivalence classes that should have the same value according to the FD is the lowest. The termination of Algorithm 4 is based on the fact that the number of equivalence class merges is bounded by the total number of equivalence classes.

Algorithm 4 `GenFDsRepair`

```
Input: Database instance $I$, a set of FDs $\Sigma$
Output: Another instance $I'$, such that $I' \models \Sigma$
1: $\mathcal{E} \leftarrow \{t[A] : t \in I, A \in R\}$
2: Initialize unResolved sets for $\Sigma$
3: while unResolved is not empty do
4:     pick the next tuple $t$, and next FD $X \rightarrow A$ to repair with the
       lowest repair cost
5:     resolve tuple $t$ and update $\mathcal{E}$
6:     update unResolved sets affected by $\mathcal{E}$
7: Return $I'$ obtained by picking a value that results in the lowest
   cost of $cost(eq)$ for each $eq \in \mathcal{E}$
```

Similar to other data repairing algorithms, the key difficulty addressed in Algorithm 4 is that repairing one constraint might introduce violations for other constraints. To capture and track the effect of changing one database cell on the possible values other cells can take, equivalence classes of database cells are used. An equivalence class eq is a set of database cells (e.g., $\{t_1[A], t_2[A], t_3[A]\}$, where t_i is a tuple identifier and A is an attribute in R) that should have the same value. The algorithm maintains a global set of equivalence classes $\mathcal{E}$. For a given cell $t_i[A]$, let $eq(t_i[A])$ denote the current equivalence class containing $t_i[A]$ in $\mathcal{E}$. The cost of updating an equivalence class with respect to a value v, $cost(eq, v)$, is the cost of changing the values of all cells in eq to v. The cost of an equivalence class $cost(eq)$ is the minimal cost for all possible values v. The cost of merging a set E of equivalence classes is $mgcost(E) = cost(\cup_{eq \in E} eq) - \sum_{eq \in E} cost(eq)$.

Example 3.1. Consider an equivalence class $eq_1 = \{t_1[A], t_2[A], t_3[A]\}$, and assume $t_1[A], t_2[A], t_3[A]$ have values a_1, a_2, and a_2, respectively. Assuming that the cost of changing any cell is 1, we have $cost(eq_1, a_1) = 2$ and $cost(eq_1, a_2) = 1$, and thus $cost(eq_1) = 1$. Consider another equivalence class $eq_2 = \{t_4[A]\}$ with $t_4[A] = a_3$. The cost of merging eq_1 with eq_2 to form $eq_3 = \{t_1[A], t_2[A], t_3[A], t_4[A]\}$ is given by $mgcost(\{eq_1, eq_2\}) = cost(eq_3) - (cost(eq_1) + cost(eq_2)) = 2 - (1+0) = 1$.

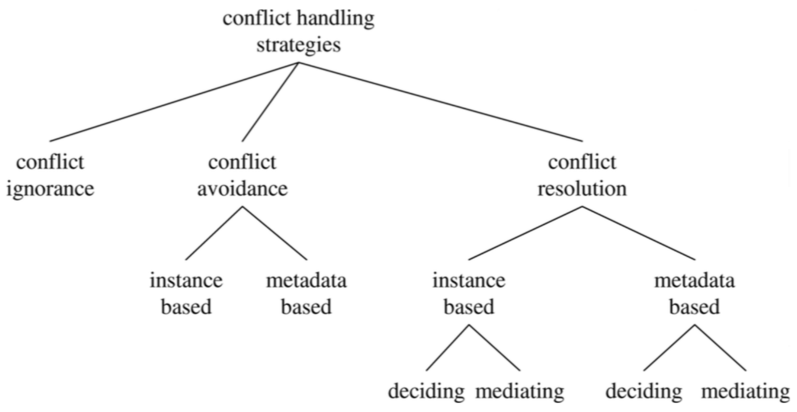


Figure 3.2: Classification of strategies to fuse records [16].

The cost of updating and merging equivalence classes become the main building blocks in finding a minimal-cost repair, while capturing the dependency among cells when assigning a new value. Algorithm 4 was later extended to handle violations of CFDs based on the concept of equivalence classes [31].

Record fusion refers to the process of consolidating multiple records representing the same real world entity into a single representation. Figure 3.2 shows the classification of different record fusing strategies [16]. *Conflict ignorance* strategies ignore the conflicts between multiple records that refer to the same entity, and pass the conflicts to the users or applications. Probabilistic deduplication discussed in Section 3.3 is an example of conflict ignorance strategies. *Conflict avoidance* strategies acknowledge the existence of conflicting records, and apply a simple rule to take a unique decision based on either the data instance or the metadata. An example of instance based conflict avoidance strategy is to prefer non-null values over null values. An example of metadata based conflict avoidance strategy is to prefer values from one relation over values from another. *Conflict resolution* strategies resolve the conflicts, by picking a value from the already present values (deciding) or by choosing a value that does not necessarily exist among present values (mediating). An example of instance based, deciding conflict resolution strategy is to take the most frequent value. An example of instance based, mediating conflict resolution strategy is to take the average of all present values. For instance, to resolve the SAL attribute of duplicate tuples t_4 and t_9 in Example 1.1, the average value of t_4[SAL] and t_9[SAL] can be taken. More advanced conflict resolution strategies [48, 47] consider the interactions of multiple types of constraints to infer the correct value. For example, a currency constraint, specifying that a person's working status can change from working to retired, and not from retired to working, and a constant CFD, specifying that the salary of a retired person is zero, can be used together to determine a person's correct working status.

Holistic Repairing

Data only repairing techniques make different assumptions about the driver of the repairing process. In Section 2, we discussed different error types that trigger data anomaly detections. For each type of error, such as duplicate records, missing values, and FD violations, multiple repairing algorithms are proposed. For example, the entity resolution algorithms mentioned in Section 2.1.2 are examples of techniques that target duplicates. We discussed one technique in Section 3.1 addressing violations of FD constraints. Similarly, violations of CFD constraints have been addressed in multiple proposals, either automatically [31] or by involving humans in the loop [126]. We describe these techniques as *One at a time* techniques. Most available data repairing solutions are in this category. They address one type of error, either to allow for theoretical quality guarantees, or to allow for a scalable system.

However, data anomalies are rarely due to a single type of errors; multiple data quality problems, such as missing values, typos, the presence of duplicate records, and business rule violations, are often observed in a single data set. These heterogeneous types of errors interplay and conflict on the same dataset, and treating them independently would miss the opportunity to correctly identify the actual errors in the data. We call the proposals that take a more holistic view of the data cleaning process *Holistic* cleaning approaches [26, 32, 55, 51, 48]

Holistic repairing algorithms consider violations coming from different types of ICs at the same time, while suggesting updates to repair the underlying data. For example, Chu et al. [26] consider a wide range of ICs including, FDs, CFDs, and DCs, as long as the violations of ICs can be encoded as a hyperedge in the conflict hypergraph; NADEEF [32], an open source data cleaning system which provides an interface for the users to define their own data quality rules, also uses techniques from [26] to holistically resolve violations; Geerts et al. [55] consider all constraints that can be expressed as equality-generating dependencies; Fan et al. [51] integrate data repairing based on CFDs and record matching based on MDs, and show that these two tasks benefit from each other when coupled together.

We give the details of a holistic data cleaning algorithm [26] as an example technique in this category and we briefly mention the main insights of other proposals [55], [51].

The holistic data cleaning algorithm is shown in Algorithm 5. The system takes as input a relational database I and a set of ICs Σ, which express the data quality rules that have to be enforced over the input database. It first projects the violations coming from different types of ICs into one homogeneous representation, *i.e.*, a conflict hypergraph. Each node in the conflict hypergraph is a cell in the database; each edge is a set of cells participating in a violation of an IC. A minimum vertex cover for the conflict hypergraph is found. The minimum vertex cover contains the cells that are mostly likely to be wrong, *i.e.*, those participating in multiple violations of different ICs. Anchoring on the cells in the minimum vertex cover, a set of repair requirements are collected, such that if they are satisfied, all the violations will be resolved.

```
Algorithm 5 Holistic data cleaning
Input: Database instance $I$, a set of ICs $\Sigma$
Output: Repaired database instance $I'$
1: build the conflict hypergraph for $I$ w.r.t. $\Sigma$
2: find the minimum vertex cover for the conflict hypergraph
3: using a recursive procedure to recursively collect repair requirements, starting from the minimum vertex cover
4: using a determination procedure to satisfy the repair requirement according to a cost function
5: update the database $I'$, and build the CH again.
6: if $CH$ is not null then
7:     Go to Line 1
8: else
9:     return
```

The set of repair requirements is fed into a determination procedure, which computes a set of cell updates to the database, such that all repair requirements are satisfied. Depending on the repair requirements, different determination procedures can be devised to suit different cost functions (cf. Section 3.2). For example, if there are $>, \geq, <, \leq$ operators in the repair requirements, a quadratic or linear programming solver may be used as the determination procedure. The database is then updated accordingly. The process is repeated until there are no violations of any ICs.

Example 3.2. Figure 3.3 shows the conflict hypergraph built for the two violations e_1 and e_2 in Example 2.16. A minimum vertex cover of this graph is $\{t_1[ST]\}$. Anchoring on $t_1[ST]$, we collect repair requirements for fixing e_1 and e_2. To fix e_1 by changing $t_1[ST]$, $t_1[ST]$ has to be changed to not equal to $t_3[ST]$. To fix e_2 by changing $t_1[ST]$, $t_1[ST]$ has to be changed to not equal to $t_2[ST]$. Thus, the two repair requirements are: $t_1[ST] = t_3[ST]$ and $t_1[ST] \neq t_2[ST]$. Given these two repair requirements, and suppose the cost function is to change the minimum number of cells, a determination procedure is invoked to change the minimum number of cells out of these three cells $t_1[ST], t_2[ST], t_3[ST]$,

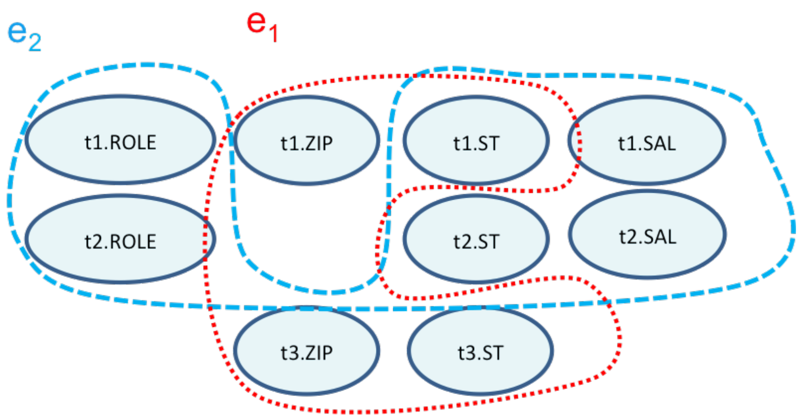


Figure 3.3: Conflict Hypergraph.

such that two requirements are satisfied. In this case, changing only one cell, *i.e.*, updating $t_1[ST]$ from "NY" to "AZ" would satisfy the two requirements.

As stated before, different determination procedures can be used for different cost functions.

Example 3.3. Consider an instance of I of a relational schema $R(A, B, C)$, where I has only one tuple t_1 with $t_1[A] = 0$, $t_1[B] = 3$ and $t_1[C] = 2$. Suppose there are two repair requirements: $t_1[A] < t_1[B]$ and $t_1[B] < t_1[C]$.

If one would like to minimize the squared distance between a repaired instance I' and I, a quadratic programming problem can be used with the objective function $(t_1[A] - 0)^2 + (t_1[B] - 3)^2 + (t_1[C] - 2)^2$ and the two repair requirements as two constraints. The optimal solution to the quadratic is $t_1[A] = 1$, $t_1[B] = 2$, and $t_1[C] = 3$.

However, if one would like to minimize the number of changed cells, only one cell needs to be changed (change $t_1[B]$ to 1) to satisfy the two repair requirements.

LLUNATIC [55] is a data cleaning framework that considers different kinds of integrity constraints, as well as different strategies, to repair conflicting values. Exactly, it specifies constraints based on equality-generating dependencies (egds), which can express FDs, CFDs, MDs, and eRs. To specify different repair strategies, it introduces the notion of *cell group*, which is a set of cells that should take on the same value along with the lineage of the value to take, *e.g.*, coming from a master relation. A *cell group* is similar to the notion of an *equivalence class*, as discussed earlier in Section 3.1. A partial order is introduced to cell groups; the partial order specifies the typical strategies to select a value for a cell group, including master data, certainty, accuracy, freshness, and currency, as well as user specified preferences. A parallel chase engine is developed to compute the repair. The chase procedure includes a cost manager, which decides which repair to retain or discard based on different repairing objectives, *e.g.*, cost minimal or cardinality minimal. Example 3.4 shows several cases where one value is preferred to another in resolving a violation. LLUNATIC is later extended to include user input into the chase procedure by allowing users to resolve conflicting values for which there is no clear preference, or by allowing users to discard unwanted repairs [56, 57]

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

Figure 3.4: Customers and Treatments.

Example 3.4. Consider the database shown in Figure 3.4, containing customer data (CUSTOMERS) with addresses and credit card numbers of customers, and medical treatments paid by insurance plans (TREATMENTS). The following constraints are defined on the database: an FD φ_1 : SSN, NAME $\rightarrow$ PHONE defined on CUS-

TOMERS, an FD φ_2 : SSN, NAME $\rightarrow$ CC# defined on CUSTOMERS, and an FD φ_3 : SSN $\rightarrow$ SALARY defined on TREATMENTS.

Tuples t_2 and t_3 are violating φ_1 and one may want to equate t_2[PHONE] and t_3[PHONE] to fix the violation. However, φ_1 does not tell which value ("122-1876" or "000-0000") t_2[PHONE] and t_3[PHONE] should take. If the PHONE attribute of CUSTOMERS comes with a confidence CONF, shown in CUSTOMERS in Figure 3.4, and the value with higher confidence is preferred, the violation is repaired by changing t_3[PHONE] to "122-1876".

