Scalable and Holistic Qualitative Data Cleaning

by

Xu Chu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	**Gerhard Weikum** Professor, Department of Databases and Information Systems Max-Planck Institute for Informatics
Supervisor(s):	**Ihab Ilyas** Professor, David R. Cheriton School of Computer Science University of Waterloo
Internal Members:	**Tamer Özsu** Professor, David R. Cheriton School of Computer Science University of Waterloo **Grant Weddell** Associate Professor, David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	**Lukasz Golab** Associate Professor, Department of Management Sciences University of Waterloo

Author's Declaration

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. Data scientists reportedly spend 60% of their time in cleaning and organizing the data according to a survey published in Forbes in 2016. Therefore, we need effective and efficient techniques to reduce the human efforts in data cleaning.

Data cleaning activities usually consist of two phases: error detection and error repair. Error detection techniques can be generally classified as either quantitative or qualitative. Quantitative error detection techniques often involve statistical and machine learning methods to identify abnormal behaviors and errors. Quantitative error detection techniques have been mostly studied in the context of outlier detection. On the other hand, qualitative error detection techniques rely on descriptive approaches to specify patterns or constraints of a legal data instance. One common way of specifying those patterns or constraints is by using data quality rules expressed in some integrity constraint languages; and errors are captured by identifying violations of the specified rules. This dissertation focuses on tackling the challenges associated with detecting and repairing qualitative errors.

To clean a dirty dataset using rule-based qualitative data cleaning techniques, we first need to design data quality rules that reflect the semantics of the data. Since obtaining data quality rules by consulting domain experts is usually a time-consuming processing, we need automatic techniques to discover them. We show how to mine data quality rules expressed in the formalism of denial constraints (DCs). We choose DCs as the formal integrity constraint language for capturing data quality rules because it is able to capture many real-life data quality rules, and at the same time, it allows for efficient discovery algorithm.

Since error detection often requires a tuple pairwise comparison, a quadratic complexity that is expensive for a large dataset, we present a distribution strategy that distributes the error detection workload to a cluster of machines in a parallel shared-nothing computing environment. Our proposed distribution strategy aims at minimizing, across all machines, the maximum computation cost and the maximum communication cost, which are the two main types of cost one needs to consider in a shared-nothing environment.

In repairing qualitative errors, we propose a holistic data cleaning technique, which accumulates evidences from a broad spectrum of data quality rules, and suggests possible data updates in a holistic manner. Compared with previous piece-meal data repairing approaches, the holistic approach produces data updates with higher accuracy because it realizes the interactions between different errors using one representation, and aims at generating data updates that can fix as many errors as possible.

Acknowledgements

First, I would to offer my most sincere gratitude to my advisor, Prof. Ihab Ilyas. Throughout my PhD study, Prof. Ilyas has always been my inspiration. He taught me how to develop research ideas, how to write research papers, and how to give academic talks. He offered tremendous support for my academic job search. I shall be indebted for everything I have learned from him on both academic levels and personal levels. As I pursue my academic career, Ihab will always be my advisor, my mentor, and my dear friend.

I also would like to thank all the collaborators I have worked with over the years: Paolo Papotti, Yeye He, Sam Madden, Surajit Chaudhuri, Peter Bailis, and many others. I would not have had a successful PhD without them.

I am deeply grateful to my thesis committee members, Prof. Tamer Özsu, Prof. Grant Weddell, Prof. Lukasz Golab, and Prof. Gerhard Weikum for the discussions and advices throughout my PhD career, and for taking the time to read and to offer suggestions to improve this thesis.

I would like to thank my parents, Xiaoping Chu and Jiying Wang, for their continuous support and endless love. They are willing to give me everything and expect nothing in return.

Finally, this dissertation would not have been possible without my lovely wife, Jianmei Shi. She is always there to share joys with me in moments of happiness, and she is always there to comfort me in moments of sadness.

Dedication

This thesis is dedicated to my beloved wife Jianmei, and my wonderful parents.

Table of Contents

List of Tables

List of Figures

Chapter 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 [40], 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 [72]. Gartner predicted that more than 25% of critical data in the world's top companies is flawed [89]. Poor data across businesses and the government costs the U.S. economy $3.1 trillion a year, according to a report by InsightSquared [34]. 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]. According to a 2016 survey of about 80 data scientists conducted by CrowdFlower, which was published in Forbes [2], data scientists spend 60% of their time in cleaning and organizing the data, and 57% of data scientists regard cleaning and organizing data as the least enjoyable part of their work. Therefore, we need effective and efficient techniques to reduce the human efforts in data cleaning.

Data cleaning activities usually consist of two phases: error detection and error repair. Error detection techniques can generally be classified as either quantitative or qualita-

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

[2] https://goo.gl/aPzGO5

tive. Quantitative error detection techniques often involve statistical and machine learning methods to identify abnormal behaviors and errors [64] (e.g., *"an employee salary that is three standard deviation away from the mean salary is abnormal"*). Quantitative error detection techniques have been mostly studied in the context of outlier detection [6]. On the other hand, qualitative error detection techniques rely on descriptive approaches to specify patterns or constraints of a legal data instance. One common way of specifying those patterns or constraints is by using data quality rules expressed in some integrity constraint languages. Rule-based qualitative error detection techniques capture errors by identifying violations of specified data quality rules. For example, a data quality rule of a legal employee instance is *"two persons with the same zip code live in the same state"*. If we identify two employees in a data instance, such that they have the same zip code but different state, we are certain that at least one of the data values of zip code and state for the two employees is erroneous.

The error repair phase of data cleaning is performed either by applying data transformation scripts, which are usually generated according to the process used for error detection, or by involving human experts in a principled manner, or by a combination of both. This dissertation focuses on tackling the challenges associated with detecting and repairing qualitative errors.

1.1 Qualitative Data Cleaning Workflow

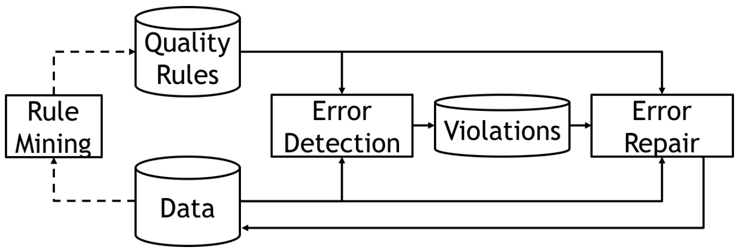


Figure 1.1: Qualitative data cleaning workflow with an optional rule mining step, the error detection step, and the error repair step

TID	FN	LN	LVL	ZIP	ST	SAL
t_1	Anne	Nash	5	10001	NM	110,000
t_2	Mark	White	6	87101	NM	80,000
t_3	Jane	Lee	4	10001	NY	75,000

Table 1.1: An example employee table

Figure 1.1 shows a typical workflow of qualitative data cleaning. In order to clean a dirty dataset using qualitative data cleaning techniques, we need to define data quality rules that reflect the semantics of the data. One way to obtain data quality rules is by consulting domain experts, which requires a lot of domain expertise and is usually a time-consuming processing. An alternative approach is to design an algorithm to automatically discover data quality rules. Given a dirty dataset and the associated data quality rules, the error detection step detects violations of the specified rules in the data. A violation is *minimal set of cells in a dataset that cannot coexist together*. Finally, given the violations and the rules that generate those violations, the error repair step produces data updates, which are applied to the dirty dataset. The error detection and the error repair loop goes on until the data conforms to the data quality rules, namely, there is no violations in the data.

Example 1: Consider Table 1.1 that contains employee records in a company. Every tuple specifies a person in a company with her id (GID), name (FN, LN), level (LVL), zip code (ZIP), state (ST), and salary (SAL). Suppose two data quality rules hold for this table. The first rule states that, if two employees have same zip code, they must have the same state. The second rule states that among employees working in the same state, a higher level employee cannot earn less salary than a lower level employee.

Given these two data quality rules, the error detection step detects two violations. The first violation consists of four cells $\{t_1[ZIP], t_1[ST], t_3[ZIP], t_3[ST]\}$, which together violate the first data quality rules. The second violation consists of six cells $\{t_1[ROLE], t_1[ST], t_1[SAL], t_2[ROLE], t_2[ST], t_2[SAL]\}$, which together violate the second data quality rule. The data repair step takes the violations and produces an update that changes $t_1[ST]$ from "NM" to "NY", and the new data now has no violation with respect to the two rules.

□

1.2 Challenges in Qualitative Data Cleaning

In this section, we discuss in detail the challenges associated with every one of the three steps of qualitative data cleaning workflow.

- **Rule Mining.** Since designing data quality rules through consultation with domain experts is an expensive and time-consuming process, automatically mining data quality rules is an appealing alternative. Obtaining data quality rules through mining algorithms incurs multiple challenges:

  (1) In order to mine data quality rules, we need a formal language to capture the space of rules. Integrity constraints (ICs), such as domain constraints, functional dependencies (FDs) [73], and their recent extension conditional functional dependencies (CFDs) [22], provide formal languages to capture data quality rules. The question is what IC language should we use to capture data quality rules? The more expressive the language is, the more rules we can potentially capture. However, with increasing expressiveness, it usually becomes harder to mine for rules due to a larger search space. Therefore, we need to identify an IC language that strikes a balance between the expressive power of the language, and the complexity of the algorithm to mine for rules expressed in that language.

  (2) We aim at mining valid ICs from a possibly dirty data instance, and use the mined constraints to detect data errors in the dirty data. Therefore, the mining algorithm has to be able to identify correct constraints in spite of potential violations of those constraints in the dataset.

  (3) Since the quality of ICs is crucial for data quality, discovered ICs need to be verified by domain experts for their validity before using them for data cleaning. Model discovery algorithms usually suffer from the problem of overfitting [20]; ICs

found on the input instance I of schema R may not hold on future data of R. Due to the potentially large number of discovered ICs, we need to assist the users in determining their validities.

- **Error Detection.** Detecting violations in a dataset with respect to a data quality rule usually requires enumerating combinations of tuples in the data, and checking whether a particular combination is in violation of the rule. For example, detecting violation with respect to the rule in Example 1 requires us to enumerate all pairs of tuples, a quadratic complexity that can be very expensive for a large dataset. Therefore, we need scalable techniques to detect violations efficiently. Big data often resides on a cluster of machines interconnected by a fast network, commonly referred to as a "data lake". Therefore, it is natural to leverage this scale-out environment to develop efficient data distribution strategies that parallelize error detection. Multiple challenges need to be addressed to perform distributed error detection:

  (1) Unlike centralized settings, where the dominating cost is almost always the computation cost of enumerating all tuple pairs, multiple factors contribute to the elapsed time in a distributed computing environment, including network transfer time, local disk I/O time, and CPU time. These costs also vary across different deployments.

  (2) In a distributed setting, data skew can be an important performance factor [12,41]. Therefore, loading-balancing is needed to allow every machine to perform a roughly equal amount of work in order to avoid situations where some machines take much longer than others to finish, a scenario that greatly affects the overall running time.

- **Error Repair.** Error repair aims at producing updates to the dataset to resolve the violations. While there is only one correct way to update the data, there might be multiple possible updates to resolve a violation in the lack of the ground truth. In Example 1, we only showed one way of resolving the violation consisting of four cells $\{t_1[ZIP], t_1[ST], t_3[ZIP], t_3[ST]\}$. There are actually many different ways of resolving it, for example, by updating $t_3[ST]$ from NY to NM, by updating $t_1[ZIP]$ from 10001 to 10002, or by updating $t_1[ZIP]$ from 10001 to 10003. Identifying the correct update among many possibilities has multiple challenges:

  (1) The error repair techniques need to identify and focus on updating cells that

are most likely to be erroneous among cells participating in violations, as not all of them are actually errors. We need techniques to accumulate different signals in determining the actual erroneous cells.

(2) Even after we have determined a particular cell to be an error, there still might be many possible ways to update it. Therefore, we need methods to collect repair requirements and distinguish which updates are more likely to be the correct updates.

1.3 Contributions and Outline

In this dissertation, we make multiple contributions that address the challenges associated with the three steps of qualitative data cleaning.

- **Rule Mining.** We propose an efficient algorithm to discover data quality rules automatically from a possibly dirty dataset, and we also present ranking mechanisms to assist the users in verifying the discovered rules [30, 32]. Specifically, we make the following contributions:

  (1) We propose to use *denial constraints* [11], a universally quantified first order logic formalism, as the formal language to capture a wide range of data quality rules. We show that denial constraints are expressive — they subsume many previously proposed integrity constraints, including FDs and CFDs [22]. We propose the FASTDC algorithm for discovering DCs [30]. FASTDC avoids enumerating the exponential number of candidate DCs by transforming the DC discovery problem to the problem of finding set covers for a data structured called *evidence sets* built from the input dataset.

  (2) In order to discover valid DCs in spite of errors in the input, we propose a notion of approximate DCs, which are candidate DCs that have a small number of violations in the input dataset. We show that an approximate set cover for the evidence sets corresponds to an approximate DC.

  (3) We propose a novel scoring function, which combines *succinctness* and *coverage* measures of discovered DCs in order to enable their ranking and pruning based on

thresholds, and thus reducing the cognitive burden for human verification of discovered DCs.

- **Error Detection.** To tackle the scalability challenges for error detection, we study the problem of how to distribute the tuple pairwise comparison in a distributed shared-nothing computing environment. Tuple pairwise comparison is commonly found operation in detecting errors. For example, detecting duplicate records requires computing similarities or running a classifier for every tuple pair; detecting violations of an IC involving two tuples, such as the FD in Example 1, requires checking whether the IC is violated for every tuple pair. We study the problem of distributing tuple pairwise comparison in the context of detecting duplicate records [29]. However, our proposed distribution strategy applies to any error detection task that requires tuple pairwise comparison. We propose a distribution strategy with optimality guarantees for distributed data deduplication in a shared-nothing environment. Specifically, we make the following contributions:

  (1) We introduce a cost model that considers both the communication cost and the computation cost, two main types of cost in a shared-nothing environment. We aim at minimizing the maximum number of input tuples any machine receives (X), and the maximum number of tuple pair comparisons any machine performs (Y). We provide a lower bound analysis for X and Y that is independent of the actual dominating cost in a cluster.

  (2) We propose a distribution strategy for distributing the workload of tuple pairwise comparison evenly to a cluster of machines. Both X and Y of our proposed strategy are guaranteed to be within a small constant factor from the lower bounds for X and Y.

  (3)Blocking techniques are often employed to avoid comparing all tuple pairs [9, 19, 65]. They first partition all records into blocks and then only records within the same block are compared. We show how our distribution strategy can be adapted to evenly distribute the data deduplication workload when blocking techniques are used.

- **Error Repair.** Since there might be many possible updates to resolve the detected violations, we propose a holistic data cleaning technique, which accumulates evidences from a broad spectrum of data quality rules, and suggests possible data updates in a holistic manner [31]. Compared with previous piece-meal data repairing approaches, the holistic approach produces data updates with higher accuracy because it has on a global view of different violations on the data. Specifically, we make the following contributions:

(1) Regardless of the data quality rules that generate the violations, we compile all detected violations onto one data structure called the *conflict hypergraph*, where a vertex corresponds to a cell, and a hyperedge corresponds to a violation. Intuitively, a cell is more likely to be erroneous if it participates in multiple violations. Therefore, we compute a minimal vertex cover for the conflict hypergraph, and we focus on coming up with updates for cells in the minimal vertex cover.

(2) For a cell in the minimal vertex cover, we collect the necessary repair expressions that need to satisfied in order to resolve the violations that cell is involved in. We aim at producing an update for that cell that resolves as many violations as possible.

The remainder of the dissertation is organized as follows. In Chapter 2, we discuss background and related work. In Chapter 3, we present our approach for discovering data quality rules expressed in the formalism of denial constraints automatically. In Chapter 4, we describe our distribution strategy to distribute the workload of detecting duplicate records. In Chapter 5, we introduce how holistic data cleaning idea is able to suggest more accurate data repairs than piece-meal data repairing approaches. Finally, in Chapter 6, we conclude the dissertation with final remarks and directions for future work.

Chapter 2

Background and Related Work

There are various surveys and books on data quality and data cleaning. Rahm and Do [85] 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; Grahne [59] focuses on the effect of incompleteness data on query answering, and the use of a Chase procedure for dealing with incomplete data [60]; Hellerstein [64] focuses on cleaning quantitative data, such as integers and floating points, using mainly statistical outlier detection techniques. Bertossi [14] provides complexity results for repairing inconsistent data, and performing consistent query answering on inconsistent data; Fan and Geerts [45] discuss the use of data quality rules in data consistency, data currency, and data completeness, how different aspects of data quality issues might interact; Dasu and Johnson [37] summarize how techniques in exploratory data mining can be integrated with data quality management.

In this chapter, we discuss backgrounds and related works of qualitative data cleaning. In Section 2.1, we review integrity constraints proposed in the literature for expressing data quality rules. In Section 2.2, we discuss general approaches used for discovering integrity constraints, and show two example algorithms for discovering functional dependencies. In Section 2.3, we present a taxonomy of different qualitative error detection techniques. In Section 2.4, we show a taxonomy of different error repair techniques. The two taxonomies were first presented in our survey [68]. We use the taxonomies to discuss related works of qualitative data cleaning, and show how our proposals in this dissertation fit in and compare with the literature.

2.1 Integrity Constraints

Integrity constraints (ICs), which are usually declared as part of the database schema, have been increasingly used to express data quality rules [46]. Data errors are captured by identifying violations of ICs. In the following, we review the most commonly used IC languages for expressing data quality rules in the literature.

2.1.1 Functional Dependencies

Functional dependencies (FDs) were originally proposed for database normalization [3]. However, there has been a recent interest in using FDs for detecting and repairing data errors [15, 21, 73].

Definition 1.1: Consider a relational schema R. A functional dependency (FD) φ is defined as $X \rightarrow Y$, where $X \subseteq R$ and $Y \subseteq R$. 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) of the FD, and Y the right hand side (RHS) and the FD. The first data quality rule in Example 1 can be expressed as an FD: $ZIP \rightarrow ST$.

2.1.2 Conditional Functional Dependencies

Conditional functional dependencies (CFDs), an extension of FDs, are capable of capturing FDs that hold partially on the data [22].

Definition 1.2: A CFD φ on R is a pair $(R : X \rightarrow Y, T_p)$, where:

- $X, Y \subset R$;
- $X \rightarrow Y$ 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 $t_\alpha[Y] = t_\beta[Y] \approx t_p[Y]$.

$\square$

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 $t_p[Y]$. 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".