Tuples t_4 and t_5 are violating φ_3 and if the more recent value for SALARY attribute of a person is preferred, the violation of can be repaired by changing t_4[SALARY] to the value of t_5[SALARY], which is more recent than the value of t_4[SALARY], according to Attribute DATE in TREATMENTS.

It is not always clear how to choose preferred values. For example, when repairing t_2[CC#] and t_3[CC#] for φ_2, there is no information available to resolve the conflict. The best one can do is to mark the conflict, and then, perhaps, ask for user-interaction to solve it.

Fan et al. [51] integrate data repairing based on CFDs and record matching based on MDs (cf. Section 2.1.1), and show that these two tasks benefit when considered together. For a relation I of schema R, a master relation I_m of schema R_m, a set Σ of CFDs defined on R, and a set Γ of MDs defined on R and R_m, a repair I' of I is another instance of R, such that (1) I' satisfies Σ; (2) I' and I_m satisfy Γ, and (3) $cost(I, I')$ (cf. Section 3.2) is minimal. Example 3.5 gives a scenario where these two tasks interplay and benefit from each other.

Example 3.5. Consider two relational tables card and tran defined in Example 2.8, that is: card(FN, LN, St, city, AC, zip, tel, dob, gd), and tran(FN, LN, St, city, AC, post, phn, gd, item, when, where). Table 3.2 shows an instance D_m of card, and an instance D of tran. The following constraints are defined on tran and card: a CFD φ_1 tran([AC = 020] $\rightarrow$ [city = Ldn]), an FD φ_2 tran([city, phn] $\rightarrow$ [St, AC, post]), a CFD φ_3 tran([FN = Bob] $\rightarrow$ [FN = Robert]), and an MD ψ

$\mathsf{tran}[\mathsf{LN}, \mathsf{city}, \mathsf{St}, \mathsf{post}] = \mathsf{card}[\mathsf{LN}, \mathsf{city}, \mathsf{St}, \mathsf{zip}] \wedge \mathsf{tran}[\mathsf{FN}] \approx \mathsf{card}[\mathsf{FN}] \rightarrow \mathsf{tran}[\mathsf{FN}, \mathsf{phn}] \rightleftharpoons \mathsf{card}[\mathsf{FN}, \mathsf{tel}]$.

Consider Tuples t_3 and t_4 in D. The bank suspects that the two records refer to the same person. If so, then these transaction records show that the same person made purchases in the U.K. and in the U.S. at about the same time (taking into account the 5-hour time difference between the two countries). This indicates that a fraud has likely been committed.

Tuples t_3 and t_4 are quite different in their FN, city, St, post and Phn attributes. No rules can identify the two directly. Nonetheless, they can indeed be matched by a sequence of *interleaved* matching and repairing operations: (a) get a repair t'_3 of t_3 such that $t'_3[\mathsf{city}] = \text{Ldn}$ via CFD φ_1, and $t'_3[\mathsf{FN}]{=}\text{Robert}$ by normalization with φ_3; (b) match t'_3 with s_2 of D_m, to which ψ can be applied; (c) as a result of the matching operation, get a repair t''_3 of t_3 by correcting $t''_3[\mathsf{phn}]$ with the master data $s_2[\mathsf{tel}]$; and (d) find a repair t'_4 of t_4 via the FD φ_2: since t''_3 and t_4 agree on their city and phn attributes, φ_2 can be applied to enrich $t_4[\mathsf{St}]$ and fix $t_4[\mathsf{post}]$ by taking corresponding values from t''_3, which have been confirmed correct with the master data in Step (c).

At this point t''_3 and t'_4 agree on every attribute. It is now evident enough that they indeed refer to the same person; hence, a fraud. Observe that not only repairing helps matching, for example, from Step (a) to (b), but matching also helps to repair the data; for example, Step (d) can be done only after the matching in (b).

3.1.2 Rules Only Repairing

The techniques in this category assume data is clean, and ICs need to be changed, such that data conforms to the changed ICs. A particular example is the pattern tableaux problem for CFDs (cf. Section 2.1.1.2), where an embedded FD is given [60]. We give the details of this algorithm as an example approach to change ICs.

A good pattern tableaux should maximize the number tuples matching the pattern tuples in the tableaux while minimizing the number of violations. Furthermore, a tableaux should be concise to capture the semantics of the data in a readable way. For example, if every tuple

	FN	LN	St	city	AC	zip	tel	dob	gd
s_1:	Mark	Smith	10 Oak St	Edi	131	EH8 9LE	3256778	10/10/1987	Male
s_2:	Robert	Brady	5 Wren St	Ldn	020	WC1H 9SE	3887644	12/08/1975	Male

(a) Master data D_m: An instance of schema card

	FN	LN	St	city	AC	post	phn	gd	item	when	where
t_1:	M.	Smith	10 Oak St	Ldn	131	EH8 9LE	9999999	Male	watch, 350 GBP	11am 28/08/2010	UK
cf	(0.9)	(1.0)	(0.9)	(0.5)	(0.9)	(0.9)	(0.0)	(0.8)	(1.0)	(1.0)	(1.0)
t_2:	Max	Smith	Po Box 25	Edi	131	EH8 9AB	3256778	Male	DVD, 800 INR	8pm 28/09/2010	India
cf	(0.7)	(1.0)	(0.5)	(0.9)	(0.7)	(0.6)	(0.8)	(0.8)	(1.0)	(1.0)	(1.0)
t_3:	Bob	Brady	5 Wren St	Edi	020	WC1H 9SE	3887834	Male	iPhone, 599 GBP	6pm 06/11/2009	UK
cf	(0.6)	(1.0)	(0.9)	(0.2)	(0.9)	(0.8)	(0.9)	(0.8)	(1.0)	(1.0)	(1.0)
t_4:	Robert	Brady	null	Ldn	020	WC1E 7HX	3887644	Male	necklace, 2,100 USD	1pm 06/11/2009	USA
cf	(0.7)	(1.0)	(0.0)	(0.5)	(0.7)	(0.3)	(0.7)	(0.8)	(1.0)	(1.0)	(1.0)

(b) Database D: An instance of schema tran

Table 3.2: Example credit card and transaction records

in the database is a pattern tuple in the pattern tableaux, then the pattern tableaux would match all the database tuples; however, the large tableaux size makes it almost unusable in practice.

Given a CFD $(R : X \rightarrow Y, T_p)$ and a database instance I of schema R, the cover of a pattern tuple $t_p \in T_p$ is defined as all the tuples in I that match t_p, *i.e.*, $Cover(t_p) = \{t | t \in I \text{ and } t \approx t_p\}$. The local support of t_p is thus defined as $LS(t_p) = \frac{|Cover(t_p)|}{|I|}$. Let $Keepers(t_p)$ be the subset of tuples in $Covers(t_p)$ after removing the fewest number of tuples to remove all violations of t_p. The local confidence of t_p is thus defined as $LC(t_p) = \frac{|Keepers(t_p)|}{|Cover(t_p)|}$. The global support of the pattern tableaux T_p is defined as the fraction of tuples in I matching any pattern tuple in T_p, *i.e.*, $GS(T_p) = \frac{|\cup_{t_p \in T_p} Covers(t_p)|}{|I|}$. The global confidence of the pattern tableaux T_p is defined as $GC(T_p) = \frac{|\cup_{t_p \in T_p} Keepers(t_p)|}{|\cup_{t_p \in T_p} Covers(t_p)|}$.

Example 3.6. Consider the CFD ({name, type, country} $\rightarrow$ {price, tax}$, T_p$) in Figure 2.4 for Table 2.2. Consider the third pattern tuple $(-, -, UK | -, -)$ in Figure 2.4, it covers seven tuples $t_8, t_9, t_{10}, t_{11}, t_{15}, t_{16}$ and t_{17}. Those seven tuples violate this pattern tuple; removing either t_{16} or t_{17} will resolve the violation. Hence, the local support of this pattern tuple is $\frac{7}{20}$ and its local confidence is $\frac{6}{7}$. Similarly, the local support of $(-, \text{clothing}, - | -, -)$ is $\frac{7}{20}$ and its local confidence is 1. The local support of $(-, \text{book, France} | -, 0)$ is $\frac{5}{20}$ and its local confidence is $\frac{4}{5}$ (t_3 causes the violation).

The global support of all three pattern tuples is $\frac{15}{20}$ with a global confidence of $\frac{13}{15}$.

Given the definition of GS, GC, LS, LC, two versions of the pattern tableaux generation problem are defined: the first version is called *pattern tableaux generation with GS and GC*, *i.e.*, given an embedded FD $X \rightarrow Y$ on R, an instance I of R, and two thresholds (s, c), find the T_p of the smallest size, such that $GS(T_p) \geq s$ and $GC(T_p) \geq c$. Not only is the problem NP-complete, but it is also provably hard to approximate with $|I|^{0.5-\epsilon}, \epsilon > 0$ [60]. The second version is called *pattern tableaux generation with GS and LC*, *i.e.*, given an embedded FD $X \rightarrow Y$ on R, an instance I of R, and two thresholds (s, c), find the T_p of the smallest size, such that $GS(T_p) \geq s$ and $LC(t_p) \geq c, \forall t_p \in T_p$. The problem is reduced to a variant of the partial set cover problem. Algorithm 6 gives a greedy approach to the problem.

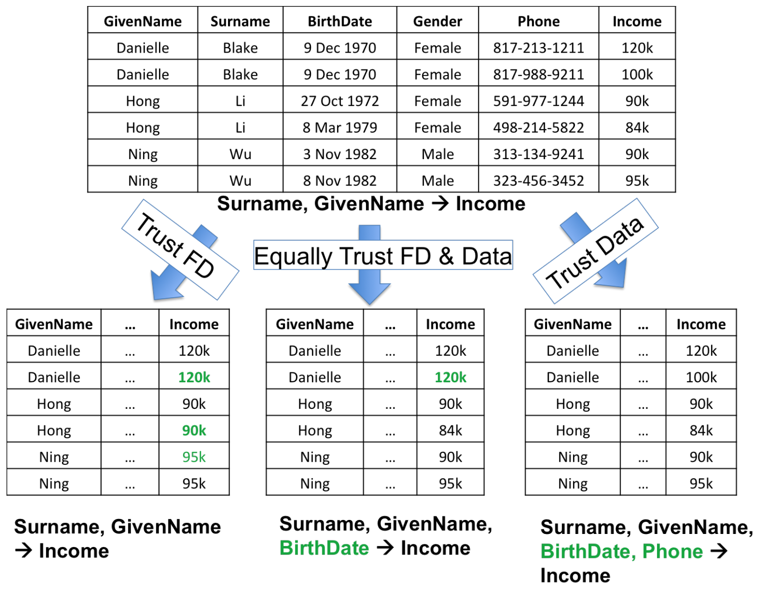
```
Algorithm 6 Tableaux Generation with GS and LC
Input: Database instance $I$ of schema $R$, two thresholds $(s, c)$, and an
    embedded FD $X \to Y$
Output: Pattern tableaux $T_p$
1: Generate all possible pattern tuples from active domain of $R$, and
   compute their local support and local confidence
2: Remove pattern tuples whose local confidence is below $c$
3: Iteratively choose the pattern tuple $t_p$ with the highest marginal
   support, add $t_p$ to $T_p$
4: Stop when the global support of $T_p$ is greater than $s$
5: return $T_p$
```

Algorithm 6 computes the support and confidence of every possible candidate pattern tuple and then iteratively chooses pattern tuples with the highest marginal support (and those which are above the confidence threshold), adjusting the marginal supports for the remaining candidate patterns after each selection, until the global support threshold is met or until all candidate patterns are exhausted.

3.1.3 Both Data and Rule Repairing

Cleaning techniques in this category assumes data and rules can be dirty at the same time [10, 23, 113]. Given a database instance I and a set of FDs Σ such that $I \not\models \Sigma$, we need to find another I' and Σ', such that $I' \models \Sigma'$.

Example 3.7. Figure 3.5 shows a table with an FD stating that given name and surname determine income. There are three violations of the FD, *i.e.*, the first and the second tuple, the third and the fourth tuple, and the fifth and the sixth tuple. If the FD is to be completely trusted, three cell changes are required, shown in the bottom left table in Figure 3.5. If the data is completely trusted, two attributes are added to the LHS of the FD, shown in the bottom middle table in Figure 3.5. If the FD and data have equal trustworthiness, a repair is to only change



GivenName	Surname	BirthDate	Gender	Phone	Income
Danielle	Blake	9 Dec 1970	Female	817-213-1211	120k
Danielle	Blake	9 Dec 1970	Female	817-988-9211	100k
Hong	Li	27 Oct 1972	Female	591-977-1244	90k
Hong	Li	8 Mar 1979	Female	498-214-5822	84k
Ning	Wu	3 Nov 1982	Male	313-134-9241	90k
Ning	Wu	8 Nov 1982	Male	323-456-3452	95k

Surname, GivenName → Income

GivenName	...	Income
Danielle	...	120k
Danielle	...	**120k**
Hong	...	90k
Hong	...	**90k**
Ning	...	95k
Ning	...	95k

Surname, GivenName → Income

GivenName	...	Income
Danielle	...	120k
Danielle	...	**120k**
Hong	...	90k
Hong	...	84k
Ning	...	90k
Ning	...	95k

Surname, GivenName, BirthDate → Income

GivenName	...	Income
Danielle	...	120k
Danielle	...	100k
Hong	...	90k
Hong	...	84k
Ning	...	90k
Ning	...	95k

Surname, GivenName, BirthDate, Phone → Income

Figure 3.5: Relative trust of FDs and data.

one cell value and add one attribute to the LHS of the FD, shown in the bottom right table in Figure 3.5.

In the following, we discuss three approaches for repairing FDs and data relying on the notion of relative trust [10], unified cost [23], and continuous cleaning [113].