ST	LVL	SAL
NJ	-	-
GA	-	-
GA	7	120,000

Figure 2.1: T_p for the CFD $(\{\text{ST, LVL }\} \rightarrow \{\text{SAL}\}, T_p)$

Example 2: Consider again Table 1.1 that contains employee records in a company. Suppose the FD that an employee's level determines her salary, i.e., $LVL \rightarrow SAL$, is not true for all employees in the company, however, it is true for employees in the state of NJ, or in the state of GA. In addition, it holds that an employee of Level 7 in the state of

GA has salary 120K. The aforementioned data quality rules can be expressed as a CFD $(\{\text{ST, LVL }\} \rightarrow \{\text{SAL}\}, T_p)$ with T_p shown in Figure 2.1.

□

2.1.3 Denial Constraints

As powerful as CFDs are, they are still not capable of capturing many real life data quality rules, such as rules include inequality comparisons. Denial constraints (DCs) [11], a universally quantified first order logic formalism, which subsume FDs and CFDs, can express the aforementioned rules.

Definition 1.3: 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 $\theta \in \{=, <, >, \neq, \leq, \geq\}$. Note that we use $t_\alpha, t_\beta, t_\gamma, \ldots \in R$ to denote possible tuples in an database instance of schema R.

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 3: The FD $ZIP \rightarrow ST$ in Example 1 can be expressed as a DC: $\forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$

The second rule in Example 1 can also be expressed as a DC: $\forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.LVL > t_\beta.LVL \wedge t_\alpha.SAL < t_\beta.SAL)$

In addition, the CFD in Example 2 can be expressed using the following three DCs:

$$\forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = \text{NJ} \wedge t_\beta.ST = \text{NJ} \wedge t_\alpha.LVL = t_\beta.LVL \wedge t_\alpha.SAL \neq t_\beta.SAL)$$

$$\forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = \text{GA} \wedge t_\beta.ST = \text{GA} \wedge t_\alpha.LVL = t_\beta.LVL \wedge t_\alpha.SAL \neq t_\beta.SAL)$$

$$\forall t_\alpha \in R, \neg(t_\alpha.ST = \text{GA} \wedge t_\alpha.LVL = 7 \wedge t_\alpha.SAL \neq 120{,}000)$$

□

2.1.4 Other Constraint Types

Besides FDs, CFDs, and DCs, many other different types of ICs have been proposed for capturing different types of data errors. Inclusion Dependencies (INDs) [3] 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) [77] can be considered as special MDs defined on one table, to capture small variations in the data; Numeric functional dependencies (NFDs) [44] can capture interesting constraints involving numeric attributes, since NFDs allow arithmetic operations; Editing rules (eRs) [51] not only provides a way to detect errors, but also tells how to fix errors by referencing a master table; Fixing rules [95] precisely captures which attribute is wrong, and how to correct the error, when enough evidence is present; Sherlock Rules [70] annotate the correct and erroneous attributes, and precisely tell how to fix the errors by referencing master tables.

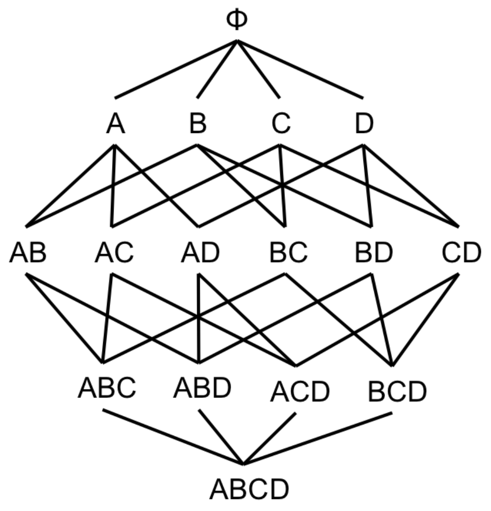
2.2 Discovery of Integrity Constraints

Since manually designing ICs is expensive and time-consuming, automatic IC discovery is an extremely useful functionality in bootstrapping the cleaning exercise. In order to discover valid ICs that are correct for a database schema, IC discovery algorithms usually first discover ICs that hold on the current input data instance, and then involve humans to verify the discovered ICs for validity. IC discovery algorithms generally fall into two categories: schema-driven approaches and instance-driven approaches [2, 68]. Schema-driven IC discovery approaches usually enumerate all candidates ICs generated based on the schema, and check whether each candidate IC holds on the data instance. On the other hand, instance-driven IC discovery approaches avoid enumerating all candidate ICs by usually building a special data structure using the data instance and searching directly for ICs that hold on the data instance based on that data structure. Schema-driven approaches are usually more sensitive to the size of the schema, while instance-driven approaches are usually more sensitive to the size of the instance.

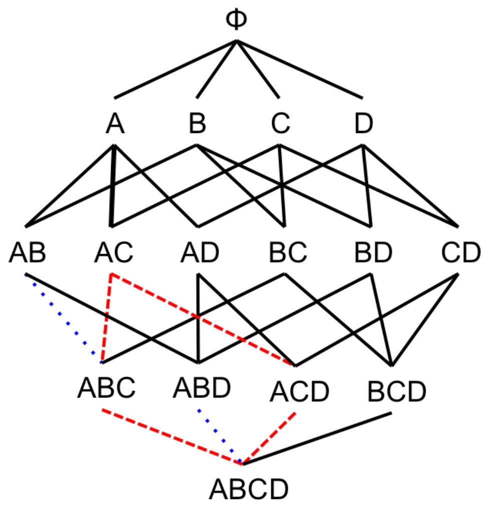
In this section, we use two example algorithms for discovery FDs to show the differences between these two approaches: TANE [67] as an example of a schema-driven approach, and FASTFD [98] as an example of an instance-driven approach. TANE adopts a level-wise candidate generation and pruning strategy and relies on a linear algorithm for checking whether a FD holds on the input instance. 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. We refer readers to our survey [68] for a more comprehensive coverage on discovery algorithms proposed for FDs, CFDs, DCs, and other types of ICs. Our algorithm FASTDC for discovering DCs in Chapter 3 is instance-driven algorithm similar to FASTFD. However, FASTDC uses different data structure and different pruning rules that are specific to DCs; in addition, FASTDC includes novel interestingness functions both to rank discovered DCs for user verification and to early stop the searching process.

2.2.1 Schema-Driven FD Discovery

An FD $\varphi : X \rightarrow A$ holds on the 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 no longer hold on I. 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



(a) Space of FDs



(b) Candidate FDs pruned if $A \to C$ holds

Figure 2.2: TANE

minimal nontrivial FDs with one attribute in the RHS that hold on I.

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 2.2(a) 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 $A \to C$.

Algorithm 1 describes TANE [67]. 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 $X \to A \in \Sigma$, then all FDs of the form $XY \to A$ are implied, and hence they can be pruned; (2) If $X \to A \in \Sigma$, then all FDs of the form $XAY \to B$ can be pruned. The reason is that if $XY \to B$ holds, then $XAY \to B$ is implied by $XY \to B$, which would be discovered earlier due to level-by-level traversal, and if $XY \to B$ does not hold, then $XAY \to B$ also does not hold due to $X \to A$; and (3) If X is a key, then any node containing X can be pruned.

Algorithm 1 TANE

```
Require: One relational instance $I$, schema $R$
Ensure: 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:     $l \leftarrow l + 1$
```

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 $\Pi_{XY} = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$. 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.

2.2.2 Data-Driven FD Discovery

FASTFD [98] is an instance-based FD discovery algorithm. We start by defining the difference set of two tuples t_1, t_2 as $D(t_1, t_2) = \{A \in R \mid t_1[A] \neq t_2[A]\}$. 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 attribute $A \in R$, the difference sets of I modulo A are $D_I^A = \{D - \{A\} \mid D \in D_I,\ \text{and}\ A \in D\}$. 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 4: 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

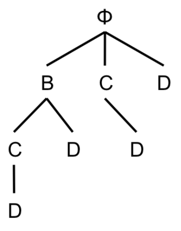
We have $D(t_1, t_2) = \{AD\}$, $D(t_1, t_3) = \{BC\}$, and $D(t_2, t_3) = \{ABCD\}$. Thus, $D_I = \{AD, BC, ABCD\}$, and $D_I^A = \{D, BCD\}$. Since $\{D\}$ is a minimal cover of D_I^A, we have $D \rightarrow A$. □

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 4.

Algorithm 2 FASTFD

```
Require: One relational instance $I$, schema $R$
Ensure: 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 $X \rightarrow A$ 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 2.3(a). 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



(a) Static ordering of attributes

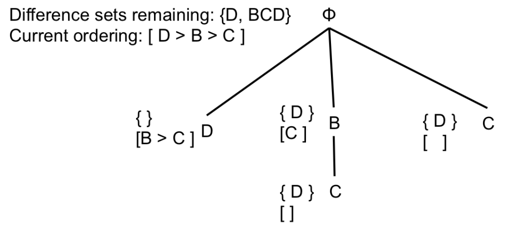

Difference sets remaining: {D, BCD}
Current ordering: [D > B > C]
Φ
{ }
[B > C] D
{ D }
[C] B
{ D } C
[ ]
{ D } C
[ ]
(b) Dynamic ordering of attributes

Figure 2.3: FASTFD

$\{BCD\}$, 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 2.3(b). 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.

2.3 Taxonomy of Qualitative Error Detection

In this section, we discuss related works in qualitative error detection, and show how our proposals in this dissertation fit in and compare with existing literature. Figure 2.4 depicts the classification we adopt to categorize the current error detection techniques. In the following, we briefly discuss our classification dimensions and we expand on them in the following subsections. The three adopted dimensions capture the three main questions involved in qualitative error detection.

- *Error Type (What to Detect?)* Error detection techniques can be classified according to which type of errors can be captured. In other words, what languages are used to describe patterns or constraints of a legal data instance. A large body of work uses integrity constraints (ICs), a fractional of first order logic, to capture data

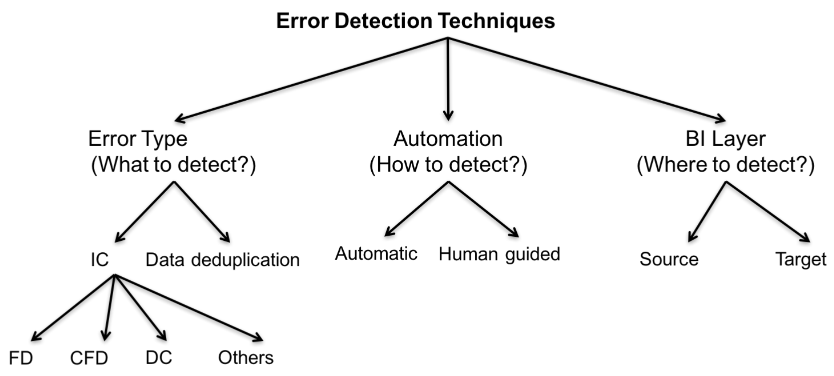

Error Detection Techniques
Error Type (What to detect?)
Automation (How to detect?)
BI Layer (Where to detect?)
IC
Data deduplication
Automatic
Human guided
Source
Target
FD
CFD
DC
Others

Figure 2.4: Classification of qualitative error detection techniques.

quality rules that the database should conform to, including functional dependencies (FDs) [73] and conditional functional dependencies [22]. While duplicate records can be considered a violation of an integrity constraint (key constraint), we recognize the large body of work that focuses on this problem and we treat it as a separate error type from other types of integrity constraints.

We propose to use denial constraints (DCs), a universally quantified first order logic formalism, as the formal language to capture a wide range of data quality rules, and we design an efficient algorithm to automatically mine DCs, as we will show in Chapter 3.

- *Automation (How to Detect?)* We classify proposed approaches according to whether and how humans are involved in the error 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.

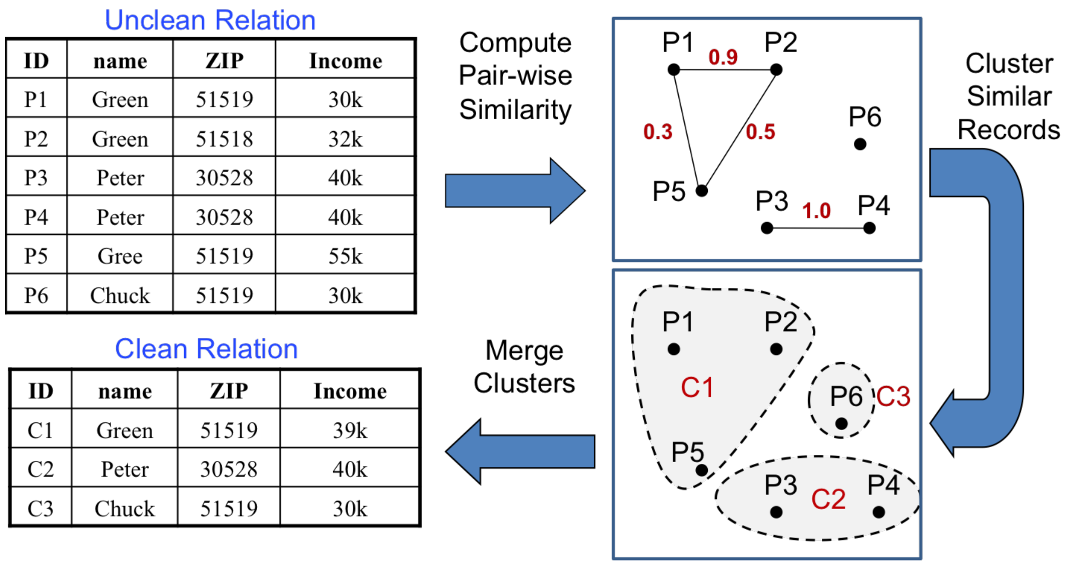
  We address the scalability challenge in the context of automatically detecting duplicate records, which discussed in Chapter 4. Data deduplication techniques usually require computing a similarity score of each tuple pair. For a dataset with n tuples, naïvely comparing every tuple pair requires $O(n^2)$ comparisons, a prohibitive cost when n is large.

	Error Type What		**Automation How**		**BI Layer Where**	
	IC	Data deduplication	Automatic	Human involved	Source	Target
FDs value modification [21]	✓		✓		✓	
Holistic data cleaning [31]	✓		✓		✓	
Distributed Data Deduplication [29]		✓	✓		✓	
CrowdER [93]		✓		✓	✓	
Corleone [56]		✓		✓	✓	
Causality Analysis [81]	✓			✓		✓
Scorpion [96]	✓			✓		✓
DBRx [23]	✓		✓			✓

Table 2.1: A sample of error detection techniques.

- *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 error 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 error detection techniques, which cover all categories of the proposed taxonomy. Our distributed data deduplication proposal targets duplicate records; it is an automatic error detection technique; and it detected the duplicates at the source data. Our holistic data cleaning proposal targets violations of ICs; it is an automatic error detection technique; and it also detects errors at the source data.



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 2.5: A typical deduplication task.

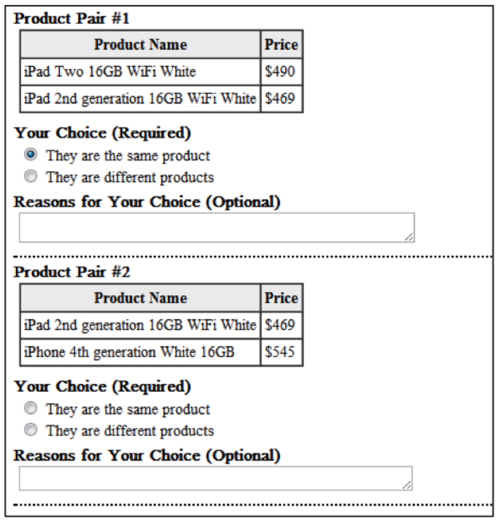
2.3.1 What to Detect

Qualitative errors include violations of integrity constraints, and presence of duplicate records. We have reviewed ICs in Section 2.1, and techniques for their discovery in Section 2.2. 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 [42, 43, 55, 66, 78, 82]: some aim at providing an extensive overview of all the steps involved in data deduplication [43, 55, 66], some focus on the design of similarity metrics [78, 82], some discuss the efficiency aspect of data deduplication [82], and some focus on how to consolidate multiple records [42].

Example 5: Figure 2.5 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 2.5). 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. □

2.3.2 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 [22, 31]. In contrast, deciding whether two records are duplicates usually requires fuzzy matching, for which humans sometimes can achieve better accuracy [56, 91, 93, 94]. We use CrowdER [93] as an example to show how crowd worker can aid a data deduplication task.



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)

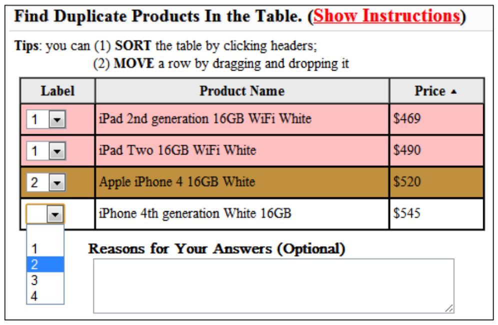
Product Pair #2

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)

(a) Pair-based HIT



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)

(b) Cluster-based HIT

Figure 2.6: HITs for data deduplication

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 2.6 shows two types for HITs used by CrowdER. The pair-based HIT in Figure 2.6(a) asks a human to check each pair of records individually; the cluster-based HIT in Figure 2.6(b) asks a human to cluster multiple records at the same time.

CrowdER proposes a human-machine workflow, which 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 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 6: Figure 2.7 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. 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. □

2.3.3 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.

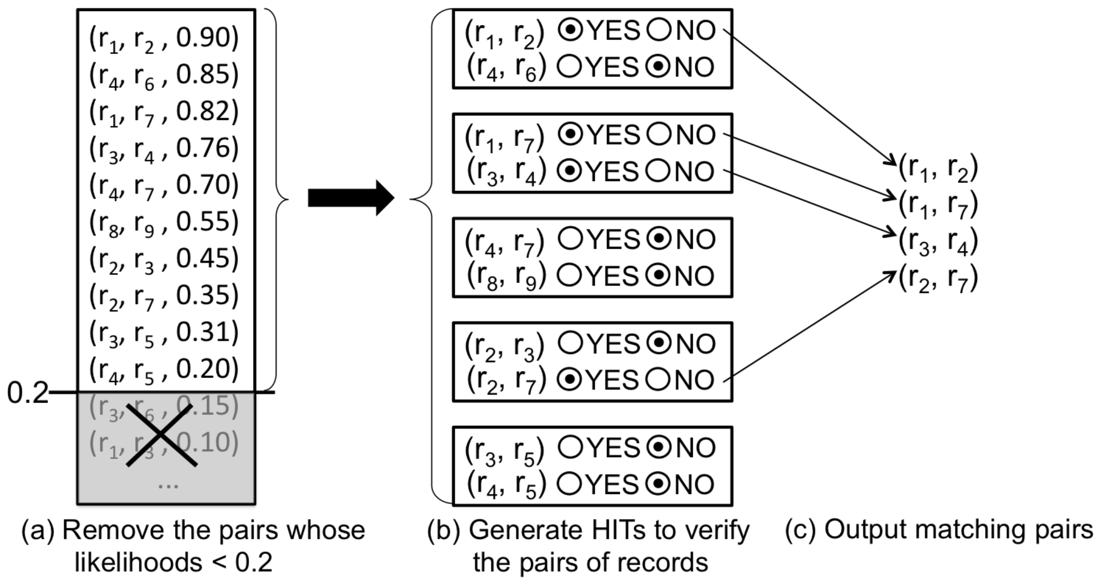


Figure 2.7: CrowdER: an example of using the hybrid human-machine workflow

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 and queries applied on a source database. Figure 2.8 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.

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 and queries assumed, such as Boolean expressions [81], aggregation on numerical attributes [96, 97], and more general SPJA queries [23].

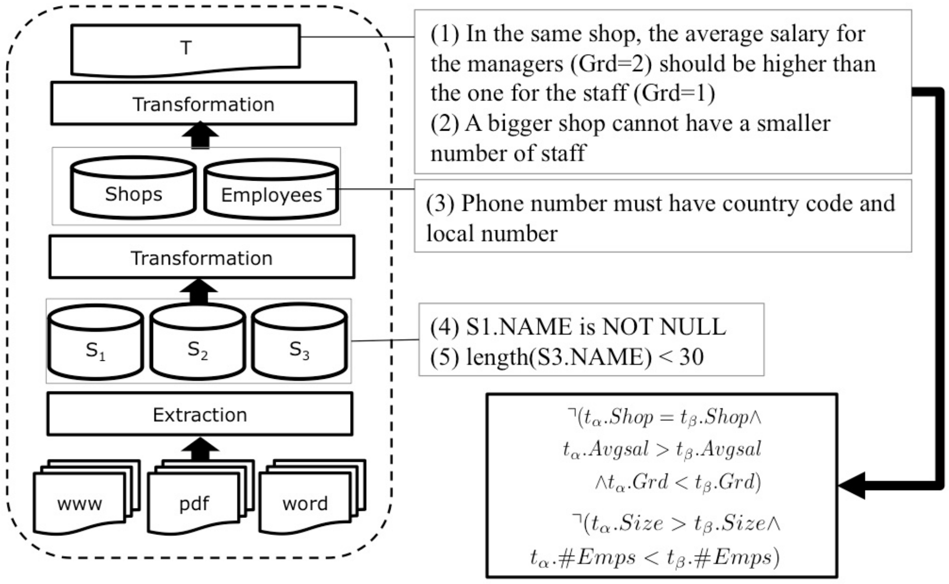

T
Transformation
Shops
Employees
Transformation
S_1
S_2
S_3
Extraction
www
pdf
word
(1) In the same shop, the average salary for the managers (Grd=2) should be higher than the one for the staff (Grd=1)
(2) A bigger shop cannot have a smaller number of staff
(3) Phone number must have country code and local number
(4) S1.NAME is NOT NULL
(5) length(S3.NAME) < 30
$\neg(t_\alpha.Shop = t_\beta.Shop \wedge t_\alpha.Avgsal > t_\beta.Avgsal \wedge t_\alpha.Grd < t_\beta.Grd)$
$\neg(t_\alpha.Size > t_\beta.Size \wedge t_\alpha.\#Emps < t_\beta.\#Emps)$

Figure 2.8: The ETL stack.

2.4 Taxonomy of Qualitative Error Repair

In this section, we discuss related works in qualitative error repair, and show how our proposals in this dissertation fit in and compare with existing literature. Figure 2.9 depicts the classification we adopt to categorize the proposed error repair techniques. In the following, we briefly discuss our classification dimensions, and their impact on the design of underlying error repair techniques. The three adopted dimensions capture the three main questions involved in repairing an erroneous database. We discuss in more detail the three dimension in the following subsections.

- *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

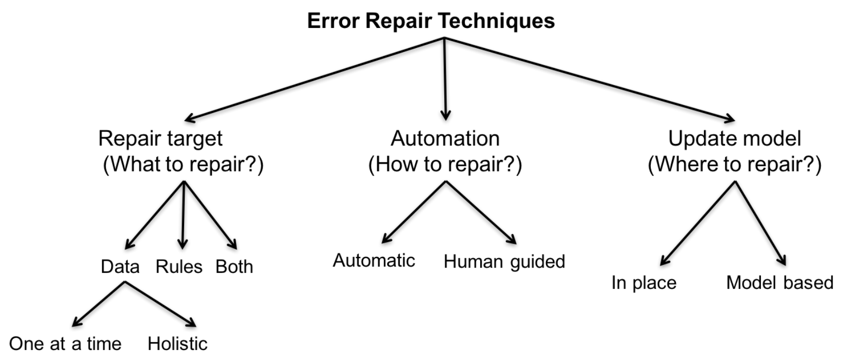


Figure 2.9: Classification of error repair techniques.

of changing both the data and the 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), such as our proposal in Chapter 5.

- *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. For none in-situ repairs, a model is often built to describe the different

	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 [21]	✓				✓		✓	
FDs hypergraph [73]	✓				✓		✓	
CFDs value modification [35]	✓				✓		✓	
Holistic data cleaning [31]		✓			✓		✓	
LLUNATIC [54]		✓			✓		✓	
Record matching and data repairing [50]		✓			✓		✓	
NADEEF [36]		✓			✓		✓	
Generate optimal tablaux [57]			✓		✓		✓	
Unified repair [25]				✓	✓		✓	
Relative trust [16]				✓	✓		✓	
Continuous data cleaning [92]				✓	✓		✓	
AJAX [52, 53]	✓					✓	✓	
Potter's Wheel [86]	✓					✓	✓	
GDR [99]	✓					✓	✓	
KATARA [33]	✓					✓	✓	
DataTamer [88]	✓					✓	✓	
Editing rules [51]	✓					✓	✓	
Sampling FDs repairs [15]	✓				✓			✓
Sampling CFDs repairs [17]	✓				✓			✓
Sampling Duplicates [18]	✓				✓			✓

Table 2.2: A sample of data repairing techniques.

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 2.2 shows a sample of error repair techniques using the taxonomy. Our holistic data cleaning technique changes the data only and is a holistic approach; it is an automatic error repair technique; and it updates the data in place.

2.4.1 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.

Repair Data Only

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. Data only repairing techniques make different assumptions about the driver of the repairing process. In Section 2.4.1, we discussed different error types. For each type of error, such as duplicate records and FD violations, multiple repairing algorithms are proposed. For example, many techniques have been focusing on detecting and consolidating duplicate records only [43, 91]. We will discuss a technique in Section 2.4.2 for repairing violations of FD constraints as an example. Similarly, violations of CFD constraints have been addressed in multiple proposals, either automatically [35] or by involving humans in the loop [99]. We describe these techniques as *One at a time* techniques. Most available data repair 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 and errors are rarely due to a single type; 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 [31,36,48,50,54]. Geerts et al. [54] consider all constraints that can be expressed as equality-generating dependencies. Fan et al. [50] 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. Our proposal in Chapter 5 considers 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.

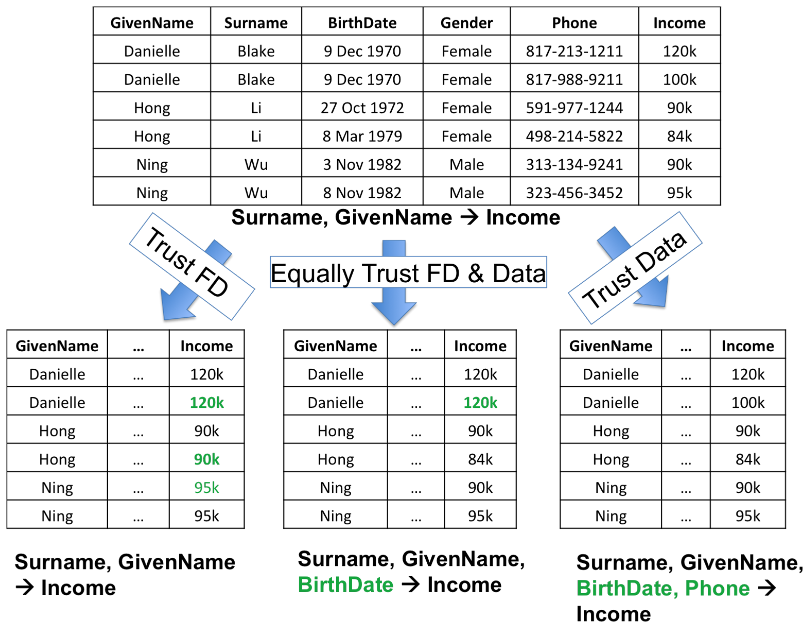
Repair Rules Only

The techniques in this category assume data is clean, and ICs need to be changed, such that data conforms to the changed ICs. In fact, the IC discovery problems discussed in Section 2.2 can be considered in this category. Another particular example is the pattern tableaux discovery problem for CFDs (cf. Section 2.1), where an embedded FD is given [57] and the goal is to discovery the pattern tableaux for the given FD.

Repair Both Data and Rules

Cleaning techniques in this category assumes data and rules can be dirty at the same time. Techniques for repairing both data and rules have mostly been studied when the rules are expressed using FDs [16,25,92]. 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 7: Figure 2.10 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 2.10. If the data is completely trusted, two attributes are added to the LHS of the FD, shown in the bottom middle table in Figure 2.10. If the FD and data have equal trustworthiness, a repair is to only change one cell value and add one attribute to the LHS of the FD, shown in the bottom right table in Figure 2.10. $\square$



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

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

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

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

Surname, GivenName → Income
Surname, GivenName, BirthDate → Income
Surname, GivenName, BirthDate, Phone → Income

Figure 2.10: Relative trust of FDs and data.

2.4.2 How to Repair

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

Automatic Error Repair

There exist multiple theoretic studies [4, 21, 26] and surveys [14, 45] 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. Most of the automatic data repairing techniques aim at updating the database in a way such that the distance between the original database I and the modified database I' is minimized, which is also the objective adopted by our holistic data cleaning proposal in Chapter 5. 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. Many different notions of minimality have been proposed in the literature, including Cardinality-Minimal Repairs [73], Cost-Minimal Repairs [21], Set-Minimal Repairs [10, 14, 79], and Cardinality-Set-Minimal [15] Repairs. In this section, we discuss an example data repair algorithm, which adopts Cardinality-Minimal Repairs.

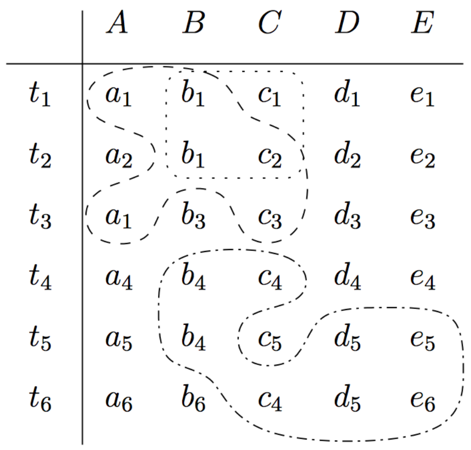
Let $\Delta(I, I')$ denote the set of cells that have different values in I and I'. A repair I' of I is cardinality-minimal if and only if there is no repair I'', such that $|\Delta(I, I'')| < |\Delta(I, I')|$. In other words, a repair I' of I is cardinality-minimal if and only if the number of changed cells in I' is minimum among all possible repairs of I. Algorithm 3 finds a repair I' whose distance to I (*i.e.,*, number of changed cell) is within a constant factor of the optimum $\Delta(I, I')$, where the constant factor depends on the set of FDs [73]. 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 implicant 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 [73].

Algorithm 3 `FindVRepairFDs`