Relative Trust Between Data and Constraints

Beskales et al. [10] model the universe of all possible repairs of (I, Σ) to include all pairs of (I', Σ') such that, $I' \models \Sigma'$. Let $\Delta(I, I')$ denote the distance between I and I' (*e.g.*, the number of different cells between I and I'). Let $\Delta(\Sigma, \Sigma')$ denote the distance between Σ and Σ' (*e.g.*, the number of attributes added to the LHS of the FDs in Σ). A repair (I', Σ') is said to be *minimal* if there does not exist another repair (I'', Σ''), such that $\Delta(I, I'') \leq \Delta(I, I')$ and $\Delta(\Sigma, \Sigma'') < \Delta(\Sigma, \Sigma')$, or $\Delta(I, I'') < \Delta(I, I')$ and $\Delta(\Sigma, \Sigma'') \leq \Delta(\Sigma, \Sigma')$. In other words, a repair

(I', Σ') is *minimal* if and only if no other repair (I'', Σ'') dominates it in terms of the two distances. Minimal repairs spread a wide spectrum of repairs, ranging from completely trusting I and only changing Σ to completely trusting Σ and only changing I. The relative trust on I is defined as the maximum number of allowed cell changes τ. A τ-*constrained repair* is a repair that has the minimum distance to Σ across all repairs with distance to I less than or equal to τ.

Algorithm 7 `Relative trust of FDs and data`

```
Input: Database instance $I$, a set of FDs $\Sigma$, a threshold $\tau$
Output: Another instance $I'$ and $\Sigma'$, such that $I' \models \Sigma'$
1: obtain $\Sigma'$ such that $\delta_{opt}(\Sigma', I) \leq \tau$, and no other $\Sigma''$ with
   $\delta_{opt}(\Sigma'', I) \leq \tau$ has $\Delta(\Sigma, \Sigma') < \Delta(\Sigma, \Sigma'')$, where $\delta_{opt}(\Sigma', I)$ is the
   minimum number of cells that need to be changed in $I$ for $I$ to
   satisfy $\Sigma'$
2: if $\Sigma' \neq \emptyset$ then
3:     obtain $I'$ that satisfied $\Sigma'$ with at most $\delta_{opt}(\Sigma', I)$ changes
4:     return $(I', \Sigma')$
5: else
6:     no repair
```

Algorithm 7 describes the procedure for computing a repair (I', Σ'), given a relative trust level τ on I. It consists of two major steps: (1) find the closest Σ' to Σ, such that there exists a repair for I that at most changes τ cells; and (2) obtain the actual repair I' given Σ. The space of possible repairs is modelled as a state space, where each state represents extending the LHS of the FDs in the original Σ. Figure 3.6 depicts an example search space for $\Sigma = \{A \rightarrow B, C \rightarrow D\}$. The root node represents adding nothing to the LHS of the two FDs, while the child node (C, ϕ) represents adding C to the LHS of the first FD and not adding anything to the LHS of the second FD, representing the new $\Sigma' = \{A \rightarrow BC, C \rightarrow D\}$.

To perform Step (1), searching the space for optimal Σ', the algorithm effectively navigates the space of all possible repairs of Σ, while computing $\delta_{opt}(\Sigma', I)$, *i.e.*, the minimum number of cells that need to be changed in I for I to satisfy Σ', without actually performing the

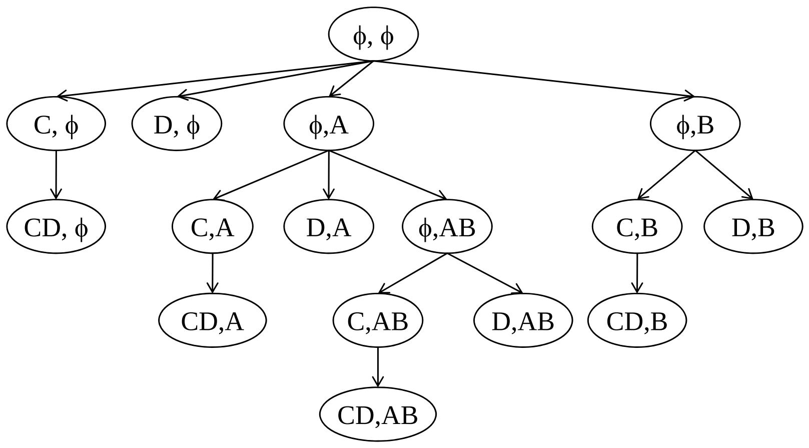


Figure 3.6: A space for $R = \{A, B, C, D\}$ and $\Sigma = \{A \to B, C \to D\}$

cleaning. Since computing $\delta_{opt}(\Sigma', I)$ is an NP-hard problem [17], A tuple-based *conflict hypergraph* is used to approximate $\delta_{opt}(\Sigma', I)$ by a factor of $2 \times \min\{|R| - 1, |\Sigma|\}$. An $A^{\star}$ based algorithm is used to navigate the space of all possible repairs of Σ. Step (2), the actual repairing of I with respect to Σ' found in Step (1), is performed using any automatic FD violation repairing algorithm, such as those described in Section 3.1.1.

Unified Cost of Changing Data and Constraints

In contrast to the aforementioned approach [10], which treats the cost of repairing constraints and data separately, Chiang et al. [23] propose a unified cost model for repairing data and FDs on an equal footing based on the Minimum Description Length (MDL) principle. Based on the cost model, Chiang et al. associate a cost for a database instance I and a set of FDs Σ. Given a database instance I and Σ, such that $I \not\models \Sigma$, the goal is to find another database instance I' and Σ', such that $I' \models \Sigma'$, and the cost associated with I' and Σ' is minimized.

We describe how the model is built, and used if there a single FD in Σ. Assume an FD $\varphi : X \to Y$ defined over relational schema R and an instance I of R; a model M is built for φ. The model M consists of a set of *signatures*, where each signature s is a single tuple in I projected on XY, *i.e.*, $s \in \Pi_{XY}(I)$. M uses a unit cost for each cell in a relation. The description length DL for M is defined as the length of the model $L(M)$, plus the length to encode the data values in the relation I, given the model $L(I|M)$. See that $L(M)$ is calculated as $L(M) = |XY| \times S$, where $|XY|$ is the number of attributes in XY and S is the number of signatures in M, and $L(I|M)$ is calculated as $L(I|M) = |XY| \times E$, where E is the number of tuples in I whose projection on XY is not represented by any signature s in M. If the model is empty, namely, $L(M) = 0$, then $DL = L(I|M) = |XY| \times |I|$, where $|I|$ is the number of tuples in I. As more signatures are added to M that do not conflict with existing ones, $L(M)$ increases while $L(I|M)$ decreases. The goal is to find an M, such that $DL = L(M) + L(I|M)$ is minimized. Given an FD φ and an instance I of R, an initial model M is built by adding those signatures into M whose support is more than a predefined threshold, where the support of a signature s is the number of tuples having s as their values for XY attributes. To resolve the violations of φ, either the data repair or the constraint repair is chosen depending on which repair results in a larger reduction in DL.

If there are multiple FDs in Σ, they are processed in an order depending on (1) the number of violations of an FD; and (2) the potential conflict an FD shares with other FDs defined based on the number of overlapping attributes.

Continuous Data Cleaning

Rather than repairing data and constraints in a single snapshot of the database, continuous data cleaning [113] considers repairing both FDs and the data in a dynamic environment, where data may change frequently and constraints may evolve. The continuous data cleaning framework is shown in Figure 3.7.

In the first stage, a probabilistic classifier that predicts the repair type (data, FDs, or a hybrid of both) is trained using repairs that have been selected and validated by a user (Figure 3.7 (a)). These repairs provide a baseline to the classifier representing the types of modifications that align with the user and the application preferences.

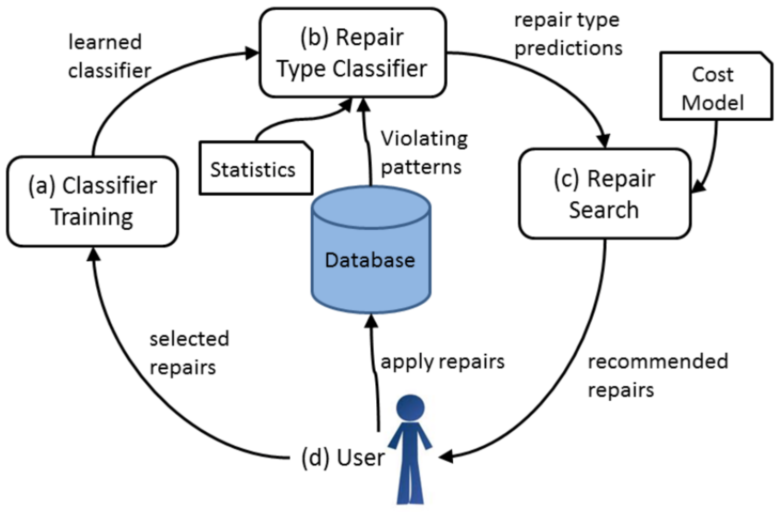


Figure 3.7: Continuous data cleaning.

As the data and the constraints change, inconsistencies may arise that need to be resolved. Once the classifier is trained, it predicts the types of repairs needed to address the violations (Figure 3.7 (b)). A set of statistics are calculated to describe the properties of the violations, such as the number of violating tuples, and the number of violating FDs. The classifier generates predications and computes the probability of each repair type (data, FD, or a hybrid of both). These repair predictions are passed to the repair search algorithm, which narrows the search space of repairs based on the classifier's recommendation (Figure 3.7 (c)). The repair search algorithm includes a cost model that determines which repairs are best to resolve the inconsistencies. The repair search algorithm recommends a set of data and/or FD repairs to the user, who will decide which repairs to apply (Figure 3.7 (d)). The applied repairs are then used to re-train the classifier, and the process is repeated. Incremental changes to the data and to the constraints are passed to the classifier (Figure 3.7 (b)), and reflected via the statistics and the patterns.

3.2 How to Repair

In this section, we discuss data repairing techniques based on whether and how humans are involved in the repairing process.

3.2.1 Automatic Repairing

There exist multiple theoretic studies [17, 2, 24] and surveys [44, 8] on studying the complexity of data repairing parameterized by different classes of ICs, such as FDs, CFDs, and DCs, and different repairing operations, such as value updating, and tuple deleting. In this section, we discuss data repairing techniques that aim at updating the database in a way such that the distance between the original database I and the modified database I' is minimized. With a lack of ground truth, the main hypothesis behind the minimality objective function is that a majority of the database is clean, and, thus, only a relatively small number of updates need to be performed compared to the database size.

Let $\Delta(I, I')$ denote the set of cells that have different values in I and I', *i.e.*, $\Delta(I, I') = \{C \in CIDs(I) : I(C) \neq I'(C)\}$.

Definition 3.1 (Cardinality-Minimal Repair). A repair I' of I is cardinality-minimal if and only if there is no repair I'', such that $|\Delta(I, I'')| < |\Delta(I, I')|$.

A repair I' of I is cardinality-minimal if and only if the number of changed cells in I' is minimum among all possible repairs of I. The automatic repairing algorithm in [77] aims to find cardinality-minimal repairs for FD violations.

A weighted version of the cardinality-minimal repair associates a weight with each cell, reflecting the confidence in the correctness of the cell [17, 31, 26]. In addition, the distance between cell $I(C)$ and $I'(C)$ is measured using a distance function $dis(I(C), I'(C))$, instead of binary 0 or 1 in cardinality-minimal repair. The cost of a repair I' of I is defined as $cost(I, I') = \sum_{C \in \Delta(I, I')} dis(I(C), I'(C))$.

Definition 3.2 (Cost-Minimal Repair). A repair I' of I is cost-minimal if and only if there is no repair I'', such that $cost(I, I'') < cost(I, I')$.

In Section 3.1 we gave the details of an example cost-minimal repairing algorithm (Algorithm 4 [17]) in the context of discussing data repair while trusting the declared quality rules. It has been shown [17] that, even the set of ICs Σ are FDs only, the problem of determining if there exists a repair I' such as $cost(I, I') < W$, for a given constant W, is NP-complete. Obviously, for constraints that are more expressive than FDs, such as DCs, the data repairing problem is even harder.

Definition 3.3 (Set-Minimal Repair)**.** A repair I' of I is set-minimal if and only if there is no repair I'', such that $\Delta(I, I'') \subset \Delta(I, I')$ and for each $C \in \Delta(I, I''), I''(C) = I'(C)$.

A repair I' of I is set-minimal if and only if no subset S of the changed cells in I' can be reverted back to their original values while keeping the current values of other cells in $Delta(I, I') \backslash S$ unchanged. Most existing literature on consistent query answering assumes set-minimal repairs [7, 83, 8].

Definition 3.4 (Cardinality-Set-Minimal Repair)**.** A repair I' of I is cardinality-set-minimal if and only if there is no repair I'', such that $\Delta(I, I'') \subset \Delta(I, I')$.

A repair I' of I is cardinality-set-minimal if and only if no subset S of the changed cells in I' can be reverted back to their original values, even if the current values of cells in $\Delta(I, I') \backslash S$ are allowed to be changed to other values.

Example 3.8. Figure 3.8 shows several examples of different notions of minimal repairs. Repair I_1 is cardinality-minimal because no other repair has fewer changed cells. By definition, Repair I_1 is also cardinality-set-minimal and set-minimal. Repairs I_2 and I_3 are set-minimal because reverting any subset of the changed cells to the values in I will violate $A \rightarrow B$. On the other hand, I_3 is not cardinality-set-minimal (hence not a cardinality-minimal) because reverting $t_2[B]$ and $t_3[B]$ back to 3 and changing $t_1[B]$ to 3 instead of 5 gives a repair of I, which is the same as I_1. I_3, however, is set-minimal, since reverting any subset of the changed cells back to the values in I will still violate the FD. Repair I_4 is not set-minimal because I_4 still satisfies $A \rightarrow B$ after reverting $t_1[A]$ to 1.

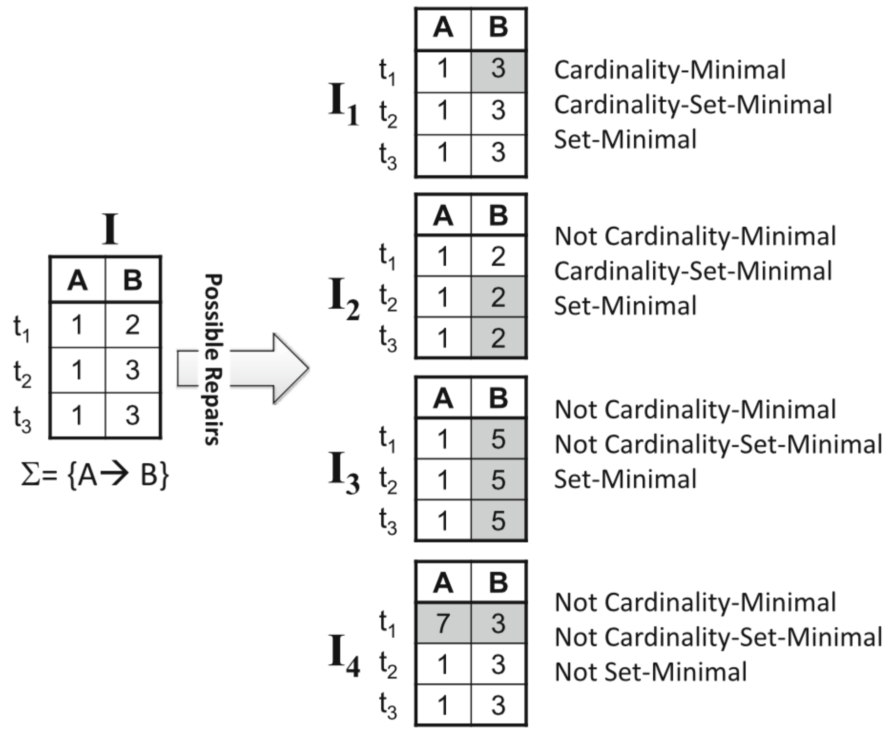


Figure 3.8: Examples of various types of repairs.

Figure 3.9 shows the relationships between different notions of minimality in a graph [11]. The set of cardinality-minimal repairs is a subset of cardinality-set-minimal repairs. The set of cardinality-set-minimal repairs is a subset of set-minimal repairs. Finally, the set of cost-minimal repairs is a subset of set-minimal repairs if for each cell $C \in CIDs(I), w(C) > 0$. In general, cost-minimal repairs are not necessarily cardinality-minimal or cardinality-set-minimal, and vice versa. However, for a constant weighting function w (*i.e.*, all cells are equally trusted) and a constant distance function dis (*i.e.*, the distance between any pair of values is the same), the set of cost-minimal repairs and the set of cardinality-minimal repairs coincide.

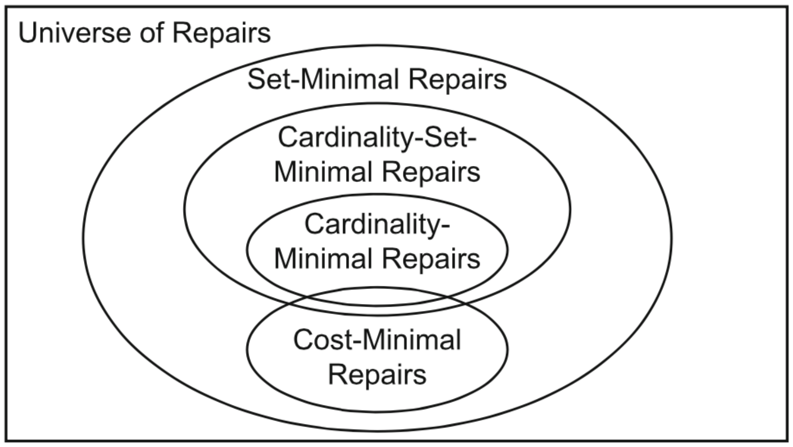


Figure 3.9: The relationships between different notions of minimality.

In the following, we give the details of an algorithm that produces cardinality-minimal repairs [77]. A technique that samples from the space of cardinality-set-minimal repairs for FDs [9] is discussed in Section 3.3 in the context of creating repairing models, instead of cleaning data in-situ.

Generating Cardinality-Minimal Repairs

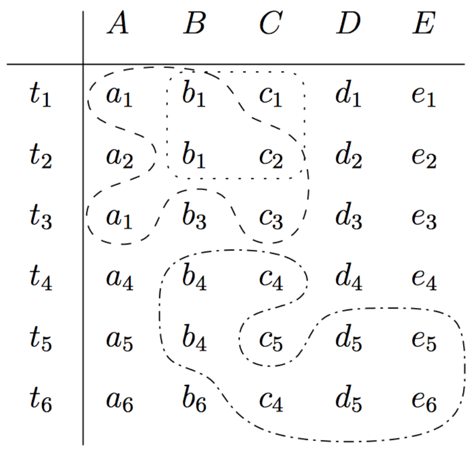
Algorithm 8 finds a repair I' whose distance to I (*i.e.*, number of changed cell) is within a constant factor of the optimum repair distance, where the constant factor depends on the set of FDs [77]. The algorithm captures the interplay among the defined FDs in a hypergraph, where each node represents a cell in the database, and a hyperedge comprising multiple cells that cannot coexist together. We call this data structure, a *Conflict Hypergraph*. The algorithm uses the notion of *core implicant* to ensure the termination of the algorithm. A *core implicant* of an attribute A w.r.t. a set of FDs Σ is a minimal set C_A of attributes such that C_A has at least one common attribute with every implicant X of A, where X is an implicant of A if Σ implies a nontrivial FD $X \rightarrow A$. A *minimal core implicant* of an attribute A is the core im-

Algorithm 8 `FindVRepairFDs`

```
Input: Database instance $I$, a set of FDs $\Sigma$
Output: Another instance $I'$, such that $I' \models \Sigma$
 1: create an initial conflict hypergraph $\mathcal{G}_I$ for $I$
 2: find an approximation $VC$ for minimum vertex cover in $\mathcal{G}_I$
 3: $change \leftarrow VC$
 4: $I' \leftarrow I$
 5: while there exists two tuples $t_1, t_2 \in I'$ violating an FD $X \rightarrow A \in \Sigma$
    and $t_1[A]$ is the only cell in $VC$ do
 6:   $t_1[A] \leftarrow t_2[A]$
 7:   $change \leftarrow change - t_1[A]$
 8: for all Cell $t[B] \in change$ do
 9:   $I'(t[B]) \leftarrow$ fresh variable
10: if there are new violations then
11:   let $t[B] \leftarrow$ a cell in $VC$ with the largest number of violations
      involving $t[B]$
12:   let $CI$ be the set of attributes in the minimal core implicant of
      Attribute $B$ w.r.t. $\Sigma$
13:   for all Attribute $C \in CI \cup B$ do
14:     $I'(t[C]) \leftarrow$ fresh variable
```

plicant with the smallest number of attributes. Intuitively, by putting variables in Attribute A and all attributes in the core implicant of A, all violations involving A are resolved, and no more new violations can be introduced, where a variable denotes an unknown value that is not in the active domain, where two different variables will have different values.

The algorithm works as follows. First of all, an initial conflict hypergraph $\mathcal{G}_I$ is built for I. Then an approximate minimum vertex cover, VC, in $\mathcal{G}_I$ is found. For each cell in VC, either a value from the active domain (values that appear in the instance) is chosen if it satisfies the set of defined FDs, if a new variable is chosen. After all cells in VC have been changed, the resulting I' may contain new violations. A new violation of an FD $X \rightarrow B$ is resolved by putting variables in one of the violating tuple for Attributes B and the attributes in the minimal core implicant of B, which ensures that no more violations are introduced [77].



	A	B	C	D	E
t_1	a_1	b_1	c_1	d_1	e_1
t_2	a_2	b_1	c_2	d_2	e_2
t_3	a_1	b_3	c_3	d_3	e_3
t_4	a_4	b_4	c_4	d_4	e_4
t_5	a_5	b_4	c_5	d_5	e_5
t_6	a_6	b_6	c_4	d_5	e_6

Figure 3.10: An initial conflict hypergraph

Example 3.9. Consider a relational schema $R(A, B, C, D, E)$ with FDs $\Sigma = \{A \rightarrow C, B \rightarrow C, CD \rightarrow E\}$. An instance I is shown in Figure 3.10 with three hyperedges of three different types (not all hyperedges are shown). The first type of hyperedge is due to violation of a single FD, such as Hyperedge e_1 that consists of four cells: $t_1[B], t_2[B], t_1[C]$ and $t_2[C]$, which together violate the FD $B \rightarrow C$. The second type of hyperedge is due to the interaction of two FDs that share the same RHS attribute, such as Hyperedge e_2 that consists of six cells: $t_1[A], t_1[B], t_2[B], t_2[C], t_3[A]$ and $t_3[C]$, which cannot coexist together due to the two FDs $A \rightarrow C$ and $B \rightarrow C$. The third type of hyperedge is due to the interaction of two FDs, where the RHS of one FD is part of the LHS of the other, such as Hyperedge e_3 that consists of eight cells: $t_4[B], t_4[C], t_5[B], t_5[D], t_5[E], t_6[C], t_6[D]$ and $t_6[E]$, which cannot coexist together due to the two FDs $B \rightarrow C$ and $CD \rightarrow E$.

There are two other hyperedges not shown in Figure 3.10: Hyperedge e_4, that consists of four cells: $t_1[A], t_1[C], t_3[A]$ and $t_3[C]$, and Hyperedge e_5, that consists of four cells: $t_4[B], t_4[C], t_5[B]$ and $t_5[C]$.

Suppose $VC = \{t_2[C], t_3[C], t_4[B]\}$. Algorithm 8 enforces $t_2[C]$ to be the value c_1 of $t_1[C]$ because $t_2[C]$ is the only cell in VC among all cells in Hyperedge e_1. Similarly, $t_3[C]$ is assigned the value c_1 of $t_1[C]$. $t_4[B]$ is changed to a fresh variable. Algorithm 8 terminates after all cells in VC are changed, because there is no more new violations introduced.

3.2.2 Human Guided Repairs

Automatic data repairing techniques use heuristics, such as minimal repairs to automatically repair the data in situ, and they often generate unverified fixes. Worse still, they may even introduce new errors during the process. It is often difficult, if not impossible, to guarantee the accuracy of any data repairing techniques without external verification via experts and trustworthy data sources.

Example 3.10. Consider two tuples t_1 and t_8 in Table 1.1; they both have the same values "25813" for ZIP attribute, but t_1 has "WA" for ST attribute and t_8 has "WV" for ST attribute. Clearly, at least one of the four cells $t_1[ZIP], t_8[ZIP], t_1[ST], t_8[ST]$ has to be incorrect. Lacking other evidence, existing automatic repairing techniques [17, 26] often randomly choose one of the four cells to update. Some of them [17] even limit the allowed changes to be $t_1[ST], t_8[ST]$, since it is unclear which values $t_1[ZIP], t_8[ZIP]$ should take if they are to be changed.

This shooting in the dark approach, adopted by most automatic data cleaning algorithms, motivated new approaches that effectively involve humans or experts in the cleaning process to generate reliable fixes. In the following, we list a few examples: AJAX [53] shows how to involve users in a data cleaning process modeled as a directed graph of data transformations; Potter's Wheel [94] is an interactive data cleaning system that tightly integrates data transformation and discrepancy detection; Data Wrangler [74, 64] extends Potter's Wheel's data transformation language; GDR (guided data repair) [126] shows how to effectively incorporate user feedback into CFDs repairing algorithms; Editing rules [50] uses tabular master data and humans to generate verified fixes; KATARA [28] combines KBs (*e.g.*, Yago and DB-

Pedia), which is a collection of curated facts, such as *China hasCapital Beijing*, and crowdsourcing to discover and verify table patterns, identify errors, and suggest possible fixes; and Data Tamer [105] is a data curation system that involves users with different expertise at multiple steps of the curation process.

AJAX

AJAX [53] is a data cleaning framework that separates the logic and physical levels of data cleaning. The logic level supports the design of the data flow graph that models the data transformations needed to clean the data, while the physical level supports the implementations and optimizations of the data transformations. AJAX provides a declarative language, which is SQL enriched with a set of specific primitives, to express data transformations, such as matching two records. AJAX raises an exception, implemented via Java exception mechanism, whenever the data transformation process fails, and the users are expected to manually examine and resolve the exceptions. AJAX was also the first to be explicit about the work flow of data deduplication by mapping it into a sequence of three operations of matching, clustering, and merging. AJAX is later extended with the notion of quality constraints [54] imposed on relations within the directed graph of data transformation, and the violations of the quality constraints can be inspected by manual data repairs.

Potter's Wheel

Potter's Wheel [94] is an interactive data cleaning system that tightly integrates data transformation and discrepancy detection. Figure 3.11 shows the architecture of Potter's Wheel, which takes as input a *data source*. Data read from the input data source is displayed on a scalable *Spreadsheet display* that allows users to interactively re-sort the data on any column, and scroll in a representation sample of the data. The online re-sorting is supported by *online reorderer* that maintains a histogram on the sort column. The *discrepancy detection* is done at the background on the newly transformed data, and anomalies are flagged as soon as they are found. Users examine those flagged anoma-

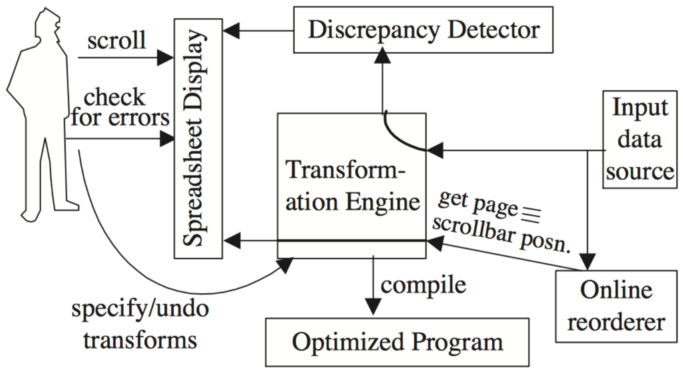


Figure 3.11: Potter's Wheel Architecture

lies, and specifies appropriate transformations, which are recorded by the *transformation engine*. The specified transformation are applied to the records already rendered on the screen, as well as those records used for discrepancy detection. After the user is satisfied with the sequence of transforms, Potter's Wheel can compile it as *optimized program*, which can be applied on the current dataset, or can be invoked on other datasets.