```
Require: Database instance $I$, a set of FDs $\Sigma$
Ensure: 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
```

Example 8: 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 2.11 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 $CD \rightarrow E$.



	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 2.11: An initial conflict hypergraph

There are two other hyperedges not shown in Figure 2.11: 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 3 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 3 terminates after all cells in VC are changed, because there is no more new violations introduced.

□

Human Involved Error Repair

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. This shooting in the dark approach, adopted by most automatic data cleaning algorithms, motivated new approaches that effectively in-

	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

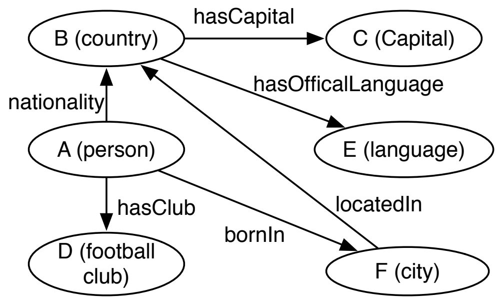
Figure 2.12: A table $\mathcal{T}$ for soccer players

volve humans or experts in the cleaning process to generate reliable fixes. In the following, we list a few examples: AJAX [52] shows how to involve users in a data cleaning process modeled as a directed graph of data transformations; Potter's Wheel [86] is an interactive data cleaning system that tightly integrates data transformation and discrepancy detection; Data Wrangler [61, 71] extends Potter's Wheel's data transformation language; GDR (guided data repair) [99] shows how to effectively incorporate user feedback into CFDs repairing algorithms; Editing rules [51] uses tabular master data and humans to generate verified fixes; KATARA [33] combines KBs (*e.g.,*, Yago and DBPedia), 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 [88] is a data curation system that involves users with different expertise at multiple steps of the curation process. We use KATARA as a concrete example to show how human can help with data repair.

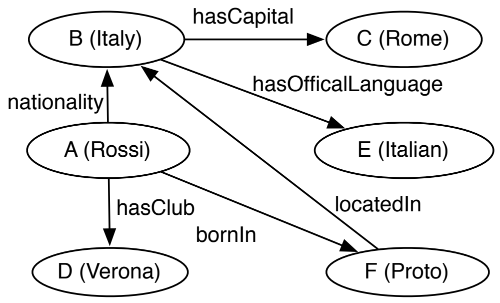
KATARA [33] aims at producing accurate repairs by relying on two authoritative data sources, namely, knowledge bases (KBs), and domain experts. KATARA first discovers table patterns to map the table to a KB, such as Yago or DBPedia. With table patterns, KATARA annotates tuples as either correct or incorrect 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. 2.12). 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 [33] works as follows:

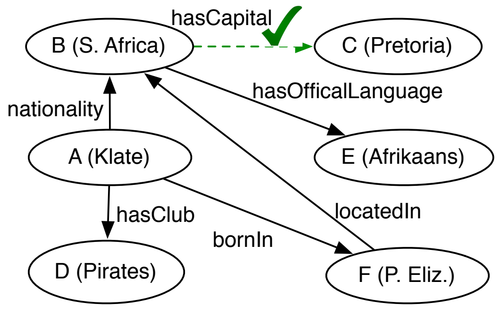
(1) Pattern discovery. KATARA first discovers table patterns that contain the types of the columns and the relationships between them. A table pattern is represented as a labelled graph (Fig. 2.13(a)) where a node represents an attribute and its associated type, *e.g.,*, "*C*



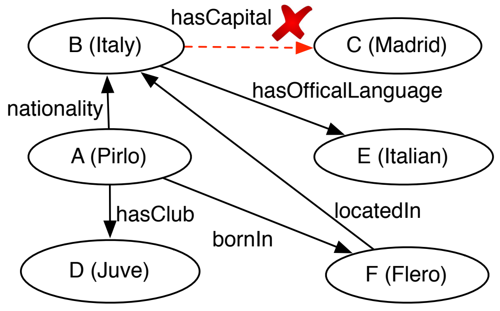
(a) A table pattern φ_s



(b) t_1: validated by KB



(c) t_2: validated by KB&crowd



(d) t_3: Erroneous tuple

Figure 2.13: KATARA patterns

(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. 2.13(a).

(2) Pattern validation. Consider a case where pattern discovery finds two similar patterns: the one in Fig. 2.13(a), and its variant with Type location for column C. To select the best table pattern, we send the crowd the question "*Which type (*capital *or* location*) is more accurate for values (*Rome, Pretoria *and* Madrid*)?*" Crowd answers will help choose the right pattern.

(3) Data annotation. Given the pattern in Fig. 2.13(a), 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. 2.13(b), indicating that Rossi (resp. Italy) is in $\mathcal{K}$ as a person (resp. country), and the relationship from Rossi to Italy is nationality. Similarly, all other values in t_1 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. 2.13(a) 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. 2.13(c). 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.

2.4.3 Where to Repair

Data repair 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 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 2.14(a)). In the following, we describe a different model-based approach for non-destructive data cleaning. Data repairing techniques in this category do not produce a

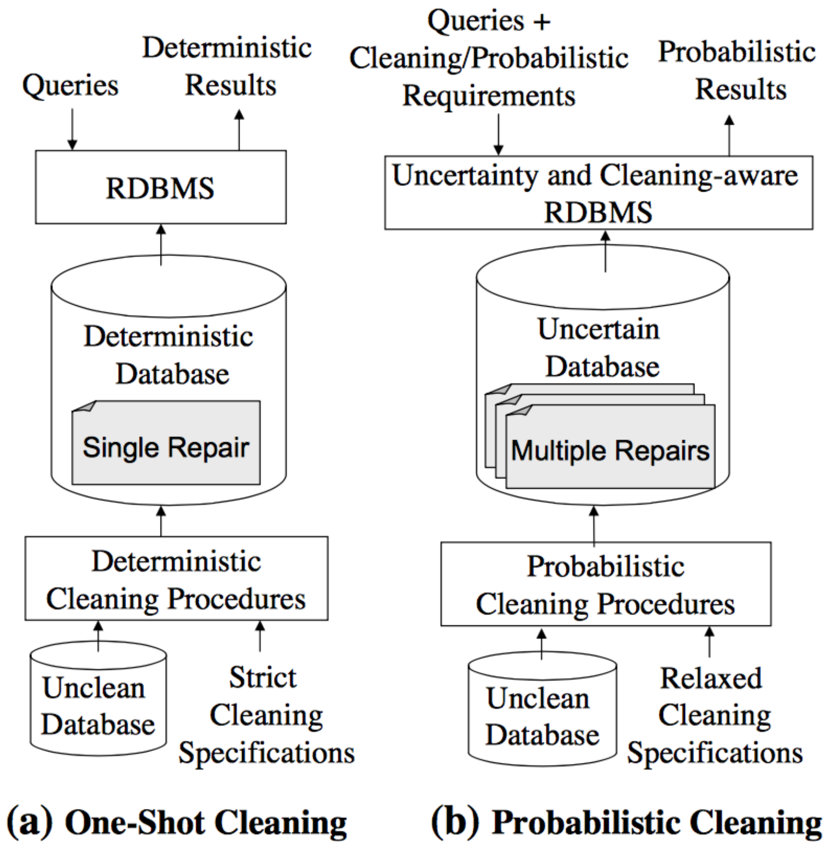
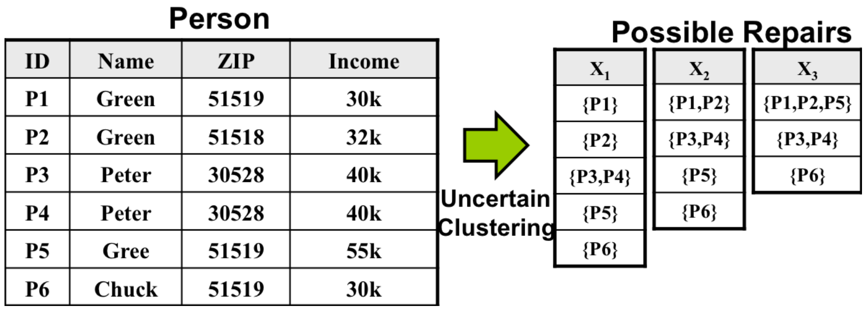

Queries
Deterministic Results
RDBMS
Deterministic Database
Single Repair
Deterministic Cleaning Procedures
Unclean Database
Strict Cleaning Specifications
(a) One-Shot Cleaning
Queries + Cleaning/Probabilistic Requirements
Probabilistic Results
Uncertainty and Cleaning-aware RDBMS
Uncertain Database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

Figure 2.14: One-shot vs. probabilistic cleaning

single repair for a database instance; instead, they produce a space of possible repairs (Figure 2.14(b)). The space of possible repairs is used either to answer queries against the dirty data probabilistically (e.g., using possible worlds semantics) [18], to sample from the space of all possible clean instances of the database [15, 17], or to be used for consistent query answering [10, 14].

We give the details of one model-based algorithm, which 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 [18]. Beskales et al. [18] 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 2.15 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 2.15), 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 2.15). 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 2.15.



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

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

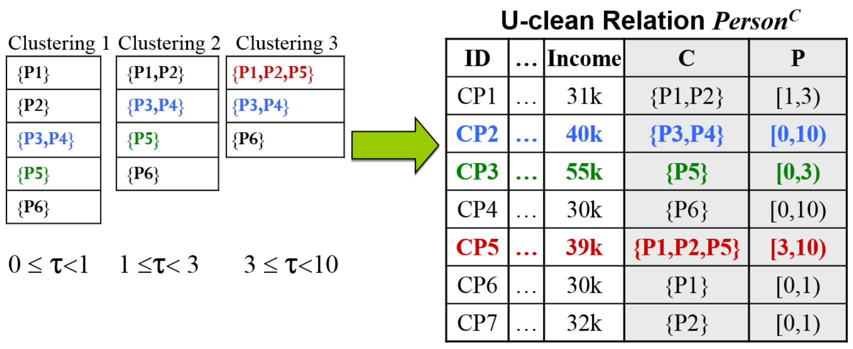
Figure 2.15: Probabilistic duplicate detection

Beskales et al. [18] 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 2.16 illustrates the model of possible repairs for the unclean relation `Person`. U-clean relation $\texttt{Person}^c$ is created by clustering algorithms $\mathcal{A}$ using parameters τ that is defined on the real interval $[0, 10]$ with uniform distributions. Relation $\texttt{Person}^c$ 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 $\texttt{Person}^c$ 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 $[3, 10)$.

Relational queries over U-clean relations are defined using the concept of *possible worlds semantics*, as shown in Figure 2.17. 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 $\texttt{Person}^c$ in Figure 2.16. 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.



U-clean Relation *Person*^{C}

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 2.16: An example of U-clean relation

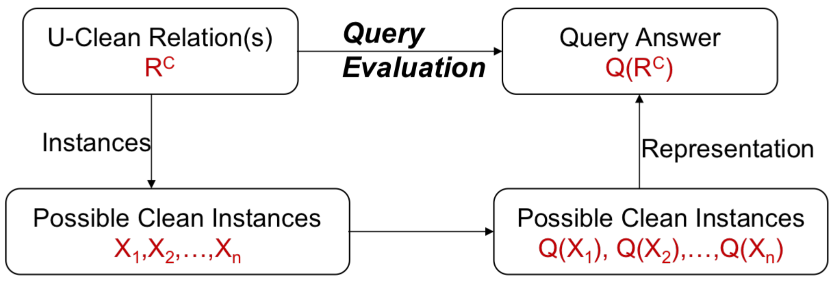


Figure 2.17: U-clean relation query model

Chapter 3

Discovering Denial Constraints

Integrity constraints (ICs) are often used as a formal mechanism to capture data quality rules; they provide a valuable tool for enforcing correct application semantics. Traditional types of ICs, such as key constraints, check constraints, functional dependencies (FDs), and their extension conditional functional dependencies (CFDs) have been proposed for data quality management [22]. However, there is still a big space of data quality rules that cannot be captured by the aforementioned types.

Example 9: Consider a table with the US tax records for individuals in Table 3.1. Each record describes an individual address and tax information with 15 attributes: first and last name (FN, LN), gender (GD), area code (AC), mobile phone number (PH), city (CT), state (ST), zip code (ZIP), marital status (MS), has children (CH), salary (SAL), tax rate (TR), tax exemption amount if single (STX), married (MTX), and having children (CTX).

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

Constraints (1), (2), and (3) can be expressed as a key constraint, an FD, and a CFD,

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

Table 3.1: Tax data records.

respectively.

(1) : $Key\{AC, PH\}$
(2) : $ZIP \rightarrow ST$
(3) : $[CT = \text{'Denver'}] \rightarrow [ST = \text{'CO'}]$

Since Constraints (4) and (5) involve order predicates ($>, <$), and (5) compares different attributes in the same predicate, they cannot be expressed by FDs and CFDs. However, they can be expressed in first-order logic.

$c_4 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL$
$\qquad \wedge t_\alpha.TR > t_\beta.TR)$
$c_5 : \forall t_\alpha \in R, \neg(t_\alpha.SAL < t_\alpha.STX)$

Since first-order logic is more expressive, Constraints (1)-(3) can also be expressed as follows:

$c_1 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$
$c_2 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$
$c_3 : \forall t_\alpha \in R, \neg(t_\alpha.CT = \text{'Denver'} \wedge t_\alpha.ST \neq \text{'CO'})$

□

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

Denial Constraints (DCs) [14, 45], a universally quantified first order logic formalism, can express all constraints in Example 9 as they are more expressive than FDs and CFDs. DCs serve as a great compromise between expressiveness and complexity for the following reasons: (1) they are defined on predicates that can be easily expressed in SQL queries for consistency checking; (2) they have been proven to be a useful language for data cleaning in many aspects, such as data repairing [31], consistent query answering [14], and expressing data currency rules [45]; and (3) while their static analysis turns out to be undecidable [11], we show that it is possible to develop a set of sound inference rules and a linear implication testing algorithm for DCs that enable an efficient adoption of DCs as an IC language, as we show in this chapter.

While DCs can be obtained through consultation with domain experts, it is an expensive process and requires expertise in the constraint language at hand. Therefore, automatic discovery techniques are necessary to suggest potential DCs, which are then verified by domain experts. We identified three challenges that hinder the adoption of DCs as an efficient IC language and in discovering DCs from an input data instance:

(1) Theoretical Foundation. The necessary theoretical foundations for DCs as a constraint language are missing [45]. Armstrong Axioms and their extensions are at the core of state-of-the-art algorithms for inferring FDs and CFDs [47,67], but there is no similar foundation for the design of tractable DCs discovery algorithms.

Example 10: Consider the following constraint, c_6, which states that there cannot exist two persons who live in the same zip code and one person has a lower salary and higher tax rate.

$$\begin{aligned} c_6 : \forall t_\alpha, t_\beta \in R, \neg(& t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.SAL < t_\beta.SAL \\ & \wedge t_\alpha.TR > t_\beta.TR) \end{aligned}$$

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

In order to systematically identify implied DCs (such as c_6), for example, to prune redundant DCs, a reasoning system is needed.

(2) Space Explosion. Consider FDs discovery on schema R, let $|R| = m$. Taking an attribute as the right hand side of an FD, any subset of remaining $m-1$ attributes could serve as the left hand side. Thus, the space to be explored for FDs discovery is $m * 2^{m-1}$. Consider discovering DCs involving at most two tuples without constants; a predicate space needs to be defined, upon which the space of DCs is defined. The structure of a predicate consists of two different attributes and one operator. Given two tuples, we have $2m$ distinct

cells; and we allow six operators $(=, \neq, >, \leq, <, \geq)$. Thus the size of the predicate space $\mathbf{P}$ is: $|\mathbf{P}| = 6 * 2m * (2m-1)$. Any subset of the predicate space could constitute a DC. Therefore, the search space for DCs discovery is of size $2^{|\mathbf{P}|}$.

DCs discovery has a much larger space to explore, further justifying the need for a reasoning mechanism to enable efficient pruning, as well as the need for an efficient discovery algorithm. The problem is further complicated by allowing constants in the DCs.

(3) Verification. Since the quality of ICs is crucial for data quality, discovered ICs are usually verified by domain experts for their validity. Model discovery algorithms suffer from the problem of overfitting [20]; ICs found on the input instance I of schema R may not hold on future data of R. This happens also for DCs discovery.

Example 11: Consider DC c_7 on Table 3.1, which states that first name determines gender.

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

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

Statistical measures have been proposed to rank the constraints and assist the verification step for specific cases. For CFDs it is possible to count the number of tuples that match their tableaux [24]. Similar support measures are used for association rules [7].

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

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

1. We give the formal problem definition of discovering DCs (Section 3.1). We introduce static analysis for DCs with three sound axioms that serve as the cornerstone for our

implication testing algorithm as well as for our DCs discovery algorithm (Section 3.2).

2. We present FASTDC, a DCs discovery algorithm (Section 3.3). FASTDC starts by building a predicate space and calculates evidence sets for it. We establish the connection between discovering minimal DCs and finding minimal set covers for evidence sets. We employ depth-first search strategy for finding minimal set covers and use DC axioms for branch pruning. To handle datasets that may have data errors, we extend FASTDC to discover approximate constraints. Finally, we further extend it to discover DCs involving constant values.

3. We propose a novel scoring function, the *interestingness* of a DC, which combines succinctness and coverage measures of discovered DCs in order to enable their ranking and pruning based on thresholds, thus reducing the cognitive burden for human verification (Section 3.4).

3.1 Preliminary and Problem Definition

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

3.1.1 Denial Constraints Preliminaries

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

We support the subset of integrity constraints identified by *denial constraints* (DCs) over relational databases. We introduce a notation for DCs of the form $\varphi : \forall t_\alpha, t_\beta, t_\gamma, \ldots \in R, \neg(P_1 \wedge \ldots \wedge P_m)$, where P_i is of the form $v_1 \phi v_2$ or $v_1 \phi c$ with $v_1, v_2 \in t_x.A, x \in$

$\{\alpha, \beta, \gamma, \ldots\}$, $A \in R$, and c is a constant. For simplicity, we assume there is only one relation R in $\mathbb{R}$.

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

The *inverse* of predicate P: $v_1 \phi_1 v_2$ is $\overline{P}$: $v_1 \phi_2 v_2$,with $\phi_2 = \overline{\phi_1}$. If P is true, then $\overline{P}$ is false. The set of *implied* predicates of P is $Imp(P) = \{Q | Q : v_1 \phi_2 v_2\}$, where $\phi_2 \in Imp(\phi_1)$. If P is true, then $\forall Q \in Imp(P)$, Q is true. The inverse and implication of the six operators in $\mathbb{B}$ is summarized in Table 3.2.

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

Table 3.2: Operator Inverse and Implication.

Semantics. A DC states that all the predicates cannot be true at the same time, otherwise, we have a violation. Single-tuple constraints (such as check constraints), FDs, and CFDs are special cases of unary and binary denial constraints with equality and inequality predicates. Given a database instance I of schema $\mathbb{S}$ and a DC φ, if I satisfies φ, we write $I \models \varphi$, and we say that φ holds on I. φ is said to be a *logically valid or valid* DC, if $I \models \varphi$ for every correct instance I of $\mathbb{S}$. Since we do not have every correct instance I, to discover valid DCs, we first discover DCs that hold on the input instance I, and then ask experts to verify the validity of discovered DCs.

If we have a set of DC Σ, $I \models \Sigma$ if and only if $\forall \varphi \in \Sigma, I \models \varphi$. A set of DCs Σ implies φ or φ is a logical consequence of Σ, i.e., $\Sigma \models \varphi$, if for every instance I of $\mathbb{S}$, if $I \models \Sigma$, then $I \models \varphi$.

In the context of this paper, we are only interested in DCs with at most two tuples. DCs involving more tuples are less likely in real life, and incur bigger predicate space to search as shown in Section 3.3. The universal quantifier for DCs with at most two tuples are $\forall t_\alpha, t_\beta$. We will omit universal quantifiers hereafter.

3.1.2 Problem Definition

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

Example 12: Consider three additional DCs for Table 3.1.

$c_8 : \neg(t_\alpha.SAL = t_\beta.SAL \wedge t_\alpha.SAL > t_\beta.SAL)$
$c_9 : \neg(t_\alpha.PH = t_\beta.PH)$
$c_{10} : \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL > t_\beta.SAL \wedge t_\alpha.TR < t_\beta.TR)$

c_8 is a trivial DC, since there cannot exist two persons that have the same salary, and one's salary is greater than the other. If we remove tuple t_7 in Table 3.1, c_9 becomes a DC holds on Table 3.1, making c_1 no longer minimal. c_{10} and c_4 are symmetric DCs.

$\square$

Problem Statement. Given a relational schema R and an instance I, the *discovery problem* for DCs is to find all minimal DCs that hold on I. We assume that there are no NULL values in I, and every cell in I contains a single value, so that there is no ambiguity when evaluating whether a DC is violated. Since not all DCs that hold on I are necessarily valid DCs, we also study the problem of ranking DCs with an objective function described in Section 3.4 to assist the users in verifying the validity of discovered DCs.

3.2 Static Analysis of DCs

Armstrong Axioms are the fundamental building blocks for static analysis for FDs [3], namely, determining whether a set of FDs implies another FD. Since DCs subsume FDs, it is natural to ask whether we can perform similar static analysis for DCs, which can be used to prune candidate DCs.

3.2.1 The Inference System of DCs

We present three symbolic inference rules for DCs, denoted as $\mathcal{I}$, analogous to Armstrong Axioms.

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

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

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

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

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

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

Proof: Soundness Proof of Inference System $\mathcal{I}$ for DCs

Reflexivity. The premise is $P_j \Rightarrow \overline{P_i}$. Taking the contrapositive, we get $P_i \Rightarrow \overline{P_j}$. Augmenting both sides with P_j, we get $P_i \wedge P_j \Rightarrow \overline{P_j} \wedge P_j$. Taking the contrapositive, we get $\urcorner(\overline{P_j} \wedge P_j) \Rightarrow \urcorner(P_i \wedge P_j)$. Since $\urcorner(\overline{P_j} \wedge P_j)$ is trivially satisfied in all cases, $\urcorner(P_i \wedge P_j)$ is trivially satisfied as well.

Augmentation. We know that $P_1 \wedge \ldots \wedge P_n \wedge Q \Rightarrow P_1 \wedge \ldots \wedge P_n$ is a tautology. Taking the contrapositive, we have $\urcorner(P_1 \wedge \ldots \wedge P_n) \Rightarrow \urcorner(P_1 \wedge \ldots \wedge P_n \wedge Q)$. $\urcorner(P_1 \wedge \ldots \wedge P_n)$ is in the premise, thus $\urcorner(P_1 \wedge \ldots \wedge P_n \wedge Q)$ is also valid.

Transitivity. Proof by contradiction. Assume that $\urcorner(P_1 \wedge \ldots \wedge P_n \wedge R_1 \wedge \ldots \wedge R_m)$ is not a valid DC. Then there must exist two tuples in r such that $P_1, \ldots, P_n, R_1, \ldots, R_m$ are true at the same time. Since $\urcorner(P_1 \wedge \ldots \wedge P_n \wedge Q_1)$ is a valid DC, then Q_1 must be false. And since $\urcorner(R_1 \wedge \ldots \wedge R_m \wedge Q_2)$ is a valid DC, then Q_2 must be false also. Since $Q_2 \in Imp(\overline{Q_1})$ and Q_1 is false, then Q_2 must be true, which contradicts the falsity of Q_2 we derived before.

Completeness Proof of Inference System for VDCs

We consider VDCs of the form $\forall t_\alpha, t_\beta \in r, \urcorner(P_1 \wedge \ldots \wedge P_m \wedge Q)$, where $P_i = t_\alpha.A_i \theta t_\beta.A_i, \forall i \in [1, m]$ and $Q = t_\alpha.B\overline{\theta}t_\beta.B$ with $A_i, B \in \mathbb{U}$.

Completeness of Axioms states that for a set of DCs Σ, and one DC φ, if φ is logically implied by Σ, denoted as $\Sigma \models \varphi$, φ can be derived from Σ using Axioms, denoted as $\Sigma \vdash_{\mathcal{I}} \varphi$. We prove the contrapositive: if $\Sigma \nvdash_{\mathcal{I}} \varphi$, then $\Sigma \not\models \varphi$. We prove by providing an instance I, s.t. $I \models \Sigma$, but $I \not\models \varphi$. Assume φ is of the form $\urcorner(\mathbf{W} \wedge Q)$, where $\mathbf{W} = \{P_1 \wedge \ldots \wedge P_m\}$

The structure of I we contrive consists of two tuples t_1, t_2, s.t. $\langle t_1, t_2 \rangle \models P, \forall P \in Clo_\Sigma(\mathbf{W})$, and $\langle t_1, t_2 \rangle \not\models P, \forall P \in \Sigma.Pres - Clo_\Sigma(\mathbf{W})$. Due to the special structure of the VDCs we are considering, we are able to construct such instance, because we could separate the attributes of I into two parts: one part of attributes are those that are involved with predicates in $Clo_\Sigma(\mathbf{W})$, the other part of attributes are those that are involved with all the other predicates. We make the following two claims: *Claim 1* $I \models \phi, \forall \phi \in \Sigma$ and *Claim 2* $I \not\models \varphi$. Now we prove two claims separately.

Claim 1: We distinguish two cases according to whether there exists a predicate in ϕ that is in $\Sigma.Pres - Clo_\Sigma(\mathbf{W})$.

(Case 1:) If $\phi.Pres \subseteq \Sigma.Pres$, but $\phi.Pres \nsubseteq Clo_\Sigma(\mathbf{W})$, then it is easy to see that $I \models \phi$.

(Case 2:) If $\phi.Pres \subseteq Clo_{\Sigma}(\mathbf{W})$, we show that such ϕ is impossible. Let $\phi.Pres = \{P_1, P_2, \ldots, P_m\}$. According to the definition of closure, we have that $\phi_1 :\neg(\mathbf{W} \wedge \overline{P_1})$, $\phi_2 :\neg(\mathbf{W} \wedge \overline{P_2})$, $\ldots$, and $\phi_m :\neg(\mathbf{W} \wedge \overline{P_m})$ are valid DCs. Apply Axiom Transitivity on $\phi, \phi_1, \ldots, \phi_m$, we have that $\neg(\mathbf{W})$ is a valid DC, which is impossible. Because if $\neg(\mathbf{W})$ is a valid DC, then according Axiom Augmentation $\varphi :\neg(\mathbf{W} \wedge Q)$ is a valid DC, which contradicting the assumption that $\Sigma \nvdash_{\mathcal{I}} \varphi$.

Claim 2: According to the assumption $I \nvdash_{\mathcal{I}} \varphi$, so $\overline{Q} \notin Clo_{\Sigma}(\mathbf{W})$. Thus $I \not\models \overline{Q}$. Thus $I \models Q$. Thus $I \models \mathbf{W} \wedge Q$. Thus $I \not\models\neg(\mathbf{W} \wedge Q)$, i.e. $I \not\models \varphi$.

□

The completeness result of $\mathcal{I}$ for that form of DCs generalizes the completeness result of Armstrong Axioms for FDs. In particular, FDs adhere to the form with θ being $=$. The partial completeness result for the inference system has no implication on the completeness of the discovery algorithms described in Section 3.3 The inference system for DCs, although not complete, has a huge impact on the pruning power of the implication test and on the FASTDC algorithm.

3.2.2 The Implication Problem of DCs

Implication testing refers to the problem of determining whether a set of DCs Σ implies another DC φ. It has been established that the complexity of the implication testing problem for DCs is coNP-Complete [11]. Given the intractability result, we have devised a linear, sound, but not complete, algorithm for implication testing to reduce the number of DCs in the discovery algorithm output.

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

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

Algorithm 4 calculates the partial closure of $\mathbf{W}$ under Σ. We initialize $Clo_{\Sigma}(\mathbf{W})$ by adding every predicate in $\mathbf{W}$ and their implied predicates due to Axiom Triviality (Line 1-2). We add additional predicates that are implied by $Clo_{\Sigma}(\mathbf{W})$ through basic algebraic transitivity (Line 3). The closure is enlarged if there exists a DC φ in Σ such that all but one predicates in φ are in the closure (Line 15-23). We use two lists to keep track of exactly when such condition is met (Line 3-11).

Example 13: Consider Σ=$\{c_1, \ldots, c_5\}$ and $\mathbf{W} = \{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL\}$.

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

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

Algorithm 5 tests whether a DC φ is implied by a set of DCs Σ, by computing the closure of $\varphi.Pres$ in φ under Γ, which is Σ enlarged with symmetric DCs. If there exists a DC ϕ in Γ, whose predicates are a subset of the closure, φ is implied by Σ.

Example 14: Consider a database with two numerical columns, High (H) and Low (L). Consider two DCs c_{11}, c_{12}.

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

Algorithm 5 identifies that c_{11} implies c_{12}. Let $\Sigma = \{c_{11}\}$ and $\mathbf{W} = c_{12}.Pres$. $\Gamma = \{c_{11}, c_{13}\}$, where c_{13}: $\forall t_\beta, (t_\beta.H < t_\beta.L)$. $Clo_{\Gamma}(\mathbf{W}) = \{t_\alpha.H > t_\beta.H, t_\beta.L > t_\alpha.H, t_\beta.H < t_\beta.L\}$, because $t_\beta.H < t_\beta.L$ is implied by $\{t_\alpha.H > t_\beta.H, t_\beta.L > t_\alpha.H\}$ through basic algebraic transitivity (Line 3).

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

Algorithm 4 Get Partial Closure:

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

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

3.3 DCs Discovery Algorithm

Algorithm 6 describes our procedure for discovering minimal DCs. Since a DC is composed of a set of predicates, we build a predicate space $\mathbf{P}$ based on schema R (Line 1). Any subset of $\mathbf{P}$ could be a set of predicates for a DC.

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

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

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

Section 3.3.1 describes the procedure for building the predicate space $\mathbf{P}$. Section 3.3.2 formally defines Evi_I, gives a theorem that reduces the problem of discovering all minimal DCs to the problem of finding all minimal covers for Evi_I, and presents a procedure for building Evi_I. Section 3.3.3 describes a search procedure for finding minimal covers for Evi_I. In order to reduce the execution time, the search is optimized with a dynamic ordering of predicates and branch pruning based on the axioms we developed in Section 3.2. In order to enable further pruning, Section 3.3.4 introduces an optimization technique that divides the space of DCs and performs DFS on each subspace. We extend FASTDC in Section 3.3.5 to discover approximate DCs and in Section 3.3.6 to discover DCs with constants.

3.3.1 Building the Predicate Space

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

Profiling algorithms [38] can be used to detect joinable and comparable columns. We

[1]For sake of presentation, parameters are described in Section 3.3.3

consider two columns joinable if they are of same type and have common values[2]. Two columns are comparable if they are both of numerical types and the arithmetic means of two columns are within the same order of magnitude.

Example 15: Consider the following Employee table with three attributes: Employee ID (I), Manager ID (M), and Salary(S).

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

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

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

□

3.3.2 Building the Evidence Set

Before giving formal definitions of Evi_I, we show an example of the satisfied predicates for the Employee table Emp above:

$Evi_{Emp} = \{\{P_2, P_3, P_5, P_8, P_{10}, P_{12}, P_{14}\},$
$\{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}, \{P_2, P_3, P_6, P_7, P_{10}, P_{11}, P_{13}\}\}$. Every element in Evi_{Emp} has at least one pair of tuples in I such that every predicate in it is satisfied by that pair of tuples.

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

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

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

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

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

Theorem 2: *$\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a minimal DC that holds on I if and only if $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ is a minimal set cover for Evi_I.* □

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

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

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

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

Example 16: Consider Evi_{Emp} for the table in Example 15.

$\mathbf{X}_1 = \{P_2\}$ is a minimal cover, thus $\neg(\overline{P_2})$, i.e., $\neg(t_\alpha.I = t_\beta.I)$ is a DC holds on I, 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 DC that holds on I, which states that a manager's salary cannot be less than her employee's. $\square$

The procedure to compute Evi_I follows directly from the definition: for every tuple pair in I, we compute the set of predicates that tuple pair satisfies, and we add that set into Evi_I. This operation is sensitive to the size of the database, with a complexity of $O(|\mathbf{P}| \times |I|^2)$. In Chapter 4, we show a distributed strategy that can distribute such "self-join" workload evenly among M parallel workers (machines).

3.3.3 DFS for Minimal Covers

Algorithm 7 presents the depth-first search (DFS) procedure for minimal covers for Evi_I. Ignore Lines (9-10) and Lines (11-12) for now, as they are described in Section 3.3.4 and in Section 3.4.3, respectively. We denote by Evi_{curr} the set of elements in Evi_I not covered so far. Initially $Evi_{curr} = Evi_I$. Whenever a predicate P is added to the cover, we remove from Evi_{curr} the elements that contain P, i.e., $Evi_{next} = \{E | E \in E_{curr} \wedge P \notin E\}$ (Line 23). There are two base cases to terminate the search:

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

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

We speed up the search procedure by two optimizations: dynamic ordering of predicates as we descend down the search tree and branching pruning based on the axioms in Section 3.2.

Opt1: Dynamic Ordering. Instead of fixing the order of predicates when descending down the tree, we dynamically order the remaining candidate predicates, denoted as $>_{next}$, based on the number of remaining evidence set they cover (Lines 23 -24). Formally,

Algorithm 7 Search Minimal Covers

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

we define the cover of P w.r.t. Evi_{next} as $Cov(P, Evi_{next}) = |\{P \in E | E \in Evi_{next}\}|$. And we say that $P >_{next} Q$ if $Cov(P, Evi_{next}) > Cov(Q, Evi_{next})$, or $Cov(P, Evi_{next}) = Cov(Q, Evi_{next})$ and P appears before Q in the preassigned order in the predicate space. The initial evidence set Evi_I is computed as discussed in Section 3.3.2. To computer Evi_{next} (Line 21), we scan every element in Evi_{curr}, and we add in Evi_{next} those elements that do not contain P.

Example 17: Consider Evi_{Emp} for the table in Example 15. We compute the cover for each predicate, such as $Cov(P_2, Evi_{Emp}) = 3$, $Cov(P_8, Evi_{Emp}) = 2$, $Cov(P_9, Evi_{Emp}) = 1$, etc. The initial ordering for the predicates according to Evi_{Emp} is $>_{init} = P_2 > P_3 > P_6 > P_8 > P_{10} > P_{12} > P_{14} > P_5 > P_7 > P_9 > P_{11} > P_{13}$. □

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

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

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

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

3.3.4 Dividing the Space of DCs

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

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

Theorem 3: $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n} \wedge P)$ *is a minimal DC that holds on I and contains predicate P, if and only if $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ is a* minimal set cover *for* Evi_I^P. □

Proof: Follow the same line of the proof for Theorem 2. Consider $\mathbf{X} \subseteq \mathbf{P}, P \notin \mathbf{X}$, that is a cover of Evi_I^P. According to the definition, Evi_I^P represents all the pieces of evidences that might violate DCs containing predicate P. For any $E \in Evi_I^P$, there exists $X \in \mathbf{X}, X \in E$. Thus, we know for sure that $\overline{X} \notin E$. In other words, the presence of $\overline{X}$ disqualifies E as a possible violation of any DC containing predicate P. □

Example 18: Consider Evi_{Emp} for the table in Example 15, $Evi_{Emp}^{P_1} = \emptyset$, $Evi_{Emp}^{P_{13}} = \{\{P_2, P_3, P_6, P_7, P_{10}, P_{11}\}\}$. Thus $\neg(P_1)$ is a DC that holds on I because there is nothing in the cover for $Evi_{Emp}^{P_1}$, and $\neg(P_{13} \wedge \overline{P_{10}})$ is a DC that holds on I as $\{P_{10}\}$ is a cover for $Evi_{Emp}^{P_{13}}$. It is evident that Evi_{Emp}^{P} is much smaller than Evi_{Emp}. □

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

First, a minimal DC containing a certain predicate P is not necessarily a global minimal DC. For instance, assume that $\neg(P, Q)$ is a minimal DC containing P because $\{\overline{Q}\}$ is a minimal cover for Evi_I^P. However, it might not be a minimal DC because it is possible that

$\neg(Q)$, which is actually smaller than $\neg(P,Q)$, is also a DC that holds on I. We call such $\neg(P,Q)$ a *local minimal DC w.r.t.* P, and $\neg(Q)$ a *global minimal DC*, or a minimal DC. It is obvious that a global minimal DC is always a local minimal DC w.r.t. each predicate in the DC. Our goal is to generate all globally minimal DCs.

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

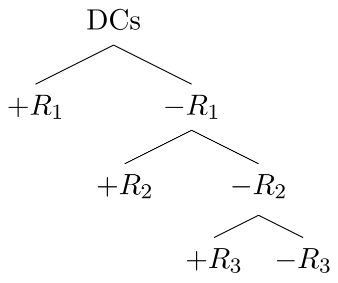


Figure 3.1: Taxonomy Tree.

We solve the second problem first, then the solution for the first problem comes naturally. We divide the DCs space and order all searches in a way, such that we ensure the output of a locally minimal DC is indeed global minimal, and a previously generated minimal DC will never appear again in latter searches. Consider a predicate space $\mathbf{P}$ that has only 3 predicates R_1 to R_3 as in Figure 3.1, which presents a taxonomy of all DCs. In the first level, all DCs can be divided into DCs containing R_1, denoted as $+R_1$, and DCs not containing R_1, denoted as $-R_1$. Since we know how to search for local minimal DCs containing R_1, we only need to further process DCs not containing R_1, which can be divided based on containing R_2 or not, i.e., $+R_2$ and $-R_2$. We will divide $-R_2$ as in Figure 3.1. We can enforce searching for DCs not containing R_i by disallowing $\overline{R_i}$ in the initial ordering of candidate predicates for minimal cover. Since this is a taxonomy of all DCs, no minimal DCs can be generated more than once.

We solve the first problem by performing DFS according to the taxonomy tree in a bottom-up fashion. We start by search for DCs containing R_3, not containing R_1, R_2. Then we search for DCs, containing R_2, not containing R_1, and we verify the resulting

DC is global minimal by checking if the reverse of the minimal cover is a super set of DCs discovered from $Evi_I^{R_3}$. The process goes on until we reach the root of the taxonomy, thus ensuring that the results are both globally minimal and complete.

Dividing the space enables more optimization opportunities:

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

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

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

Example 19: Consider Evi_{Emp} for the table in Example 15, since $Evi_{Emp}^{P_1} = \emptyset$ and $Evi_{Emp}^{P_4} = \emptyset$, then $\mathbf{Q} = \{P_1, P_2, P_3, P_4\}$. Thus we perform $|\mathbf{P}| - |\mathbf{Q}| = 10$ searches instead of $|\mathbf{P}| = 14$. □

2. Additional Branch Pruning. Since we perform DFS according to the taxonomy tree in a bottom-up fashion, DCs discovered from previous searches are used to prune branches in current DFS described by Lines(9-10) of Algorithm 7.

Since Algorithm 7 is an exhaustive search for all minimal covers for Evi_I, Algorithm 6 produces all minimal DCs that hold on I.

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

3.3.5 Approximate DCs: A-FASTDC

Algorithm FASTDC consumes the whole input data set and requires no violations for a DC to be declared valid. In real scenarios, there are multiple reasons why this request may need to be relaxed:

(1) overfitting: data is dynamic and as more data becomes available, overfitting constraints on current data set can be problematic; (2) data errors: while in general learning from unclean data is a challenge, the common belief is that errors constitute small percentage of data, thus discovering constraints that hold for most of the dataset is a common workaround [24,47,67].

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

Different tuple pairs might have the same satisfied predicate set. For every element E in Evi_I, we denote by $count(E)$ the number of tuple pairs $\langle t_x, t_y\rangle$ such that $E = SAT(\langle t_x, t_y\rangle)$. For example, $count(\{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}) = 2$ for the table in Example 15 since $SAT(\langle t_{10}, t_9\rangle) = SAT(\langle t_{11}, t_9\rangle) = \{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}$.

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

Theorem 4 transforms approximate DCs discovery problem into the problem of searching for ϵ-minimal covers for Evi_I.

Theorem 4: *$\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid approximate minimal DC if and only if $\boldsymbol{X}=\{X_1, \ldots, X_n\}$ is a* ϵ-minimal cover *for Evi_I.* □

Proof: The proof of Theorem 4 follows directly from the proof of Theorem 2. □

There are two modifications for Algorithm 7 to search for ϵ-minimal covers for Evi_I: (1) the dynamic ordering of predicates is based on $Cov(P, Evi) = \sum_{E \in \{E \in Evi, P \in E\}} count(E)$; and (2) the base cases (Lines 12-17) are either when the number of violations of the corresponding DC drops below $\epsilon|I|(|I|-1)$, or the number of violation is still above $\epsilon|I|(|I|-1)$ but there are no more candidate predicates to include.

3.3.6 Constant DCs: C-FASTDC

FASTDC discovers DCs without constant predicates. However, just like FDs may not hold on the entire dataset, thus CFDs are more useful, we are also interested in discovering constant DCs (CDCs). Algorithm 8 describes the procedure for CDCs discovery. The first step is to build a constant predicate space $\mathbf{Q}$ (Lines 1-6)[3]. After that, one direct way to discover CDCs is to include $\mathbf{Q}$ in the predicate space $\mathbf{P}$, and follow the same procedure in Section 3.3.3. However, the number of constant predicates is linear w.r.t. the number of constants in the active domain, which is usually very large. Therefore, we follow the approach of [47] and focus on discovering τ-frequent CDCs. The support for a set of constant predicates $\mathbf{X}$ on I, denoted by $sup(\mathbf{X}, I)$, is defined to be the set of tuples that satisfy all constant predicates in $\mathbf{X}$. A set of predicates is said to be τ-frequent if $\frac{|sup(\mathbf{X},I)|}{|I|} \geq \tau$. A CDC φ consisting of only constant predicates is said to be τ-frequent if all strict subsets of $\varphi.Pres$ are τ-frequent. A CDC φ consisting of constant and variable predicates is said to be k-frequent if all subsets of φ's constant predicates are τ-frequent.