Data Wrangler

Data Wrangler [74, 64] is a follow-up work built on Potter's Wheel, which extends the data transformation language to include additional common data transformation operations, such as positional transformations and reshaping transformations. One example of positional transformations is *Fill*, which fills missing values based on neighboring values in a row or column; one example of reshaping transformations is *Unfold*, which creates new column headers from data values. Data Wrangler reactively suggests a list of possible transformations from user selection in a visualized interface [74], and also proactively suggests data transformations which maps input data to a relational format expected by analysis tools [64]. To aid the user to select one transformation from the list of suggested transformations, Data Wrangler provides a short natural language description for each transformation, as well as vi-

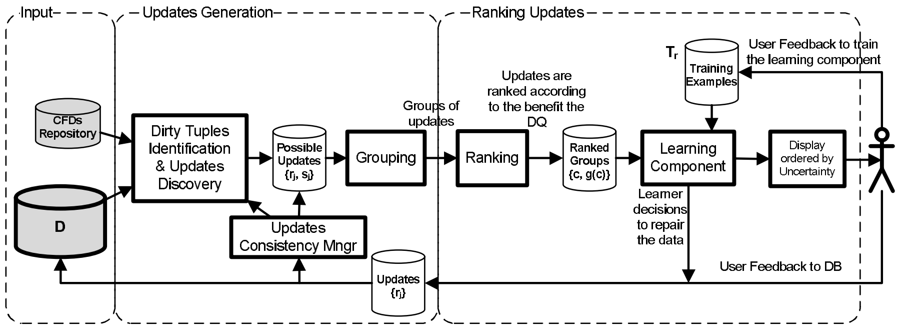

Input
Updates Generation
Ranking Updates
CFDs Repository
D
Dirty Tuples Identification & Updates Discovery
Possible Updates {rj, sj}
Grouping
Updates Consistency Mngr
Updates {rj}
Groups of updates
Updates are ranked according to the benefit the DQ
Ranking
Ranked Groups {c, g(c)}
Tr
Training Examples
User Feedback to train the learning component
Learning Component
Display ordered by Uncertainty
Learner decisions to repair the data
User Feedback to DB

Figure 3.12: Guided data repair framework

sual previews to enable users to quickly evaluate the effect of a data transform. The design framework that relieves users from the burden of specifying the details of desired transformation is called *Predictive Interaction* [66].

Guided Data Repair

Guided data repair (GDR) [126] incorporates user feedback in the data cleaning process to enhance and accelerate automatic data repairing techniques for CFDs while minimizing user involvement. Figure 3.12 shows the GDR framework.

Given a database instance and a set Σ of CFDs, violations of Σ are considered *dirty* tuples; they are identified and stored in a *DirtyTuples* list. For an attribute A in a dirty tuple t violating a CFD $\varphi \in \Sigma$, an *on demand* update discovery process based on the mechanism described in [31] for resolving CFDs violations and generating candidate updates is used to suggest an update for the cell $t[A]$. Initially, the process is called for all dirty tuples and their attributes. Later, during the phase of user interaction, a *consistency manager* triggers the repairing process upon receiving user feedback. The generated updates are tuples in the form $r_j = \langle t, A, v, s_j \rangle$ stored in the *PossibleUpdates* list, where v is the suggested value for $t[A]$ and s_j is the *update score* assigned to each update r_j to reflect the confidence of the repairing technique in the suggested update. Once an update $r = \langle t, A, v, s \rangle$ is confirmed to

	Name	SRC	STR	CT	STT	ZIP
t1:	Jim	H1	REDWOOD DR	MICHIGAN CITY	MI	46360
t2:	Tom	H2	REDWOOD DR	WESTVILLE	IN	46360
t3:	Jeff	H2	BIRCH PARKWAY	WESTVILLE	IN	46360
t4:	Rick	H2	BIRCH PARKWAY	WESTVILLE	IN	46360
t5:	Joe	H1	BELL AVENUE	FORT WAYNE	IN	46391
t6:	Mark	H1	BELL AVENUE	FORT WAYNE	IN	46825
t7:	Cady	H2	BELL AVENUE	FORT WAYNE	IN	46825
t8:	Sindy	H2	SHERDEN RD	FT WAYNE	IN	46774

(a) Data

$$\phi_1 : (\texttt{ZIP} \rightarrow \texttt{CT}, \texttt{STT}, \{46360 \parallel \text{MichiganCity}, \text{IN}\})$$
$$\phi_2 : (\texttt{ZIP} \rightarrow \texttt{CT}, \texttt{STT}, \{46774 \parallel \text{NewHaven}, \text{IN}\})$$
$$\phi_3 : (\texttt{ZIP} \rightarrow \texttt{CT}, \texttt{STT}, \{46825 \parallel \text{FortWayne}, \text{IN}\})$$
$$\phi_4 : (\texttt{ZIP} \rightarrow \texttt{CT}, \texttt{STT}, \{46391 \parallel \text{Westville}, \text{IN}\})$$
$$\phi_5 : (\texttt{STR}, \texttt{CT} \rightarrow \texttt{ZIP}, \{\_ , \text{FortWayne} \parallel \_\})$$

(b) CFD Rules

Figure 3.13: Guided data repair example.

be correct, either by the user or by the *learning component*, it is immediately applied to the database resulting into a new database instance. A set of updates are grouped together if they are updating the same attribute to the same value; grouping provides contextual information to make it easier for the user to verify the suggested repairs. The groups are ranked according to the expected quality gain of each group. The quality gain is estimated by computing the difference between the expected number of violations before and after processing the updates in that group. The cost of acquiring user feedback for verifying each update is reduced by training a machine learning classifier (using an active learning technique) to replace the user later in the process. The use of a learning component in GDR is motivated by the correlation between the original data and the correct updates. If this correlation can be identified and represented in a classification model, then the model can be trained to predict the correctness of a suggested update and hence replace the user for similar (future) situations.

Example 3.11. Consider the following example. Let Relation `Customer`(`Name`, `SRC`, `STR`, `CT`, `STT`, `ZIP`) specify personal address information Street (`STR`), City (`CT`), State (`STT`) and (`ZIP`), in addition to the source (`SRC`) of the data. An instance of this relation is shown in Figure 3.13 along with a set of CFDs.

Assume that a cleaning algorithm gives two groups of updates: the first group suggests assigning Attribute `CT` to the value 'Michigan City' for t_2, t_3, and t_4; and the second group suggests assigning Attribute `ZIP` with the value 46825 for t_5 and t_8. Assume further that the user provides the correct values for these tuples; the user has confirmed 'Michigan City' as the correct `CT` for t_2, t_3, but an incorrect `CT` for t_4, and 46825 as the correct `ZIP` for t_5, but an incorrect `ZIP` for t_8. The hypothesis in GDR is that consulting the user on the first group, which has more correct updates, is better and would allow for faster convergence to a cleaner database instance as desired by the user.

There could be a correlation between the attribute values in a tuple and the correct updates. For example, when `SRC` = 'H2', `CT` is incorrect most of the time, while `ZIP` is correct. This is an example of a recurrent mistake that exists in real data. Patterns such as this with a correlation between the original tuple values and the correct updates, if captured by a machine learning algorithm, can reduce user involvement.

Editing Rules

Another example of involving humans to generate verified fixes is to use editing rules (cf. Section 2.1.1) [50]. Given an input tuple $t \in I$ of schema R to be fixed, a set of eRs Σ, and a master data relation I_m of schema R_m, a certain fix t' of t needs to be found by interacting with the users, *i.e.*, (1) no matter how Σ and tuples in I_m are applied, Σ and I_m will yield a unique t', and (2) all attributes of t' are guaranteed to be correct. Figure 3.14 depicts the framework for using Σ and I_m to derive t' for t. Specifically, the framework works as follows:

- *Initialization.* Given t, it picks a precomputed certain region Z

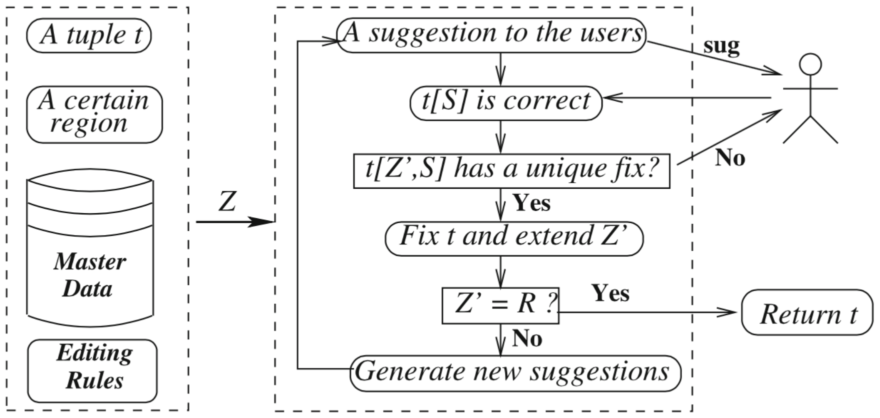


Figure 3.14: Repairing framework for using editing rules and master data.

and recommends Z to the user. A certain region is a set of attributes that are guaranteed to be correct. Z' denotes a set of attributes that have already been validated by the user to be correct.

- *Generating Correct Fixes.* In every interaction with the user, a set of attributes, initially Z, is shown to the user, who chose a subset S of asserted correct attributes. If $t[S \cup Z']$ leads to a unique fix, t is fixed and Z' is extended; otherwise, users are asked to provide new suggestions.
- *Generating New Suggestions.* If Z' covers all attributes, a certain fix has been found; otherwise, a new set of suggestions are computed for the next round of user interaction.

KATARA

KATARA [28] aims at producing accurate repairs by relying on two authoritative data sources, namely, knowledge bases (KBs), and domain experts. KATARA first discovers table patterns to map the table to a KB, such as Yago or DBPedia. With table patterns, KATARA annotates

	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 3.15: A table $\mathcal{T}$ for soccer players

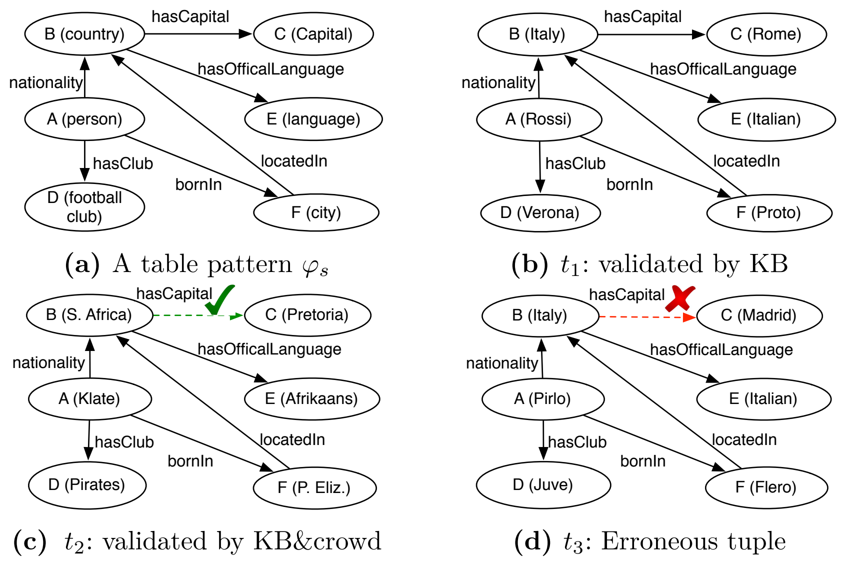


Figure 3.16: KATARA patterns

tuples as either correct or incorrect by interleaving the KB and humans. For incorrect tuples, KATARA will extract top-k mappings from the KB as possible repairs that are to be examined by humans.

Consider a table $\mathcal{T}$ for soccer players (Fig. 3.15). Table $\mathcal{T}$ has no table header, thus its semantics are completely unknown. Assume that a KB $\mathcal{K}$ (*e.g.* Yago) contains some information related to $\mathcal{T}$. KATARA [28] works as follows:

(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. 3.16a) 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 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 described by the pattern in Fig. 3.16a.

(2) Pattern validation. Consider a case where pattern discovery finds two similar patterns: the one in Fig. 3.16a, 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. 3.16a, KATARA annotates each tuple with one of the following three labels:

(i) *Validated by the* KB. By mapping tuple t_1 in table $\mathcal{T}$ to $\mathcal{K}$, we found a full match, shown in Fig. 3.16b, 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 with respect to attributes A-F are found in $\mathcal{K}$. We consider t_1 to be correct with respect to the pattern in Fig. 3.16a and to attributes A-F.

(ii) *Jointly validated by the* KB *and the crowd.* Consider t_2 about Klate, whose explanation is depicted in Fig. 3.16c. In $\mathcal{K}$, we find that S. Africa is a country, 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.* Similar to case (*ii*). For tuple t_3, there is no link from Italy to Madrid in $\mathcal{K}$. 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 extracts evidence from $\mathcal{K}$, *e.g.*, Italy hasCapital Rome and Spain hasCapital Madrid, joins them and generates a set of possible repairs for this tuple.