Example 20: Consider c_3 in Example 9, $sup(\{t_\alpha.CT = \text{'Denver'}\}, I) = \{t_2, t_6\}$, $sup(\{t_\alpha.ST \neq \text{'CO'}\}, I) = \{t_1, t_3, t_4, t_5, t_7, t_8\}$, and $sup(\{c3.Pres\}, I) = \emptyset$. Therefore, c_3 is a τ-frequent CDC, with $\frac{2}{8} \geq \tau$. □

We follow an "Apriori" approach to discover τ-frequent constant predicate sets. We first identify frequent constant predicate sets of length L_1 from $\mathbf{Q}$ (Lines 7-15). We then generate candidate frequent constant predicate sets of length m from length $m-1$ (Lines

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

Algorithm 8 C-FASTDC

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

22-28), and we scan the database I to get their support (Line 24). If the support of the candidate c is 0, we have a valid CDC with only constant predicates (Lines 12-13 and 25-26); if the support of the candidate c is greater than τ, we call FASTDC to get the variable DCs (VDCs) that hold on $sup(c, I)$, and we construct CDCs by combining the τ-frequent constant predicate sets and the variable predicates of VDCs (Lines 18-21).

3.4 Ranking DCs

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

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

Succinctness is motivated by the Occam's razor principle. This principle suggests that among competing hypotheses, the one that makes fewer assumptions is preferred. It is also recognized that overfitting occurs when a model is excessively complex [20].

Coverage is also a general principle in data mining to rank results [7]. They design scoring functions that measure the statistical significance of the mining targets in the input data.

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

3.4.1 Succinctness

Minimum description length (MDL), which measures the code length needed to compress the data [20], is a formalism to realize the Occam's razor principle. Inspired by MDL, we measure the length of a DC $Len(\varphi)$, and we define the *succinctness* of a DC φ, i.e.,

$Succ(\varphi)$, as the minimal possible length of a DC divided by $Len(\varphi)$ thus ensuring the scale of $Succ(\varphi)$ is between 0 and 1.

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

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

Example 21: Consider a database schema R with two columns A, B, with 3 DCs as follows:

$c_{14} : \neg(t_\alpha.A = t_\beta.A)$, $c_{15} : \neg(t_\alpha.A = t_\beta.B)$,
$c_{16} : \neg(t_\alpha.A = t_\beta.A \wedge t_\alpha.B \neq t_\beta.B)$

Len(c_{14}) = 4 < Len(c_{15}) = 5 < Len(c_{16}) = 6. Succ(c_{14}) = 1, Succ(c_{15}) = 0.8, and Succ(c_{16})=0.67. However, if we use $|\varphi.Pres|$ as Len(φ), Len(c_{14}) = 1 < Len(c_{15}) = 1 < Len(c_{16}) = 2, and Succ(c_{14})=1, Succ(c_{15})=1, and Succ(c_{16})=0.5. □

3.4.2 Coverage

Frequent itemset mining recognizes the importance of measuring statistical significance of the mining targets [7]. In this case, the support of an itemset is defined as the proportion of transactions in the data that contain the itemset. Only if the support of an itemset is above a threshold, it is considered to be frequent. CFDs discovery also adopts such principle. A CFD is considered to be interesting only if their support in the data is above a certain threshold, where support is in general defined as the percentage of single tuples that match the constants in the patten tableaux of the CFDs [24, 47].

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

Example 22: Consider c_2, which is a FD, in Example 9. If we look at single tuples, just as the statistical measure for CFDs, every tuple matches c_2 since it does not have constants. However, it is obvious that the tuple pair $\langle t_4, t_7 \rangle$ gives more support than the tuple pair $\langle t_2, t_6 \rangle$ because $\langle t_4, t_7 \rangle$ matches the left hand side of c_2. $\square$

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

Definition 5 *Given a DC φ:*

A k-evidence (kE) for φ w.r.t. a relational instance I is a tuple pair $\langle t_x, t_y \rangle$, where k is the number of predicates in φ that are satisfied by $\langle t_x, t_y \rangle$ and $k \leq |\varphi.Pres| - 1$.

The weight for a kE ($w(k)$) for φ is $w(k) = \frac{(k+1)}{|\varphi.Pres|}$.

Example 23: Consider c_7 in Example 11, which has 2 predicates. There are two types of evidences, i.e., 0E and 1E.

$\langle t_1, t_2 \rangle$ is a 0E since $t_1.FN \neq t_2.FN$ and $t_1.GD = t_2.GD$.
$\langle t_1, t_3 \rangle$ is a 1E since $t_1.FN \neq t_3.FN$ and $t_1.GD \neq t_3.GD$.
$\langle t_1, t_6 \rangle$ is a 1E since $t_1.FN = t_6.FN$ and $t_1.GD = t_6.GD$.

Clearly, $\langle t_1, t_3 \rangle$ and $\langle t_1, t_6 \rangle$ have higher weight than $\langle t_1, t_2 \rangle$. □

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

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

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

Example 24: Given 8 tuples in Table 3.1, we have 8*7=56 evidences. $Coverage(c_7) = 0.80357$, $Coverage(c_2) = 0.9821$. It can be seen that coverage score is more confident about c_2, thus reflecting our intuitive comparison between c_2 and c_7.

Coverage for CDC is calculated using the same formula, such as $Coverage(c_3) = 1.0$. □

3.4.3 Rank-aware Pruning in DFS Tree

Having defined *Inter*, we can use it to prune branches in the DFS tree when searching for minimal covers in Algorithm 7. We can prune any branch in the DFS tree, if we can upper bound the *Inter* score of any possible DC resulting from that branch, and the upper bound is either (i) less than a minimal *Inter* threshold, or (ii) less than the minimal *Inter* score of the Top-k DCs we have already discovered. We use this pruning in Algorithm 7 (Lines 11-12), a branch with the current path $\mathbf{X}$ will result in a DC φ: $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$, with $\mathbf{X}$ known and $\mathbf{W}$ unknown.

Succ score is an anti-monotonic function: adding more predicates increases the length of a DC, thus decreases the *Succ* of a DC. Therefore we bound $Succ(\varphi)$ by $Succ(\varphi) \leq$

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

3.5 Experimental Study

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

Synthetic. We use the *Tax* data generator from [22]. Each record represents an individual's address and tax information, as in Table 3.1. The address information is populated using real semantic relationship. Furthermore, salary is synthetic, while tax rates and tax exemptions (based on salary, state, marital status and number of children) correspond to real life scenarios.

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

- *Hospital* data is from the US government. There are 17 string attributes, including

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

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

Provider # (PN), measure code (MC) and name (MN), phone (PHO), emergency service (ES) and has 115k tuples.

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

3.5.1 Scalability Evaluation

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

Algorithms. We implemented FASTDC in Java, and we test various optimizations techniques. We use FASTDC+M to represent running FASTDC on a cluster consisting of M machines. We use FASTDC-DS to denote running FASTDC without dividing the space of DCs as in Section 3.3.4. We use FASTDC-DO to denote running FASTDC without dynamic ordering of predicates in the search tree as in Section 3.3.3.

Exp-1: Scalability in $|I|$. We measure the running time in minutes on all 13 attributes, by varying the number of tuples (up to 1 million tuples), as reported in Figure 3.2. The size of the predicate space $|\mathbf{P}|$ is 50. The Y axis of Figure 3.2 is in log scale. We compare the running time of FASTDC and FASTDC+M with number of blocks B=$2M$ to achieve load balancing. Figure 3.2 shows a quadratic trend as the computation is dominated by the tuple pair-wise comparison for building the evidence set. In addition, Figure 3.2 shows that we achieve almost linear improvement w.r.t the number of machines on a cluster; for example, for 1M tuples, it took 3257 minutes on 7 machines, but 1228 minutes on 20 machines. Running FASTDC on a cluster is a viable approach if the number of tuples is too large to run on a single machine.

Exp-2: Scalability in $|\mathbf{P}|$. We measure the running time in seconds using 10k tuples, by varying the number of predicates through including different number of attributes in the Tax dataset, as in Figure 3.3(a). We compare the running time of FASTDC, FASTDC-DS,

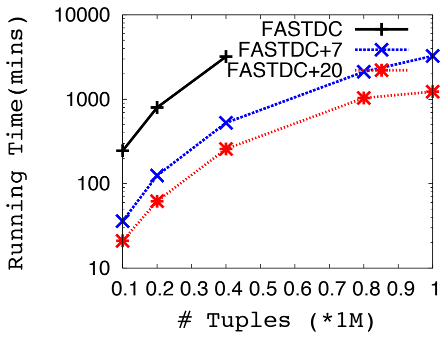
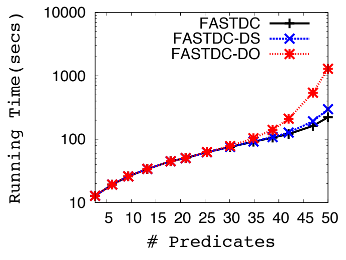


Figure 3.2: Scalability in $|I|$ - Tax

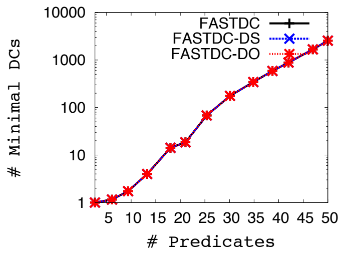
and FASTDC-DO. The ordering of adding more attributes is randomly chosen, and we report the average running time over 20 executions. The Y axes of Figures 3.3(a), 3.3(b) and 3.3(c) are in log scale. Figure 3.3(a) shows that the running time increases exponentially w.r.t. the number of predicates. This is not surprising because the number of minimal DCs, as well as the amount of wasted work, increases exponentially w.r.t. the number of predicates, as shown in Figures 3.3(b) and 3.3(c). The amount of wasted work is measured by the number of times Line 15 of Algorithm 7 is hit. We estimate the wasted DFS time as a percentage of the running time by *wasted work / (wasted work + number of minimal DCs)*, and it is less than 50% for all points of FASTDC in Figure3.3(c). The number of minimal DCs discovered is the same for FASTDC, FASTDC-DS, and FASTDC-DO as optimizations do not alter the discovered DCs.

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

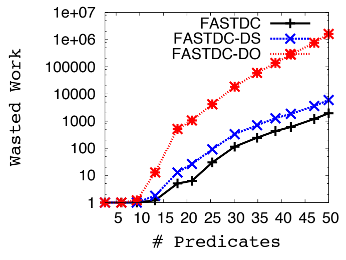
Exp-3: Joinable Column Analysis. Figure 3.4 shows the number of predicates by varying the % of common values required to declare joinable two columns. Smaller values lead to a larger predicate space and higher execution times. Larger values lead to faster execution but some DCs involving joinable columns may be missed. The number of predicates gets stable with low percentage of common values, and with our datasets the quality of the output is not affected when at least 30% common values are required.



(a)



(b)



(c)

Figure 3.3: Scalability in $|\mathbf{P}|$ - Tax

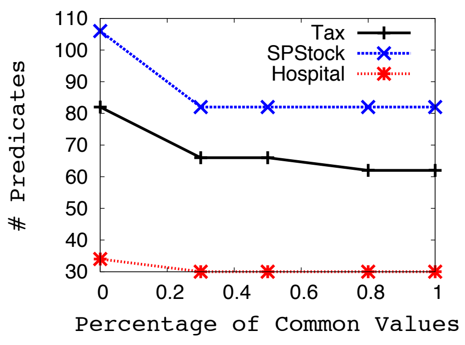


Figure 3.4: Threshold for Joinable Columns

Exp-4: Ranking Function in Pruning. Figure 3.5 shows the DFS time taken for the Tax dataset varying the minimum *Inter* score required for a DC to be in the output. The threshold has to exceed 0.6 to have pruning power. The higher the threshold, the more aggressive the pruning. In addition, a bigger weight for *Succ* score (indicated by smaller a in Figure 3.5) has more pruning power. Although in our experiment golden DCs are not dropped by this pruning, in general it is possible that the upper bound of *Inter* for interesting DCs falls under the threshold, thus this pruning may lead to losing interesting DCs. The other use of ranking function for pruning is omitted since it has little gain.

Exp-5: Implication Reduction. The number of DCs returned by FASTDC can be large, and many of them are implied by others. Table 3.3 reports the number of DCs we have before and after implication testing for datasets with 10k tuples. To prevent

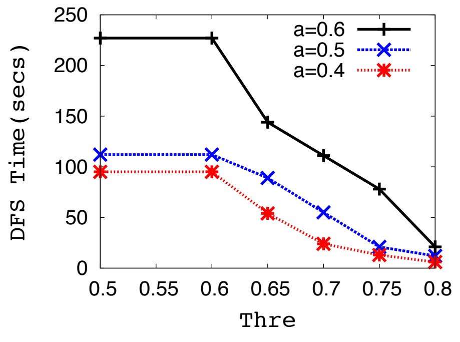


Figure 3.5: Ranking Function in Pruning

interesting DCs from being discarded, we rank them according to their *Inter* function. A DC is discarded if it is implied by DCs with higher *Inter* scores. It can be seen that our implication testing algorithm, though incomplete, is able to prune a large amount of implied DCs.

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

Table 3.3: # DCs before and after reduction through implication.

3.5.2 Qualitative Analysis

Table 3.4 reports some discovered DCs, with their semantics explained in English[6]. We denote by Σ_g the golden VDCs that have been designed by domain experts on the datasets. Specifically, Σ_g for Tax dataset has 8 DCs; Σ_g for Hospital is retrieved from [31] and has 7 DCs; and Σ_g for SP Stock has 6 DCs. DCs that are implied by Σ_g are also golden DCs. We denote by Σ_s the DCs returned by FASTDC. We define *G-Precision* as the percentage

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

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

Table 3.4: Sample DCs discovered in the datasets.

of DCs in Σ_s that are implied by Σ_g, *G-Recall* as the number of DCs in Σ_s that are implied by Σ_g over the total number of golden DCs, and *G-F-Measure* as the harmonic mean of *G-Precision* and *G-Recall*. In order to show the effectiveness of our ranking function, we use the golden VDCs to evaluate the two dimensions of $Inter$ function in Exp-6, the performance of A-FASTDC in Exp-7. We evaluate C-FASTDC in Exp-8. However, domain experts might not be exhaustive in designing all interesting DCs. In particular, humans have difficulties designing DCs involving constants. We show with *U-Precision*(Σ_s) the percentage of DCs in Σ_s that are verified by experts to be interesting, and we report the result in Exp-9. All experiments in this section are done on 10k tuples.

Exp-6: Evaluation of $Inter$ score.

We report in Figures 3.6(a)– 3.6(c) G-Precision, G-Recall, and G-F-Measure for Tax, with Σ_s being the Top-k DCs according to $Inter$ by varying the weight a from 0 to 1. Every line is at its peak value when a is between 0.5 and 0.8. Moreover, Figure 3.6(b) shows that $Inter$ score with $a = 0.6$ for Top-20 DCs has perfect recall; while it is not the case for using $Succ$ alone ($a = 0$), or using $Coverage$ alone ($a = 1$). This is due to two reasons. First, $Succ$ might promote shorter DCs that are not true in general, such as c_7. Second, $Coverage$ might promote longer DCs that have higher coverage than shorter ones, however, those shorter DCs might be in Σ_g; for example, the first entry in Table 3.4 has higher coverage than $\neg(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$, which is actually in Σ_g. For Hospital, $Inter$ and $Coverage$ give the same results, which are better than $Succ$ because golden DCs for Hospital are all FDs with two predicates, therefore $Succ$ has no effect on the interestingness. For Stock, all scoring functions give the same results because its golden DCs are simple DCs, such as $\neg(t_\alpha.Low > t_\alpha.High)$.

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

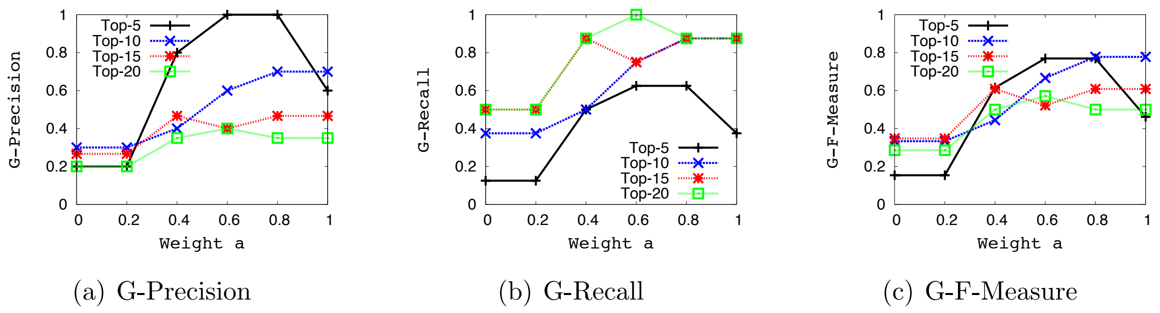
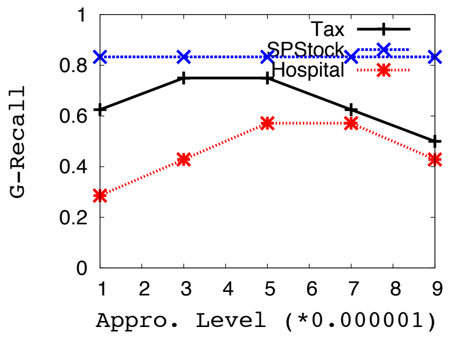


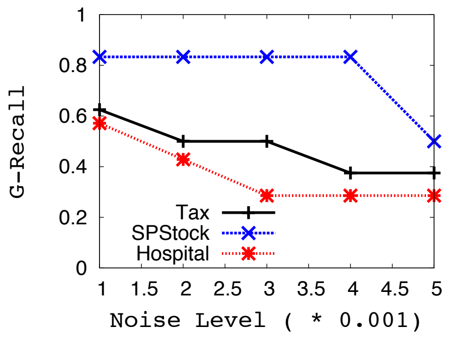
Figure 3.6: G-Precision,Recall, and F-Measure on Tax

Exp-7: A-FASTDC. In this experiment, we test A-FASTDC on noisy datasets. A noise level of α means that each cell has α probability of being changed, with 50% chance of being changed to a new value from the active domain and the other 50% of being changed to a typo. For a fixed noise level $\alpha = 0.001$, which will introduce hundreds of violating tuple pairs for golden DCs, Figure 3.7(a) plots the G-Recall for Top-60 DCs varying the approximation level ϵ. A-FASTDC discovers an increasing number of correct DCs as we increase ϵ, but, as it further increases, G-Recall drops because when ϵ is too high, a DC whose predicates are a subset of a correct DC might get discovered, thus the correct DC will not appear. For example, the fifth entry in Table 3.4 is a correct DC; however, if ϵ is set too high, $\neg(t_\alpha.PN = t_\beta.PN)$ would be in the output. G-Recall for SPStock data is stable and higher than the other two datasets because most golden DCs for SPStock data are one tuple DCs, which are easier to discover. Finally, we examine Top-60 DCs to discover golden DCs, which is larger than Top-20 DCs in clean datasets. However, since there are thousands of DCs in the output, our ranking function is still saving a lot of manual verification.

Figure 3.7(b) shows that for a fixed approximate level $\epsilon = 4 \times 10^{-6}$, as we increase the amount of noise in the data, the G-Recall for Top-60 DCs shows a small drop. This is expected because the nosier gets the data, the harder it is to get correct DCs. However, A-FASTDC is still able to discover golden DCs.



(a) Varying Approximation Level



(b) Varying Noise Level

Figure 3.7: AFASTDC

Exp-8: C-FASTDC. Figure 3.8 reports the running time of C-FASTDC varying minimal frequent threshold τ from 0.02 to 1.0. When $\tau = 1.0$, C-FASTDC falls back to FASTDC. The smaller the τ, the more the frequent constant predicate sets, the bigger the running time. For the SP Stock dataset, there is no constant predicate set, so it is a straight line. For the Tax data, $\tau = 0.02$ results in many frequent constant predicate sets. Since it is not reasonable for experts to design a set of golden CDCs, we only report U-Precision.

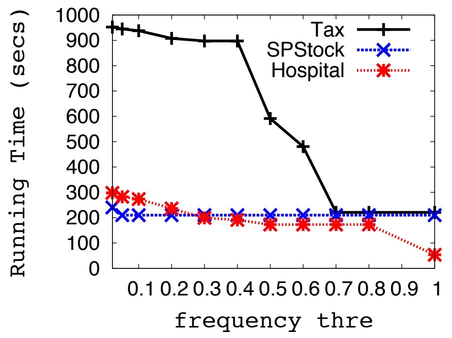


Figure 3.8: C-FASTDC Running Time

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

Table 3.5: U-Precision.

Exp-9: U-Precision. We report in Table 3.5 the U-Precision for all datasets using 10k tuples, and the Top-k DCs as Σ_s. We run FASTDC and C-FASTDC on clean data, as well as noisy data. For noisy data, we insert 0.001 noise level, and we report the best result of A-FASTDC using different approximate levels. For FASTDC on clean data, Top-10 DCs have U-precision 1.0. In fact in Figure 3.6(a), Top-10 DCs never achieve perfect G-precision because FASTDC discovers VDCs that are correct, but not easily designed by humans, such as the second and third entry in Table 3.4. For FASTDC on noisy data, though the results degrade w.r.t. clean data, at least half of the DCs in Top-20 are correct.

For C-FASTDC on either clean or noisy data, we achieve perfect U-Precision for the Tax and the Hospital datasets up to hundreds of DCs. SP Stock data has no CDCs. This is because C-FASTDC is able to discover many business rules such as entries 8-10 in Table 3.4, domain constraints such as entry 11 in Table 3.4, and CFDs such as c_3 in Example 9.

Chapter 4

Distributed Data Deduplication

Data deduplication, also known as record linkage, or entity resolution, refers to the process of identifying tuples in a relation that refer to the same real world entity. Data deduplication is a pervasive problem, and is extremely important for data quality and data integration [43]. For example, finding duplicate customers in enterprise databases is essential in almost all levels of business. In our collaboration with Thomson Reuters, we observed that a data deduplication project takes 3-6 months to complete, mainly due to the scale and variety of data sources.

Data deduplication techniques usually require computing a similarity score of each tuple pair. For a dataset with n tuples, naïvely comparing every tuple pair requires $O(n^2)$ comparisons, a prohibitive cost when n is large. A commonly used technique to avoid the quadratic complexity is blocking [9, 19, 65], which avoids comparing tuple pairs that are obviously not duplicates. Blocking methods first partition all records into blocks and then only records within the same block are compared. A simple way to perform blocking is to scan all records and compute a hash value for each record based on a subset of its attributes, commonly referred to as *blocking key attributes*. The computed hash values are called *blocking key values*. Records with the same blocking key values are grouped into the same block. For example, blocking key attributes can be the zipcode, or the first three characters of the last name. Since one blocking function might miss placing duplicate tuples in the same block, thus resulting in a false negative (for example, zipcode can be wrong or obsolete), multiple blocking functions [43] are often employed to reduce the number of false negatives.

Despite the use of blocking techniques, data deduplication remains a costly process that can take hours to days to finish for real world datasets on a single machine [76]. Most of the previous work on data deduplication is situated in a centralized setting [9, 19, 27], and does not leverage the capabilities of a distributed environment to scale out computation; hence, it does not scale to large distributed data. Big data often resides on a cluster of machines interconnected by a fast network, commonly referred to as a "data lake". Therefore, it is natural to leverage this scale-out environment to develop efficient data distribution strategies that parallelize data deduplication. Multiple challenges need to be addressed to achieve this goal. First, unlike centralized settings, where the dominating cost is almost always computing the similarity scores of all tuple pairs, multiple factors contribute to the elapsed time in a distributed computing environment, including network transfer time, local disk I/O time, and CPU time for pair-wise comparisons. These costs also vary across different deployments. Second, as it is typical in a distributed setting, any algorithm has to be aware of data skew, and achieve load-balancing [12, 41]. Every machine must perform a roughly equal amount of work in order to avoid situations where some machines take much longer than others to finish, a scenario that greatly affects the overall running time. Third, the distribution strategy must be able to handle effectively multiple blocking functions; as we show in this paper, the use of multiple blocking functions impacts the number of times each tuple is sent across nodes, and also induces redundant comparisons when a tuple pair belongs to the same block according to multiple blocking functions.

A recent work Dedoop [74, 75] uses MapReduce [39] for data deduplication; however, it only optimizes for computation cost, and requires a large memory footprint to keep the necessary statistics for its distribution strategy, thus limiting its performance and applicability.The problem of data deduplication is also related to distributed join computation. However, parallel join algorithms are not directly applicable to our setting: (1) most of the work on parallel join processing [5, 83] is for two-table joins, so the techniques are not directly applicable to self-join without wasting almost half of the available workers; (2) even with an efficient self-join implementation, applying it to every block directly without considering the block sizes yields a sub-optimal strategy; and (3) to the best of our knowledge, there is no existing work on processing a disjunction of join queries, a problem we have to tackle in dealing with multiple blocking functions.

In this paper, we propose a distribution strategy with optimality guarantees for distributed data deduplication in a shared-nothing environment. Our proposed strategy aims at minimizing elapsed time by minimizing the maximum cost across all machines. Note that while blocking affects the quality of results (by introducing false negatives), we do not introduce a new blocking criteria, rather we show how to execute a given set of blocking functions in a distributed environment. In other words, our technique does not change the quality but tackles the performance of the deduplication process. We make the following contributions:

• We introduce a cost model that consists of the maximum number of input tuples any machine receives (X), and the maximum number of tuple pair comparisons any machine performs (Y) (Section 4.1). We provide a lower bound analysis for X and Y that is independent of the actual dominating cost in a cluster.

• We propose a distribution strategy for distributing the workload of comparing tuples in a single block (Section 4.2). Both X and Y of our strategy are guaranteed to be within a small constant factor from the lower bound X_{low} and Y_{low}.

• We propose Dis-Dedup for distributing a set of blocks produced by a single blocking function. The X and Y of Dis-Dedup are both within a small constant factor from X_{low} and Y_{low}, regardless of block-size skew (Section 4.3). Dis-Dedup also handles multiple blocking functions effectively, and avoids producing the same tuple pair more than once even if that tuple pair is in the same block according to multiple blocking functions (Section 4.4).

Although we tackle the problem of scalable error detection in the context of data deduplication, our proposed strategy for distributing the workload of comparing tuples in a single block in Section 4.2 is applicable to any error detection that requires tuple pairwise comparison, such as detecting violations of the FD in Example 1 that requires checking whether the IC is violated for every tuple pair, and detecting violations of DCs shown in Example 9.

4.1 Problem Definition and Solution Overview

In this section, we present the parallel computation model we will use in this paper. We formally introduce the problem definition, and provide an overview of our solution.

4.1.1 Parallel Computation Model

We focus on scale-out environments, where data is usually stored in what is called a *data lake*. Such an environment usually adopts a shared-nothing architecture, where multiple machines, or nodes, communicate via a high-speed interconnect network, and each node has its own private memory and disk. In every node there are typically multiple virtual processors running together, so as to take advantage of the multiple CPUs and disks available on each machine and thus increase parallelism. These virtual processors that run in parallel are called *workers* in this paper.

In a shared-nothing system, there is usually a trade-off between the *communication cost* and the *computation cost* [90]. For a particular data processing task, it is often hard to predict which type of cost is dominating, let alone constructing an objective function that combines these two costs. In addition, the influence of each cost on the running time is dependent on many parameters of the cluster configuration. For example, there are more than 250 parameters that are tunable in a Hadoop cluster[1]. In this paper, we follow a similar strategy used in parallel join processing [83], and seek to minimize both costs simultaneously.

Since all workers are running in parallel, to minimize the overall elapsed time, we focus on minimizing the largest cost across all workers. For worker i, let X_i be the communication cost, and Y_i be the computation cost. Assume that there are k workers available. We define X (resp. Y) to be the maximum X_i (resp. Y_i) at any worker:

$$X = \max_{i \in [1,k]} X_i \qquad Y = \max_{i \in [1,k]} Y_i \tag{4.1}$$

[1]http://tinyurl.com/puqduqj

A typical example of a parallel shared-nothing system is MapReduce [39]. MapReduce has two types of workers: the *mapper* and the *reducer*. A mapper takes a key-value pair, and generates a list of key-value pairs; while a reducer takes a key associated with a list of values, and generates another list of key-value pairs. The input keys of the reducers are the output keys of the mappers. Users have the option of implementing a customized *partitioner*. Partitioners decide which key-value pairs are sent to which reducers, based on the key. MapReduce balances the load between mappers very well, however, it is the programmers' responsibility to ensure that the workload across different reducers is balanced.

4.1.2 Formal Problem Definition

We are given a dataset I with n tuples, s blocking functions $h_1, \ldots, h_s$, and a tuple-pair similarity function f. Every blocking function $h \in \{h_1, \ldots, h_s\}$ is applied to every tuple t, and returns a blocking key value $h(t)$. A blocking function h divides all n tuples into a set of m blocks $\{B_1, B_2, \ldots, B_m\}$, where tuples in the same block have the same blocking key value. Tuple pairs in the same block are compared using f to obtain a similarity score. Based on the similarity scores, a clustering algorithm is then applied to group tuples together.

We perform data deduplication using the computational model described in Section 4.1.1. In this case, X_i represents the number of tuples that Worker i receives; and Y_i represents the number of tuple pair comparisons that Worker i performs. We aim at designing a distribution strategy that minimizes (a) the maximum number of tuples any worker receives, namely, X, and (b) the maximum number of tuple pair comparisons any worker performs, namely, Y, at the same time. This may not be possible for a given data deduplication task, but we show that we can always achieve optimality for both X and Y within constant factors. Hence, our algorithm will perform optimally independent of how the runtime is as a function of X and Y.

Example 25: Consider a scenario where a single blocking function produces few large blocks and many smaller blocks. To keep the example simple, suppose that a blocking function partitions a relation of $n = 100$ tuples into 5 blocks of size 10 and 25 blocks of size 2. The total number of comparisons W in this case is $W = 5 \cdot \binom{10}{2} + 25 \cdot \binom{2}{2} = 250$ comparisons.

Assume $k = 10$ workers. Consider first a strategy that sends all tuples to every worker. In this case, $X_i = 100$ for every worker i, which results in $X = 100$ according to Equation (4.1). We then assign $Y_i = \frac{W}{k} = 25$ comparisons to worker i (for example by assigning to worker i tuple pairs numbered $[(i-1)\frac{W}{k}, i\frac{W}{k}]$). Therefore, $Y = 25$ according to Equation (4.1).This strategy achieves the optimal Y, since W is evenly distributed to all workers. However, it has a poor X, since every tuple is replicated 10 times.

Consider a second strategy that assigns one block entirely to one worker. For example, we could assign each of the 5 blocks of size 10 to the first 5 workers, and to each of the remaining 5 workers we assign 5 blocks of size 2. In this case, $X = 10$, since each worker receives exactly the same number of tuples; moreover, each tuple is replicated exactly once. However, even though the input is evenly distributed across workers, the number of comparisons is not. Indeed, the first 5 workers perform $\binom{10}{2} = 45$ comparisons, while the last 5 workers perform only $5 \cdot \binom{2}{2} = 5$ comparisons. Therefore, $Y = 45$ according to Equation (4.1). □

The above example demonstrates that the distribution strategy has significant impact on both X and Y, even in the case of a single blocking function. In the next three sections, we show how we can construct a distribution strategy that achieves an optimal behavior for both X and Y for any distribution of block sizes, and outperforms in practice any alternative strategies.

4.1.3 Solution Overview

Consider a blocking function h that produces m blocks $B_1, B_2, \ldots, B_m$. A distribution strategy would have to assign, for every block B_i a subset of the k workers of size $k_i \leq k$ to handle B_i.

A straightforward strategy assigns one block entirely to one worker, i.e., $k_i = 1, \forall i \in [1, m]$, hence, parallelism happens only across blocks. Another straightforward strategy uses all the available workers to handle every block, i.e., $k_i = k, \forall i \in [1, m]$, hence, parallelism is maximized for every block, and uses an existing parallel join algorithms [5, 83] to handle every block. However, both strategies are not optimal, as we will show in Section 4.3.2.

In light of these two straightforward strategies, we first study how to distribute the workload of one block B_i to k_i workers to minimize X and Y (Section 4.2). Given the distribution strategy for a single block, we then show how to assign workers to blocks $B_1, \ldots, B_m$ generated by a single blocking function h, so as to minimize both X and Y across all blocks (Section 4.3). Given the distribution strategy for a single blocking function h, we will finally present how to assign workers given multiple blocking functions $h_1, \ldots, h_s$, so that the overall X and Y are minimized (Section 4.4).

The optimality results we will show next for X and Y hold as long as the distribution strategy can be implemented in a shared-nothing system, where the distribution strategy specifies (1) which tuples are sent to which workers; and (2) which tuple pairs are compared inside each worker. However, whether our distribution strategy can be implemented in a particular shared-nothing system depends on the APIs of the system. For simplicity, we describe and implement our distribution strategy using MapReduce, which uses mappers and partitioners to specify (1) and uses reducers to specify (2). Other example platforms where our distribution strategies could be implemented include Spark [100], Apache Flink (previously known as Stratosphere) [8], and Myria [63].

4.2 Single Block Deduplication

In this section, we study the problem of data deduplication for tuples in a single block that is produced by one blocking function; in other words, we need to compare every tuple with every other tuple in the block. The distribution strategy presented in this section will serve as a building block when discussing distribution strategies in Sections 4.3 and 4.4. Assume that there are n tuples in the block, and k available reducers to compute the pair-wise similarities.

4.2.1 Lower Bounds

We first analyze the lower bounds X_{low} and Y_{low} for X and Y, respectively. The lower bounds are necessary to reason about the optimality of our distribution strategies.

Theorem 5: *For any distribution strategy that performs data deduplication on a block of size n using k reducers, the maximum input is $X > X_{low} = \frac{n}{\sqrt{k}}$ and the maximum number of comparisons is $Y \geq Y_{low} = \frac{n(n-1)}{2k}$.* □

Proof: To show the lower bound on the number of comparisons Y, observe that the total amount of comparisons required is $\binom{n}{2} = \frac{n(n-1)}{2}$. Since there are k available reducers, there must exist at least one reducer j with $Y_j \geq \frac{n(n-1)}{2k}$.

For the lower bound on the input, suppose for the sake of contradiction that the maximum input is $n' \leq n/\sqrt{k}$. Then, each reducer will perform at most $\binom{n'}{2}$ comparisons, which means that the total number of comparisons will be at most $k\binom{n'}{2} = n(n-\sqrt{k})/2 < n(n-1)/2$, a contradiction (since the comparisons must be at least $\binom{n}{2}$). □

As we show in Section 4.2.2, our algorithm matches the lower bound Y_{low}, but not X_{low}. The problem of designing a strategy that matches X_{low} is tightly related to an extensively studied problem in combinatorics called *covering design* [87]. A (n, ℓ, t)-covering design is a family $\mathcal{F}$ of subsets of size ℓ from the universe $\{1, \ldots, n\}$, such that every subset of size t from the universe is a subset of a set in $\mathcal{F}$. The task in hand is to compute the minimum size $C(n, \ell, t)$ of such a family. To see the connection with our distribution problem, consider a $(n, X, 2)$-covering design $\mathcal{F}$ of size k. Then, we can assign to each of the k reducers a set from the family (that will be of size X); but now, we can perform every comparison in some reducer, since the covering design guarantees that every subset of size 2 (i.e., every pair) will be in some set (i.e., in some reducer). Thus, designing a strategy that achieves X_{low} means finding a $(n, X_{low}, 2)$-covering design, such that $C(n, X_{low}, 2) \leq k$.

The lower bound for X presented in Theorem 5 is called the Schönheim bound [87], but the only constructions that match it are explicit constructions for fixed values of n, ℓ. There exists a large literature of such constructions [58], and it is an open problem to find tight upper and lower bounds. Hence, instead of looking for an optimal solution, our algorithm provides a constant-factor approximation of the lower bound.

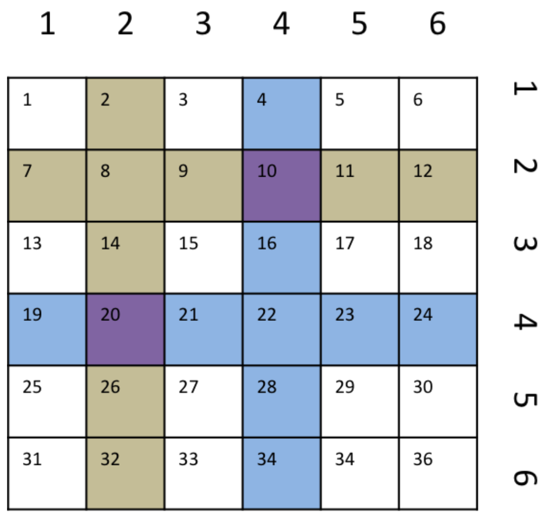
4.2.2 Triangle Distribution Strategy

We present here a distribution strategy, called *triangle distribution strategy*, which guarantees with high probability a small constant-factor approximation of the lower bounds.

The name of the distribution strategy comes from the fact that we arrange the k reducers in a triangle whose two sides have size l (thus $k = l(l+1)/2$ for some integer l). To explain why we organize the reducers in such a fashion, consider the scenario studied in [5, 12] where we compute the cartesian product $R \times S$ of two relations of size n: in this case, the reducers are organized in a $\sqrt{k} \times \sqrt{k}$ square, as shown in Figure 4.1(a) for $k = 36$. Each tuple from R is sent to the reducers of a random row, and each tuple from S is sent to all the reducers of a random column; the reducer function then computes all pairs it receives. However, if we apply this idea directly to a self-join (where $R = S$), the comparison of each pair would be repeated twice, since if a tuple pair ends up together in the reducer (i, j), it will also be in the reducer (j, i). For example, in Figure 4.1(a) tuple t_1 is sent to all reducers in row 2 and column 2, and tuple t_2 is sent to all reducers in row 4 and column 4. Therefore, the joining of t_1 and t_2 is duplicated at reducers $(2, 4)$ and $(4, 2)$. Because of the symmetry, the lower left half of the reducers in the square are doing redundant work. Arranging the reducers in a triangle circumvents this problem and allows us to use all available reducers.

Figure 4.1(b) gives an example of such an arrangement for $k = 21$ reducers with $l = 6$. Every reducer is identified by a two dimensional index (p, q), where p is the row index, and q is the column index, and $1 \leq p \leq q \leq l$. Each reducer (p, q) has a unique reducer ID, which is calculated as $(2l - p + 2)(p - 1)/2 + (q - p + 1)$. For example, Reducer $(2, 4)$ marked purple in Figure 4.1(b) is Reducer 9.

Algorithm 9 describes the distribution strategy given the arrangement for the k reducers. For any tuple t, the mapper randomly chooses an integer a, called an *anchor*, between $[1, l]$, and distributes t to all reducers whose row or column index $= a$ (Lines 3-11). By replicating each tuple l times, we can ensure that for every tuple pair, there exists at least


(a) $R \times S$ join

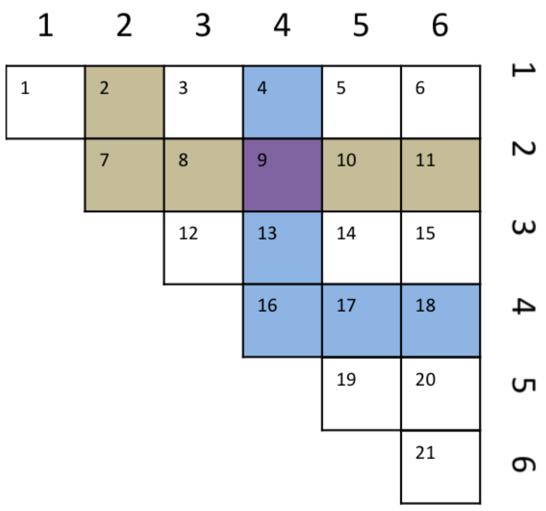

(b) Self-join

Figure 4.1: Reducer arrangement. (The number in the upper left corner of each cell is the reducer id.)

one reducer that receives both tuples. In fact, if two tuples have different anchor points, there is *exactly one* reducer that receives both tuples; while if two tuples have the same anchor point a, both tuples will be replicated on the same set of reducers, but we only compare the tuple pair on reducer (a, a). The key of the key-value pair of the mapper output is the reducer id, and the value of the key-value pair of the mapper output is the tuple augmented with a flag L, S or R to avoid comparing tuple pairs that have the same anchor points a in reducers other than (a, a). Within each reducer, tuples with flag L are compared with tuples with flag R (Lines 28-31), and tuples with flag S are compared only with each other (Lines 33-35).

Example 26: Figure 4.2 gives an example for three tuples t_1, t_2, t_3 given the arrangement of the reducers in Figure 4.1(b). Suppose that tuple t_1 has anchor point $a = 2$, and tuples t_2, t_3 have the same anchor point $a = 4$. The mapping function takes t_1 and generates the key-value pairs $(2, L\#t_1)$, $(7, S\#t_1)$,$(8, R\#t_1)$,$(9, R\#t_1)$,$(10, R\#t_1)$,$(11, R\#t_1)$. Note the different tags L, S, R for different key-value pairs. Reducer 9 receives a list of values

Algorithm 9 Triangle distribution strategy