Data Tamer

Data Tamer [105] is a data curation system that cleans and transforms large scale data sources at the enterprise level. Data Tamer integrates the schema, and instances of these sources, through a series of mapping, deduplication and linking exercises. The core technological innovation of Data Tamer is the automation of the data curation process while involving various roles of data experts, including data owners, data stewards, data scientists, and data curators. This closely coupled design of machine and human enables practical, end-to-end curation at the scale of of hundreds to thousands of disparate data sets. Human experts and owners are involved in multiple tasks, including: (1) answering pairwise comparison questions for training machine learning models (e.g., classifiers), (2) validating the machine decision on matching attributes or records from data sources, and (3) providing explicit business rules for deduplication or cleaning data sources. A task-expertise matching system is used to dispatch human tasks to solve curation tasks at different granularities (e.g., comparing pairs of columns, comparing pairs of values, or validating a cluster of related entities). Data Tamer is an example of involving users to guide the cleaning process at multiple levels in the curation stack and at multiple granularities.

3.3 Where to Repair

Data repairing techniques are classified based on whether the database will be changed in place by the repairing techniques, or using a model that describes the possible changes that will be used to answer queries against the dirty data. Most of the proposed data repairing techniques (all discussed so far) identify errors in the data, and find a unique fix of the data either by minimally modifying the data according to a cost function or by using human guidance (Figure 3.17(a)).

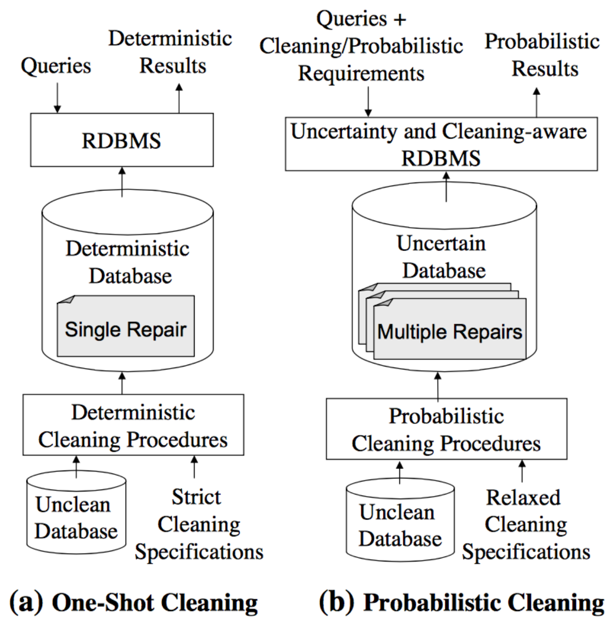

Queries
Deterministic Results
RDBMS
Deterministic Database
Single Repair
Deterministic Cleaning Procedures
Unclean Database
Strict Cleaning Specifications
(a) One-Shot Cleaning
Queries + Cleaning/Probabilistic Requirements
Probabilistic Results
Uncertainty and Cleaning-aware RDBMS
Uncertain Database
Multiple Repairs
Probabilistic Cleaning Procedures
Unclean Database
Relaxed Cleaning Specifications
(b) Probabilistic Cleaning

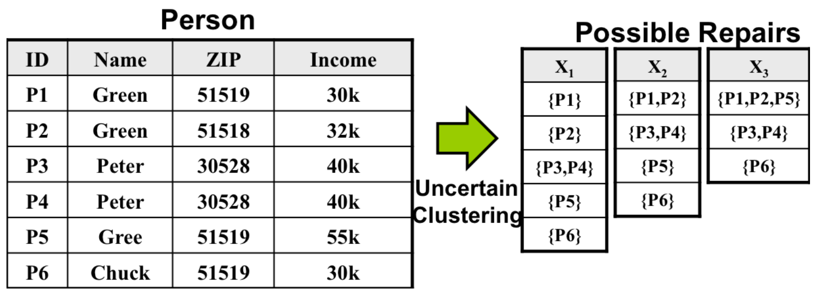
Figure 3.17: One-shot vs. probabilistic cleaning

As follows, we describe a different model-based approach for non-destructive data cleaning. Data repairing techniques in this category do not produce a single repair for a database instance; instead, they produce a space of possible repairs (Figure 3.17(b)). The space of possible repairs is used either to answer queries against the dirty data probabilistically (e.g., using possible worlds semantics) [12], or to sample from the space of all possible clean instances of the database [9, 11].

We give the details of two algorithms in this category. The first creates a succinct model of all possible duplicate-free instances from a dirty database with duplicates and provides a probabilistic query engine to answer queries against the dirty data [12]. The second algorithm considers the space of possible repairs of FD and CFD violations and provides a sampling technique to sample possible clean instances with certain minimality guarantees, more specifically, with cardinality-set-minimal repairs (cf. Section 3.2)[9, 11]. Techniques from consistent query answering [7, 83] also fall under this category, since they consider a tuple in the original dirty database instance I to be in the answer of a query, if that tuple is present in every possible repair I' of I. We refer readers to a survey [8] for a comprehensive treatment of consistent query answering.

Probabilistic Deduplication

Beskales et al. [12] study the problem of modeling and querying possible repairs in the context of duplicate detection, which is the process of detecting records that refer to the same real-world entity. Figure 3.18 shows an input relation representing sample census data that possibly contains duplicate records. Duplicate detection algorithms generate a clustering of records (represented as sets of record IDs in Figure 3.18), where each cluster is a set of duplicates that are eventually merged into one representative record per cluster. A one-shot duplicate detection approach identifies records as either duplicates or non-duplicates based on the given cleaning specifications (e.g., a single threshold on record similarity). Hence, the result is a single clustering (repair) of the input relation (*e.g.*, any of the three possible repairs shown in Figure 3.18). However, in the probabilistic duplicate detection approach, this restriction is relaxed to allow for uncertainty in deciding on the true duplicates (*e.g.*, based on multiple similarity thresholds). The result is a set of multiple possible clusterings (repairs), as shown in Figure 3.18.



Person

ID	Name	ZIP	Income
P1	Green	51519	30k
P2	Green	51518	32k
P3	Peter	30528	40k
P4	Peter	30528	40k
P5	Gree	51519	55k
P6	Chuck	51519	30k

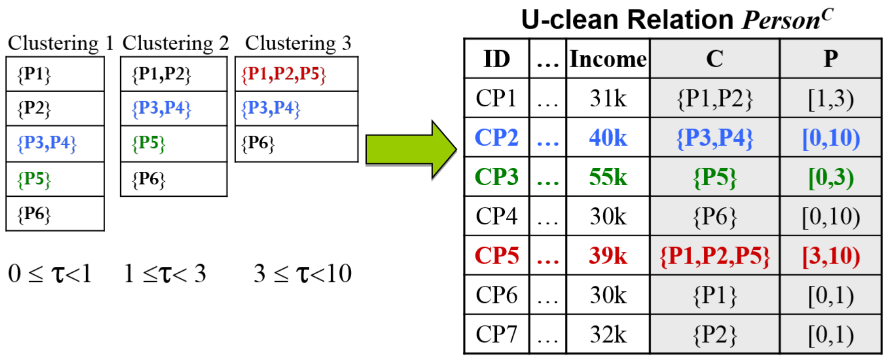
Possible Repairs

X_1	X_2	X_3
{P1}	{P1,P2}	{P1,P2,P5}
{P2}	{P3,P4}	{P3,P4}
{P3,P4}	{P5}	{P6}
{P5}	{P6}	
{P6}		

Figure 3.18: Probabilistic duplicate detection

Beskales et al. [12] constrain the space of all possible repairs to repairs generated by parameterized hierarchical clustering algorithms for two reasons: (1) the size of the space of possible repairs is linear in the number of records in the unclean relation, and (2) a probability distribution on the space of possible repairs can be induced based on the probability distribution on the values of the parameters of the algorithm. Specifically, let τ represent possible parameter values of a duplicate detection algorithm $\mathcal{A}$ (*e.g.*, τ could be the threshold value of deciding whether two clusters should be merged in a hierarchical clustering algorithm), let $[\tau^l, \tau^u]$ represent the possible values of τ, and let f_τ represent the probability density function of τ defined over $[\tau^l, \tau^u]$. The set of possible repairs $\mathcal{X}$ is defined as $\{\mathcal{A}(R,t) : t \in [\tau^l, \tau^u]\}$. The set $\mathcal{X}$ defines a probability space created by drawing random parameter values from $[\tau^l, \tau^u]$, based on the density function f_τ, and using the algorithm $\mathcal{A}$ to generate the possible repairs corresponding to these values. The probability of a specific repair $X \in \mathcal{X}$, denoted $\Pr(X)$, is equal to the probability of the parameter range that generates such repair.

Uncertain clean relation (*U-clean relation* for short) is used to encode the possible repairs $\mathcal{X}$ of an unclean relation R generated by a parameterized clustering algorithm $\mathcal{A}$. A U-clean relation, denoted R^c, is a set of c-records where each c-record is a representative record of a cluster of records in R. Attributes of R^c are all attributes of Relation R, in addition to two special attributes: C and P. Attribute C of a c-record is the set of record identifiers in R that are clustered together to form this c-record. Attribute P of a c-record represents the parameter settings of the clustering algorithm $\mathcal{A}$ that lead to generating the cluster represented by this c-record. Figure 3.19 illustrates the model of possible repairs for the unclean relation `Person`. U-clean relation `Person`^{c} is created by clustering algorithms $\mathcal{A}$ using parameters τ that is defined on the real interval $[0, 10]$ with uniform distributions. Relation `Person`^{c} captures all repairs of the base relations corresponding to possible parameter values. For example, if $\tau \in [1, 3]$, the resulting repair of Relation `Person` is equal to $\{\{\texttt{P1}, \texttt{P2}\}, \{\texttt{P3}, \texttt{P4}\}, \{\texttt{P5}\}, \{\texttt{P6}\}\}$, which is obtained using c-records in `Person`^{c} whose parameter settings contain the interval $[1, 3]$. Moreover, the U-clean relation allows for identifying the parameter settings of the clustering algorithm that lead to generating a specific cluster of records. For example, the cluster $\{\texttt{P1}, \texttt{P2}, \texttt{P5}\}$ is generated by algorithm $\mathcal{A}$ if the value of parameter τ belongs to the range $[3, 10)$.



ID	…	Income	C	P
CP1	…	31k	{P1,P2}	[1,3)
CP2	**…**	**40k**	**{P3,P4}**	**[0,10)**
CP3	**…**	**55k**	**{P5}**	**[0,3)**
CP4	…	30k	{P6}	[0,10)
CP5	**…**	**39k**	**{P1,P2,P5}**	**[3,10)**
CP6	…	30k	{P1}	[0,1)
CP7	…	32k	{P2}	[0,1)

Figure 3.19: An example of U-clean relation

Relational queries over U-clean relations are defined using the concept of *possible worlds semantics*, as shown in Figure 3.20. More specifically, queries are semantically answered against individual clean instances of the dirty database that are encoded in input U-clean relations, and the resulting answers are weighted by the probabilities of their originating repairs. For example, consider a selection query that reports persons with `Income` greater than 35k, considering all repairs encoded by Relation `Person`^{c} in Figure 3.19. One qualified record is `CP3`. However, such a record is valid only for repairs generated at the parameter settings $\tau \in [0, 3)$. Therefore, the probability that record `CP3` belongs to the query result is equivalent to the probability that τ is within $[0, 3)$, which is 0.3.

Sampling the Space of Possible Repairs

For any two tuples t_1, t_2 that violate an FD $X \rightarrow A$, the violation can be repaired by either changing $t_1[A]$ to be the value of $t_2[A]$ (or vice versa), or by modify an attribute $B \in X$ in either t_1 or t_2, so that $t_1[B] \neq t_2[B]$. Generalizing this observation, if a set of CleanCells does

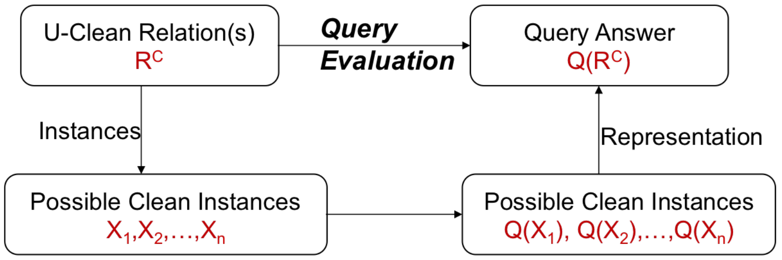


Figure 3.20: U-clean relation query model

not violate any FD in Σ, the consistency of CleanCells$\cup C$, for any cell C, can always be ensured by modifying C if necessary. To systematically determine whether a set of cells is clean, the equivalence classes $\mathcal{E}$ for that set of cells is built. An equivalence $eq \in \mathcal{E}$ denotes a subset of cells that should be equal according to Σ. Thus, to check if a set of cells is clean or not, it is sufficient to check if any two cells in an equivalent class $eq \in \mathcal{E}$ indeed have the same value.

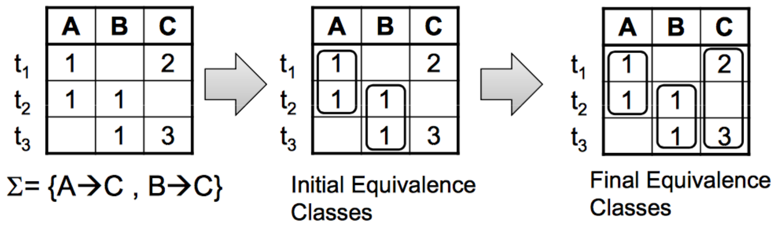

$\Sigma = \{A \rightarrow C, B \rightarrow C\}$
Initial Equivalence Classes
Final Equivalence Classes

Figure 3.21: An example of checking whether a set of cells is clean

Example 3.12. For example, in order to determine if the set of six cells $t_1[A], t_1[C], t_2[A], t_2[B], t_3[B], t_3[C]$ in Figure 3.21 is clean or not, a set of equivalence classes $\mathcal{E}$ is built. Initially, $t_1[A], t_2[A]$ belong to the same equivalence class, since they have the same value, so do $t_2[B], t_3[B]$, $t_1[C]$ and $t_3[C]$ belong to their own equivalence classes. According to the FD $A \rightarrow C$, $t_1[C]$ and $t_2[C]$ should belong to the same equivalence class. According to the FD $B \rightarrow C$, $t_2[C]$ and $t_3[C]$ should belong to the same equivalence class. Thus, $t_1[C]$ and $t_3[C]$ end up in the

Algorithm 9 `Sampling FDs Repairs`

```
Input: Database instance $I$, a set of FDs $\Sigma$
Output: Possible repairs $I'$
1: $I' \leftarrow I$
2: CleanCells $\leftarrow \emptyset$
3: while CleanCells $\neq CIDs(I')$ do
4:   Insert a random cell $t[A] \in CIDs(I') \backslash$CleanCells to CleanCells,
     where $t \in I$ and $A \in R$
5:   Build the equivalence classes $\mathcal{E}$ of CleanCells according to $\Sigma$
6:   if CleanCells is not clean w.r.t. $\mathcal{E}$ then
7:     Build the equivalence classes $\mathcal{E}_p$ of CleanCells $\backslash t[A]$ according
       to $\Sigma$
8:     if $t[A]$ belongs to a non-singleton equivalence class in $\mathcal{E}_p$ then
9:       set $I'(t[A])$ to the value of other cells that are in the same
         equivalence class in $\mathcal{E}_p$
10:    else
11:      randomly set $I'(t[A])$ to one of the three alternatives: a ran-
         domly selected constant from $Dom(A)$, a randomly selected
         variable that appears in $I'$, or a new variable such that
         CleanCells is clean w.r.t. $\mathcal{E}$
12: return $I'$
```

same equivalence class. However, $t_1[C]$ and $t_3[C]$ have different values. Therefore, the set of six cells is not clean.

Algorithm 9 describes the procedure for generating repairs. Cells are inserted into CleanCells in random order. At each iteration, the algorithm checks whether CleanCells is clean or not by building the equivalence classes $\mathcal{E}$ according to Σ. If CleanCells is not clean, the last inserted Cell C is changed so that CleanCells is clean again.

An example of executing Algorithm 9 is shown in Figure 3.22. At each step, the cells that have been selected so far by the algorithm are shown. Equivalence classes are shown as rectangles. The cells $t_1[A], t_1[B], t_2[A]$ and $t_3[B]$ are added to *CleanCells* in step (a). They are all clean and do not need to change. The cell $t_2[B]$ is added to

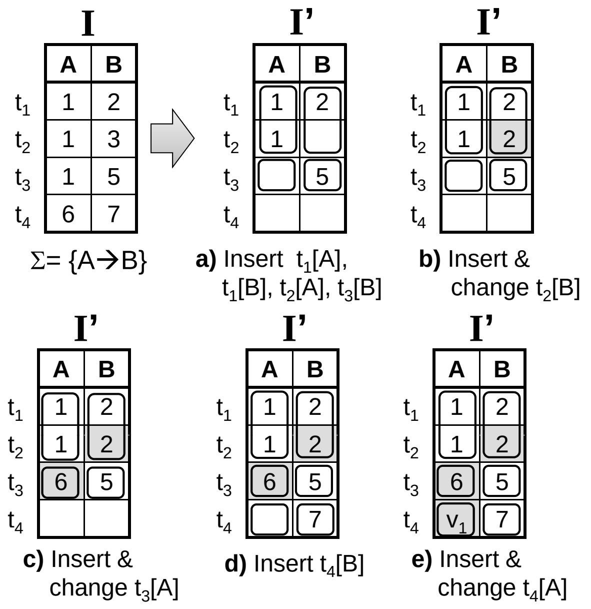


Figure 3.22: An example of executing Algorithm 9

CleanCells in step (b). Because the cells $t_1[B]$ and $t_2[B]$ belong to the same equivalence class, the value of $t_2[B]$ must be changed to the value of $t_1[B]$, which is 2. In step (c), the cell $t_3[A]$ is added to *CleanCells*. The value of $t_3[A]$ is changed to a randomly selected constant, namely, 6, to resolve the violation. We continue adding the remaining cells and modifying them as needed to make sure that *CleanCells* is clean after each insertion. Finally, the resulting instance I' represents a repair of I. Algorithm 9 is extended to sampling from the space of all cardinality-set-minimal repairs for CFDs [11].

4

Big Data Cleaning

With the advent of big data era, data cleaning has come more important and challenging than ever. Due to the sheer volume of generated data, and the fast velocity of arriving data, data cleaning activities need to be performed in a scalable and timely manner, and at the same time cope with the increasing variety of data sources. In this section, we discuss various algorithmic and systematic approaches in cleaning big data, including blocking for duplicate detection, sampling for data cleaning, incremental data cleaning, distributed data cleaning.

DeDuplicating Big Data

As discussed in Section 2.1.2, detecting duplicate records in a database of n records requires $O(n^2)$ comparisons for every pair of records. When the number of tuples n is large, duplicate detection is expensive. To avoid computing the similarity between all pairs of records, three methods are often employed, namely, *blocking*, *windowing*, and *canopy clustering*.

Blocking methods partition all records into disjoint blocks; only records within the same block are compared, while records residing in different blocks are considered non-duplicates [4, 14]. A simple way to perform blocking is to scan all the records and compute the hash value of a hash function for each record based on some attributes, commonly referred to as blocking keys. Those records with the same hash value are within the same block. Examples of blocking keys are first three characters of last name, and the concatenation of city, state, and zip attribute. Although blocking can substantially increase the comparison efficiency, it can result in many false negatives when two duplicate records do not agree on the blocking key, and thus reside in two different blocks. One way to alleviate such problem is to perform the deduplication algorithm in multiple passes, using a different blocking criteria for each pass; another approach is to build a complex blocking function that is a combination of multiple blocking criteria [98, 88].

Windowing methods, also known as sorted neighborhood approaches, sort all the records according to some keys, and then slide a fixed sized window across the sorted records; only records within the same window are compared [68, 69]. The windowing methods rely on the assumption that duplicate records are close to each other in the sorted list. It can be seen that the effectiveness of the windowing methods is highly dependent on the sorting keys. Often times, a single key is not sufficient to place all duplicate record pairs in the same window. Thus, similar to blocking methods, multiple passes based on different ordering keys can be employed.

Canopy clustering [85] is another approach to speed up the pair-wise comparison of records. Canopy clustering places records into overlapping clusters, referred to as canopies, using an inexpensive comparison metric. Canopies are different from blocks in that two canopies can contain the same records, while two blocks must be disjoint. After all records are grouped into canopies, the pair-wise comparison is performed within each canopy using a more expensive similarity metric that leads to better deduplication quality. Canopy clustering hinges on the existence of a cheap similarity function that serves as a "quick-and-dirty" way to check for a more expensive similarity function. For instance, two strings with length difference more than 10 cannot have their edit distance less than 10. Thus, the length comparison can serve

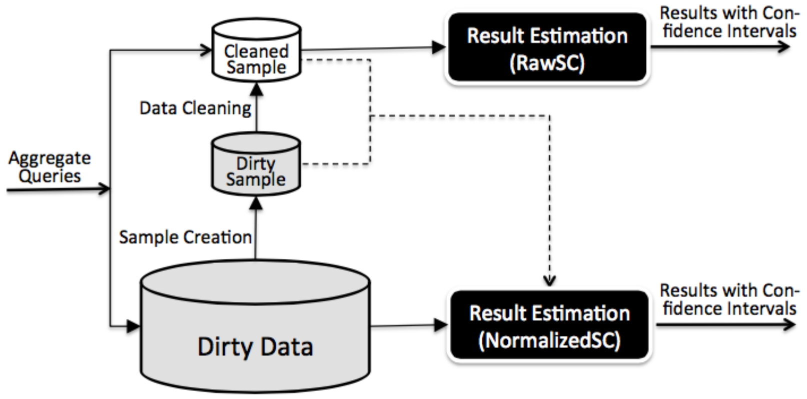

Aggregate Queries
Cleaned Sample
Data Cleaning
Dirty Sample
Sample Creation
Dirty Data
Result Estimation (RawSC)
Results with Confidence Intervals
Result Estimation (NormalizedSC)
Results with Confidence Intervals

Figure 4.1: SampleClean Framework.

as the cheap similarity function for the more expensive edit distance computation.

Sampling for Data Cleaning

Despite all the developments of data cleaning techniques, cleaning large sized data is a challenging task. Particularly, in order to obtain reliable data cleaning results, humans often need to be involved, which is highly inefficient for large dataset. SampleClean [115] applies data cleaning to a relatively small sample of the data and uses the cleaned sample for answering queries.

Figure 4.1 illustrates the SampleClean framework. SampleClean first generates a sample of dirty data, and then invokes data cleaning techniques that possibly involve crowds and experts to clean the dirty sample. SampleClean then uses the cleaned sample to answer aggregate queries, and gives unbiased estimates with confidence intervals, meaning that in expectation the estimates are equal to the query results if the entire dataset was first cleaned, and then used to answer the query. SampleClean provides two types of estimates, namely, RawSC and NormalizedSC. RawSC directly estimates the true query result based on the cleaned sample, and NormalizedSC uses the difference between the cleaned sample and the dirty sample to correct the error in a query result over the entire dirty data. To obtain unbiased estimates, SampleClean has to account for duplicates when sampling. For example, consider a query asking for the average number of citations of all papers published every year, grouping by year. Assume that the table has duplicates, and papers with a higher citation count tend to have more duplicates. The more duplicates a paper has, the more likely that paper is sampled, leading to over-estimated average citation count per year when random sampling is used. Therefore, SampleClean samples a tuple that has more duplicates with a smaller probability. Specifically, a tuple with no duplicates is c times more likely to be included in the sample than a tuple with c duplicates.

Incremental Data Cleaning

Data may arrive constantly, and the data quality rules might also evolve over time. Cleaning one snapshot of the data with respect to one set of data quality rules is already a challenging and time-consuming task, applying data cleaning techniques every time as data or rules change is impractical. Therefore, incremental data cleaning techniques are valuable in dealing with constantly changing data and rules. Whang and Garcia-Molina [120] study the problem of entity resolution (ER) with evolving rules, that is the logic of comparing two records are evolving over time. For example, an entity resolution logic, saying that *two persons match if their names are similar*, might produce inaccurate ER results as more persons are added to the database. Therefore, the logic should be changed to *two persons match if their names are similar and the zip codes of the two persons are the same.* Since the new logic is "stricter" than the old logic, the ER results using the new logic could be obtained from the previous ER results using the old logic. Gruenheid et al. [63] study the problem of ER with changing data, including insertions, deletions, and value modifications. Instead of running an ER algorithm on the entire set of records, two algorithms are proposed that only need to apply the ER algorithm on a subset of the records that are affected by the changed records. Since the subset of the records that are affected by the changed records could also be large, a greedy algorithm is also proposed that merges, splits clusters connected to the changed records, and moves records between those clusters. Continues data cleaning [113] discussed in Section 3.1.3 study the problem of ICs based data cleaning, when the data and the ICs are both evolving.

Distributed Data Cleaning

Recently, cleaning large datasets was implemented using a distributed framework, such as MapReduce [35], or Spark [127]. Dedoop [79, 78] specializes in performing ER using MapReduce. Despite the use of blocking techniques, ER remains a costly process. A straightforward implementation using MapReduce would be to distribute every block to one reducer machine. However, such a basic implementation is susceptible to severe load balancing issues due to skewed block sizes. Dedoop proposes two strategies for load balancing: BlockSplit, and PairRange. BlockSplit assigns every small block into one reducer machine if this does not violate load balancing constraint, and splits large blocks into smaller chunks to enable parallel processing. PairRange, on the other hand, evenly distributes all the tuple pairs in all blocks to all available reducer machines. There are also techniques that balance the workload of comparing all record pairs by exploiting the properties of the similarity measure used for comparing two record [97, 110].

BigDansing [76] is a distributed data cleaning system that runs on top of common data processing platforms, such as DBMS, or MapReduce frameworks. It takes as input a set of ICs or user defined data quality rules, and runs these rules through a series of transformations that enable distributed computations and optimizations. For example, to detect violations for an FD $X \rightarrow Y$, instead of enumerating all the tuple pairs in the database, BigDansing groups all the tuples based on their values for Attribute X, and only enumerates all tuple pairs that have the same value for X, which is similar to blocking in duplicate detection. BigDansing also includes other optimizations for violation detection, such as, scoping, which limits the amount of data that needs to be treated, and shared scan, which identifies common operations for detecting violation for multiple rules. BigDansing also utilizes distributed framework for data repairing, for example, every connected component in the conflict hypergraph (cf. Section 3.1.1) is being processed by an independent machine.

The data cleaning techniques we've discussed so far assume that the data resides in a centralized database. In practice, a relation can be fragmented and distributed across different sites, which make the data cleaning problem harder. In order to detect violations, data might need to be shipped from one site to another. For example, if Tuple t_1 and Tuple t_8 in Example 1.1 reside at two different sites, then some data shipment has to happen to ensure that both t_1 and t_8 appear together at least at one site in order to be able to detect the violation consisting of four cells $\{t_1[\text{ZIP}],\ t_8[\text{ZIP}],\ t_1[\text{ST}],\ t_8[\text{ST}]\}$. Fan et al. [46] study the problem of detecting violations for CFDs when the data is either horizontally partitioned, or vertically partitioned, across multiple sites. The objective now is to minimize data shipment. The work is further extended to incrementally detect violations when the distributed data is updated [52].

5

Conclusion

In this paper we shed some light on some of the foundational aspects and trends in data cleaning efforts. We primarily focused on the two phases of data cleaning: error detection and repairing.

For anomaly detection, we provided a classification for data repairing techniques based on What, How and Where to detect the errors:

- *What* Surveys many integrity constraints languages proposed to capture data errors, along with algorithms for their automatic discovery, as well as major steps involved for detecting duplicate records.

- *How* Discusses how humans may be helpful for detecting anomalies.

- *Where* Presents multiple errors propagation approaches that allow for detecting errors in source data when anomalies are detected at higher levels of the processing stack.

For data repairing, we also provided a classification for data repairing techniques based on What, How and Where to repair the data:

- *What* Reflects the different options repairing techniques take when deciding on the repairing target (data, rules, or both). Data only repairing further depends on the different types of errors that drive the repairing process, and whether it is a single error type (*e.g.*, FD violations) or a set of heterogeneous types of errors that need to be repaired holistically.

- *How* Highlights the different repairing methodologies followed by most repairing algorithms, mainly, whether and how humans and machines are used to figure out the correct updates to the erroneous data. This dimension also highlights the different objective functions adopted by automatic repairing algorithms, for example, cardinality-minimal repairs.

- *Where* Contrasts in-situ repairing to model-based repairing, where models are built to describe possible ways to repair the data, instead of repairing it in place.

Data quality and data cleaning are becoming more important than ever, with direct and timely needs in the Big Data era. Data cleaning is the first line of defense in extracting value from the huge amounts of heterogeneous, incomplete, and continuously growing data sets. We envision multiple future work directions, we list some of them in the following:

- *Anomaly Detection.* While we have discussed several ways to detect anomalies in the data, many data errors may still remain undetected. One direction is to devise more expressive integrity constraint languages that allow data owners to easily specify data quality rules and to effectively involve human experts in anomaly detection.

- *Master data curation.* To perform reliable data repairing, master data often needs to be referenced. However, existing master data sources, such as knowledge bases, often cannot provide a comprehensive coverage for the data to be repaired. Automatic creation and maintenance of relevant master and authoritative data catalogs are essential tasks in enabling high-quality repairs.

- *Human involved data repairing.* Although much research has been done about involving humans to perform data deduplication, involving humans in other data cleaning tasks, such as repairing integrity constraint violations is yet to be explored.

- *Scalability.* Large volumes of data render most current techniques unusable in real settings. The obvious trade-off between accuracy and performance has to be taken more seriously in designing the next generation cleaning algorithms that take time and space budget into account. Example tools include sampling, and approximate cleaning algorithms, with clear approximation semantics that can be leveraged by analytics applications.

References

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

[2] F. N. Afrati and P. G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *Proceedings of the 12th International Conference on Database Theory*, pages 31–41, 2009.

[3] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):23, 2008.

[4] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 586–597, 2002.

[5] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 783–794, 2010.

[6] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 952–963, 2009.

[7] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 68–79, 1999.

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

[9] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *Proceedings of the VLDB Endowment*, 3(1-2):197–207, 2010.

[10] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *29th IEEE International Conference on Data Engineering*, pages 541–552, 2013.

[11] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. Sampling from repairs of conditional functional dependency violations. *The VLDB Journal*, 23(1):103–128, 2014.

[12] G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David. Modeling and querying possible repairs in duplicate detection. *Proceedings of the VLDB Endowment*, pages 598–609, 2009.

[13] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):5, 2007.

[14] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *6th International Conference on Data Mining*, pages 87–96, 2006.

[15] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 39–48. ACM, 2003.

[16] J. Bleiholder and F. Naumann. Data fusion. *ACM Computing Survey*, 41(1), 2008.

[17] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 143–154. ACM, 2005.

[18] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 746–755, 2007.

[19] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 445–456, 2014.

[20] S. Chaudhuri, B. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 327–338, 2007.

[21] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proceedings of the 21st International Conference on Data Engineering*, pages 865–876, 2005.

[22] F. Chiang and R. J. Miller. Discovering data quality rules. *Proceedings of the VLDB Endowment*, 1(1):1166–1177, 2008.

[23] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pages 446–457, 2011.

[24] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1):90–121, 2005.

[25] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proceedings of the VLDB Endowment*, 6(13):1498–1509, 2013.

[26] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *29th IEEE International Conference on Data Engineering*, pages 458–469, 2013.

[27] X. Chu, I. F. Ilyas, P. Papotti, and Y. Ye. Ruleminer: Data quality rules discovery. In *IEEE 30th International Conference on Data Engineering*, pages 1222–1225, 2014.

[28] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1247–1261, 2015.

[29] S. Clemens. 7 facts about data quality. *InsightSquared*, 2012.

[30] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *ACM SIGMOD Record*, volume 27, pages 201–212, 1998.

[31] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 315–326. VLDB Endowment, 2007.

[32] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 541–552, 2013.

[33] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning.* John Wiley & Sons, Inc., 2003.

[34] F. De Marchi, S. Lopes, and J.-M. Petit. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, 32(1):53–73, 2009.

[35] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[36] 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.

[37] T. Diallo, J.-M. Petit, and S. Servigne. Discovering editing rules for data cleaning. In *Proceedings of AQB conference*, page 40, 2012.

[38] A. Doan, Y. Lu, Y. Lee, and J. Han. Profile-based object matching for information integration. *IEEE Intelligent Systems*, 18(5):54–59, 2003.

[39] X. L. Dong and F. Naumann. Data fusion: resolving data conflicts for integration. *Proceedings of the VLDB Endowment*, 2(2):1654–1655, 2009.

[40] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[41] M. Elsner and E. Charniak. You talking to me? a corpus and algorithm for conversation disentanglement. In *Association for Computational Linguistics (ACL)*, pages 834–842, 2008.

[42] M. Elsner and W. Schudy. Bounding and comparing methods for correlation clustering beyond ilp. In *Proceedings of the Workshop on Integer Linear Programming for Natural Langauge Processing*, pages 19–27, 2009.

[43] G. Fan, W. Fan, and F. Geerts. Detecting errors in numeric attributes. In *Web-Age Information Management*, pages 125–137. Springer, 2014.

[44] W. Fan and F. Geerts. *Foundations of Data Quality Management.* Synthesis Lectures on Data Management. 2012.

[45] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):683–698, 2011.

[46] W. Fan, F. Geerts, S. Ma, and H. Müller. Detecting inconsistencies in distributed data. In *Proceedings of the 26th International Conference on Data Engineering*, pages 64–75, 2010.

[47] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *29th IEEE International Conference on Data Engineering*, pages 470–481, 2013.

[48] W. Fan, F. Geerts, N. Tang, and W. Yu. Conflict resolution with data currency and consistency. *Journal of Data and Information Quality*, 5(1-2):6:1–6:37, 2014.

[49] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *Proceedings of the VLDB Endowment*, 2(1):407–418, 2009.

[50] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *Proceedings of the VLDB Endowment*, 3(1-2):173–184, 2010.

[51] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 469–480. ACM, 2011.

[52] W. Fan, J. Li, N. Tang, and W. Yu. Incremental detection of inconsistencies in distributed data. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1367–1383, 2014.

[53] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases*, pages 371–380, 2001.

[54] H. Galhardas, A. Lopes, and E. Santos. Support for user involvement in data cleaning. In *Data Warehousing and Knowledge Discovery - 13th International Conference, DaWaK 2011*, pages 136–151, 2011.

[55] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The llunatic data-cleaning framework. *Proceedings of the VLDB Endowment*, 6(9):625–636, 2013.

[56] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and cleaning. In *IEEE 30th International Conference on Data Engineering*, pages 232–243, 2014.

[57] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks! LLUNATIC goes open source. *Proceedings of the VLDB Endowment*, 7(13):1565–1568, 2014.

[58] L. Getoor and A. Machanavajjhala. Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5(12):2018–2019, 2012.

[59] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 601–612, 2014.

[60] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *Proceedings of the VLDB Endowment*, 1(1):376–390, 2008.

[61] G. Grahne. *The Problem of Incomplete Information in Relational Databases*, volume 554 of *Lecture Notes in Computer Science*. 1991.

[62] S. Greco, C. Molinaro, and F. Spezzano. Incomplete data and data dependencies in relational databases. *Synthesis Lectures on Data Management*, 2012.

[63] A. Gruenheid, X. L. Dong, and D. Srivastava. Incremental record linkage. *Proceedings of the VLDB Endowment*, 7(9):697–708, 2014.

[64] P. J. Guo, S. Kandel, J. Hellerstein, and J. Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *ACM User Interface Software & Technology (UIST)*, 2011.

[65] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 805–810, 2005.

[66] J. Heer, J. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[67] J. M. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[68] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. *ACM SIGMOD Record*, 24(2):127–138, 1995.

[69] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[70] T. N. Herzog, F. J. Scheuren, and W. E. Winkler. *Data Quality and Record Linkage Techniques*. Springer Science & Business Media, 2007.

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

[72] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *31st IEEE International Conference on Data Engineering*, 2015.

[73] M. A. Jaro. Unimatch: A record linkage system: User's manual. *U.S. Bureau of the Census*, 1976.

[74] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Human Factors in Computing Systems (CHI)*, 2011.

[75] K. Kerr, T. Norris, and R. Stockdale. Data quality information and decision making: a healthcare case study. In *Proceedings of the 18th Australasian Conference on Information Systems Doctoral Consortium*, pages 5–7, 2007.

[76] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1215–1230, 2015.

[77] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *12th International Conference on Database Theory*, pages 53–62, 2009.

[78] L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012.

[79] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *IEEE 28th International Conference on Data Engineering*, pages 618–629, 2012.

[80] N. Koudas, A. Saha, D. Srivastava, and S. Venkatasubramanian. Metric functional dependencies. In *Proceedings of the 25th International Conference on Data Engineering*, pages 1275–1278, 2009.

[81] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 802–803, 2006.

[82] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady*, volume 10, page 707, 1966.

[83] A. Lopatenko and L. E. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *11th International Conference on Database Theory*, pages 179–193, 2007.

[84] S. Ma, W. Fan, and L. Bravo. Extending inclusion dependencies with conditions. *Theoretical Computer Science*, 515:64–95, 2014.

[85] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, 2000.

[86] A. McCallum and B. Wellner. Conditional models of identity uncertainty with application to noun coreference. In *Advances in Neural Information Processing Systems 17 Neural Information Processing Systems*, pages 905–912, 2004.

[87] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 505–516, 2011.

[88] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 440, 2006.

[89] A. E. Monge and C. Elkan. The field matching problem: Algorithms and applications. In *Knowledge Discovery and Data Mining*, pages 267–270, 1996.

[90] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection.* Synthesis Lectures on Data Management. 2010.

[91] V. Ng and C. Cardie. Improving machine learning approaches to coreference resolution. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 104–111, 2002.

[92] T. Papenbrock, S. Kruse, J.-A. Quiané-Ruiz, and F. Naumann. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment*, 8(7):774–785, 2015.

[93] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

[94] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases*, pages 381–390, 2001.

[95] R. Russell. Index., Apr. 2 1918. US Patent 1,261,167.

[96] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 269–278, 2002.

[97] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *Proceedings of the VLDB Endowment*, 7(12):1059–1070, 2014.

[98] A. D. Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *21st ACM International Conference on Information and Knowledge Management*, pages 1055–1064, 2012.

[99] G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *International Conference on Machine Learning*, pages 839–846, 2000.

[100] B. Settles. Active learning literature survey. *University of Wisconsin, Madison*, 52:55–66, 2010.

[101] P. Singla and P. Domingos. Entity resolution with markov logic. In *2013 IEEE 13th International Conference on Data Mining*, pages 572–582, 2006.

[102] S. Song and L. Chen. Discovering matching dependencies. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 1421–1424, 2009.

[103] W. M. Soon, H. T. Ng, and D. C. Y. Lim. A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544, 2001.

[104] B. Steuart and P. I. Staff. *The Daitch-Mokotoff Soundex Reference Guide*. Heritage Quest, 1994.

[105] 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, 6th Biennial Conference on Innovative Data Systems Research*, 2013.

[106] N. Swartz. Gartner warns firms of "dirty data". *Information Management Journal*, 41(3), 2007.

[107] R. Taft. *Name Search Techniques*. Special report (New York State Identification and Intelligence System). Bureau of Systems Development, New York State Identification and Intelligence System, 1970.

[108] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems*, 26(8):607–633, 2001.

[109] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *The Journal of Machine Learning Research*, 2:45–66, 2002.

[110] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[111] V. S. Verykios, A. K. Elmagarmid, and E. N. Houstis. Automating the approximate record-matching process. *Information Sciences*, 126(1):83–98, 2000.

[112] N. Vesdapunt, K. Bellare, and N. Dalvi. Crowdsourcing algorithms for entity resolution. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[113] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *IEEE 30th International Conference on Data Engineering*, pages 244–255, 2014.

[114] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.

[115] 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 *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 469–480, 2014.

[116] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 229–240, 2013.

[117] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 457–468. ACM, 2014.

[118] M. S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.

[119] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. Industry-scale duplicate detection. *Proceedings of the VLDB Endowment*, 1(2):1253–1264, 2008.

[120] S. E. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *Proceedings of the VLDB Endowment*, 3(1-2):1326–1337, 2010.

[121] W. E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. *Proceedings of the Section on Survey Research*, 1990.

[122] W. E. Winkler. The state of record linkage and current research problems. In *Statistical Research Division, U.S. Census Bureau*, 1999.

[123] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proceedings of the VLDB Endowment*, 6(8):553–564, 2013.

[124] E. Wu, S. Madden, and M. Stonebraker. A demonstration of dbwipes: clean as you query. *Proceedings of the VLDB Endowment*, 5(12):1894–1897, 2012.

[125] C. M. Wyss, C. Giannella, and E. L. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 101–110, 2001.

[126] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *Proceedings of the VLDB Endowment*, 4(5):279–289, 2011.

[127] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, volume 10, page 10, 2010.