```
1: class Reducer
2:     method Map(Tuple $t$, null)
3:         Int $a \leftarrow$ a random value from [1,l]
4:         for all $p \in [1, a)$ do
5:             $rid \leftarrow$ rid of Reducer $(p, a)$
6:             Emit(Int $rid$, $L\#$Tuple $t$)
7:         $rid \leftarrow$ rid of Reducer $(a, a)$
8:         Emit(Int $rid$, $S\#$Tuple $t$)
9:         for all $q \in (a, l]$ do
10:            $rid \leftarrow$ rid of Reducer $(a, q)$
11:            Emit(Int $rid$, $R\#$Tuple $t$)
12: class Partitioner
13:     method Partition(Key $rid$, Value $v, k$)
14:         Return $rid$
15: class Reducer
16:     method Reduce(Key $rid$, Values $[v_1, v_2 \ldots]$)
17:         $Left \leftarrow \emptyset$
18:         $Right \leftarrow \emptyset$
19:         $Self \leftarrow \emptyset$
20:         for all Value $(v) \in$ Values $[v_1, v_2 \ldots]$ do
21:             $t \leftarrow v.subString(2)$
22:             if $v$ starts with L then
23:                 $Left \leftarrow Left + t$
24:             else if $v$ starts with R then
25:                 $Right \leftarrow Right + t$
26:             else if $v$ starts with S then
27:                 $Self \leftarrow Self + t$
28:         if $Left \neq \emptyset$ and $Right \neq \emptyset$ then
29:             for all $(t_1) \in Left$ do
30:                 for all $(t_2) \in Right$ do
31:                     compare $t_1$ and $t_2$
32:         else
33:             for all $(t_1) \in Self$ do
34:                 for all $(t_2) \in Self$ do
35:                     compare $t_1$ and $t_2$
```

$R\#t_1, L\#t_2, L\#t_3$ associated with key 9, and compares tuples marked with R with tuples marked with L, but not tuples marked with the same tag. Reducer 16 receives a list of values $S\#t_2, S\#t_3$, and performs comparisons among all tuples marked with S. $\square$

Theorem 6: *The distribution of Algorithm 9 achieves with high probability*[2] *maximum*

[2]The term "with high probability" means that the probability of success is of the form $1 - 1/f(n)$,

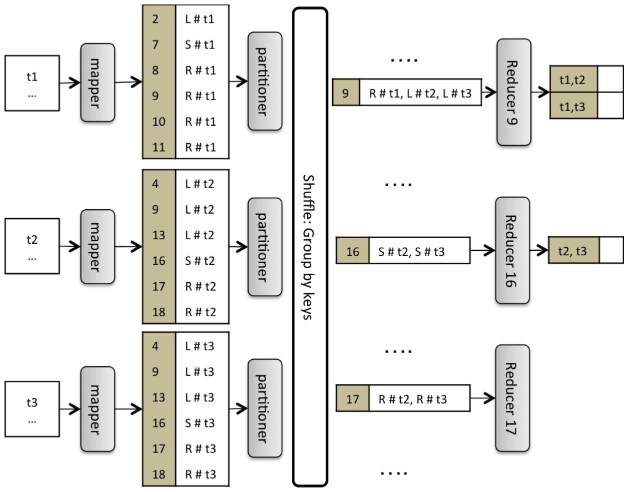


Figure 4.2: Single block distribution example using three tuples, given reducers in Figure 4.1(b)

input $X \leq (1 + o(1))\sqrt{2}X_{low}$ *and maximum number of comparisons* $Y \leq (1 + o(1))Y_{low}$.

$\square$

We first provide the intuition , and then we provide a detailed proof. Fix a reducer $i = (p, q)$. If $p \neq q$, the reducer will receive in expectation n/l tuples with flag L (the ones with anchor p) and n/l tuples with flag R (the ones with anchor R), therefore in expectation $X_i = 2n/l$ and $Y_i = n^2/l^2$. If $p = q$, the reducer will receive in expectation n/l tuples with flag S, therefore in expectation $X_i = n/l$ and $Y_i = n^2/2l^2$. We can show that the X_i and Y_i will also be concentrated around the expectation, and since $k = l(l+1)/2$, we have $X \approx \frac{\sqrt{2}n}{\sqrt{k}}$ and $Y \approx \frac{n^2}{2k}$. Comparing X, Y with the lower bounds in Theorem 5, we have Theorem 6.

Proof: Fix a reducer $i = (p, q)$. If $p \neq q$, the receiver will receive in expectation n/l tuples with flag L (the ones with anchor p) and n/l tuples with flag R (the ones with anchor R). If $p = q$, the reducer will receive in expectation n/l tuples with flag S. We will next use Chernoff bounds to show that with high probability the tuples with flag L, S or

where $f(n)$ is some polynomial function of the size of the dataset n.

R are heavily concentrated around the expected value.

For some flag $f \in \{L, S, R\}$, let X_i^f be the tuples that end up in reducer i with flag f. By applying the Chernoff bound, we have that for some $0 < \delta < 1$:

$$Pr[X_i^f \geq (1+\delta)n/l] \leq e^{\frac{-\delta^2 n}{3l}}$$

Choose now $\delta = \sqrt{3l \ln(n)/n}$, which is $o(1)$ assuming that l is much smaller than n. Then, we obtain:

$$Pr[X_i^f \geq (1+\delta)n/l] \leq 1/n.$$

We now compute the probability that some input X_i^f exceeds $(1+\delta)n/l$ by taking the union bound over all inputs X_i^f. The number of inputs consists of ℓ inputs with flag S (the diagonal reducers), and $(k-\ell)$ inputs with flags L, R (the remaining reducers). Summing up this gives $\ell + 2(k-\ell) = \ell^2$ different inputs. Thus:

$$Pr[\exists i, f : X_i^f \geq (1+\delta)n/l] \leq l^2/n.$$

Since n is much larger than l, this will hold with high probability (with high probability). In this case, with high probability the maximum input is $X \leq (1+\delta)2n/l$, and the maximum number of comparisons is $Y \leq (1+\delta)n^2/l^2$. Since $2k = l(l+1)$, we have that $\sqrt{k} \leq (l+1)/\sqrt{2}$ and thus:

$$\frac{X}{\frac{n}{\sqrt{k}}} \leq \frac{2(1+\delta)\sqrt{k}}{l} \leq (1+\delta)(1+1/l)\sqrt{2} = (1+o(1))\sqrt{2}$$
$$\frac{Y}{\frac{n(n-1)}{2k}} \leq (1+\delta)\frac{n}{n-1} \cdot \frac{2k}{l^2} = 1 + o(1)$$

This concludes the analysis of the distribution algorithm. □

What happens if the k reducers cannot be arranged in a triangle? Following the same idea that applies when k reducers cannot be arranged in a square for an $R \times S$ join [28], we choose the largest possible integer l', such that $l'(l'+1)/2 \leq k$. We have $l'(l'+1)/2 = k' \leq k$ and $(l'+1)(l'+2)/2 > k$. Since both $(l'+1)(l'+2)/2$ and k are integers, we have $(l'+1)(l'+2)/2 - 1 \geq k$. Therefore, the reducer utilization rate is $u = \frac{k'}{k} \geq \frac{l'(l'+1)/2}{(l'+1)(l'+2)/2-1} = 1 - \frac{2}{l'+3} \geq 0.5$. Even for $k = 50$ reducers, we have $l' = 9$, and $u = 0.83$. Observe also that the utilization rate u increases as l' grows.

4.3 Deduplication using Single Blocking Function

In this section, we study distribution strategies to handle a set of disjoint blocks $\{B_1, \ldots, B_m\}$ produced by a single blocking function h. Let m denote the number of blocks, and for each block B_i, where $i \in [1, m]$, we denote by $W_i = \binom{|B_i|}{2}$ the number of comparisons needed. Thus, the total number of comparisons across all blocks is $W = \sum_{i=1}^{m} W_i$.

We will discuss the case where a single blocking function produces a set of overlapping blocks when we discuss multiple blocking functions in Section 4.4.

4.3.1 Lower Bounds

We first prove a lower bound on the maximum input size X and maximum number of comparisons Y for any reducer.

Theorem 7: *For any distribution strategy that performs data deduplication for n tuples and W total comparisons resulting from a set of disjoint blocks using k reducers, we have $X \geq X_{low} \geq \max(\frac{n}{k}, \frac{\sqrt{2W}}{\sqrt{k}})$ and $Y \geq Y_{low} = \frac{W}{k}$.*

□

Proof: Since the total amount of comparisons required is W, there must exist at least one reducer j such that $Y_j \geq \frac{W}{k}$. To prove a lower bound for the maximum input, consider the input size X_j of the reducer j. The maximum number of comparisons that can be performed will then be $X_j(X_j - 1)/2$, which happens when all tuples of the input belong in the same block. Hence, $Y_j \leq X_j(X_j - 1)/2 < X_j^2/2$. Since $Y_j \geq \frac{W}{k}$, we obtain that $X_j^2 > 2W/k$. The $X \geq n/k$ bound comes from the fact that every tuple will have to be sent to at least one reducer, and thus the total size of the inputs must be at least n. □

Notice that Theorem 5 can be viewed as a simple corollary of the above lower bound, since in the case of a self-join we have a single block of size n, so $W = \binom{n}{2}$.

4.3.2 Baseline Distribution Strategies

Assume we have k reducers to handle a set of blocks $B_1, B_2, \ldots, B_m$ produced by a single blocking function. We analyze the baseline strategies Naive-Dedup and PJ-Dedup.

The first baseline strategy Naive-Dedup assigns every block B_i entirely to one reducer. Consider the scenario where there exists a single block B_1 with $|B_1| = n$; then, Naive-Dedup assigns B_1 to one reducer, resulting in $X = n$ and $Y = W$, which is k times worse than X_{low} and Y_{low} (cf. Section 4.3.1), completely defeating the purpose of having k reducers. However, there are scenarios where Naive-Dedup behaves optimally as we show in Example 27.

The second baseline strategy PJ-Dedup uses all k reducers to handle every block B_i, and it uses the triangle distribution strategy discussed in Section 4.2 to perform self-join for every block. However, instead of invoking Algorithm 9 m times for every block $B_i, \forall i \in [1, m]$, which includes the overhead of initializing m MapReduce jobs, we design PJ-Dedup to distribute the tuples as if there was a single block (hence using the triangle distribution strategy of a self-join), and then perform grouping into the smaller blocks inside the reducers. The mapper of PJ-Dedup is similar to that of Algorithm 9, except that the key of the mapper output is a *composite key*, which includes both the reducer ID (as in Algorithm 9) and the blocking key value. The partition function of PJ-Dedup simply takes the composite key and returns the reducer ID part. The reduce function of PJ-Dedup is exactly the same as that of Algorithm 9, since the MapReduce framework automatically groups by blocking key values within each reducer. Regardless of the block sizes, PJ-Dedup has $X \approx \frac{\sqrt{2}n}{\sqrt{k}}$ because the tuples are sent by the mappers in the same way as Algorithm 9, and $Y \approx \frac{W}{k}$ because the workload W is roughly evenly distributed amongst k reducers.

Example 27: We consider three blocking functions that generate blocks of different sizes. For each of them, we show the lower bounds X_{low} and Y_{low}, and analyze how Naive-Dedup, and PJ-Dedup perform w.r.t. those lower bounds. For PJ-Dedup, regardless of the blocking function, $X \approx \frac{\sqrt{2}n}{\sqrt{k}}$ and $Y \approx \frac{W}{k}$, as explained previously.

(1) The first blocking function h_1 produces βk blocks of equal size, for an integer $\beta > 1$,

that is, $|B_i| = \frac{n}{\beta k}$ for all $i \in [1, \beta k]$. In this case, $W = \frac{\frac{n}{\beta k}(\frac{n}{\beta k}-1)}{2}\beta k$, and thus Theorem 7 gives us $X_{low} = \frac{n}{k}$ and $Y_{low} = \frac{W}{k}$. For Naive-Dedup, every reducer receives $\frac{\beta k}{k} = \beta$ blocks. Therefore, $X = \frac{n}{\beta k}\beta = \frac{n}{k}$ and $Y = \frac{\frac{n}{\beta k}(\frac{n}{\beta k}-1)}{2}\beta = \frac{W}{k}$, which is optimal.

(2) The second blocking function h_2 produces only one block of size n. In this case, $W = \frac{n(n-1)}{2}$, and thus Theorem 7 gives us $X_{low} \approx \frac{n}{\sqrt{k}}$ and $Y_{low} = \frac{W}{k}$. For Naive-Dedup, one reducer does all the work, and thus $X = n$ and $Y = W$.

(3) The third blocking function h_3 produces $\frac{k}{\beta}$ blocks of equal size for some $1 \leq \beta < k$, that is, $|B_i| = \frac{\beta n}{k}$ for all $i \in [1, \frac{k}{\beta}]$. In this case, $W = \frac{\frac{\beta n}{k}(\frac{\beta n}{k}-1)}{2}\frac{k}{\beta}$, and thus Theorem 7 gives us $X_{low} = \frac{\sqrt{\beta}n}{k}$ and $Y_{low} = \frac{W}{k}$. For Naive-Dedup, since the number of blocks is less than the number of reducers, one block is assigned to one reducer, leading to $X = \frac{\beta n}{k}$ and $Y = \frac{\frac{\beta n}{k}(\frac{\beta n}{k}-1)}{2} = \frac{\beta W}{k}$, both of which are not bounded. For PJ-Dedup Y is optimal, but X is a factor $\sqrt{\frac{2k}{\beta}}$ away from the lower bound.

The comparison is summarized in Table 4.1. For h_1, Naive-Dedup matches the lower bounds for both X and Y; for h_2, PJ-Dedup matches the lower bounds; and for h_3, neither matches the lower bounds.

□

	h_1		h_2		h_3	
	X	Y	X	Y	X	Y
Lower bounds	$\frac{n}{k}$	$\frac{W}{k}$	$\frac{n}{\sqrt{k}}$	$\frac{W}{k}$	$\frac{\sqrt{\beta}n}{k}$	$\frac{W}{k}$
Naive-Dedup	$\frac{n}{k}$	$\frac{W}{k}$	n	W	$\frac{\beta n}{k}$	$\frac{\beta W}{k}$
PJ-Dedup	$\frac{\sqrt{2}n}{\sqrt{k}}$	$\frac{W}{k}$	$\frac{\sqrt{2}n}{\sqrt{k}}$	$\frac{W}{k}$	$\frac{\sqrt{2}n}{\sqrt{k}}$	$\frac{W}{k}$

Table 4.1: Three example blocking functions

Example 27 demonstrates that (1) when the block sizes are small and uniform, such as the ones produced by h_1, we should use one reducer to handle each block, as in Naive-Dedup; (2) when there are dominating blocks, such as the ones produced by h_2, we should use multiple reducers to divide the workload, as in PJ-Dedup; and (3) when there are multiple relatively large blocks, we should use multiple reducers to handle every large block to avoid unbalanced computation. However, using all k reducers for every large block sends more tuples than necessary, since tuples from different blocks might be sent to same reducer, even though they will not be compared, as in PJ-Dedup.

4.3.3 The Proposed Strategy

Dis-Dedup adopts a distribution strategy which guarantees that both X and Y are always within a constant factor from X_{low} and Y_{low}, by assigning reducers to blocks in proportion to the workload of every block.

Intuitively, since we want to balance computation, a block of a larger size needs more reducers than a block of a smaller size. Since the blocks are independent, we allocate the reducers to blocks in proportion to their workload, namely, block B_i will be assigned to $k_i = \frac{W_i}{W}k$ reducers. However, k_i might not be an integer, and it is meaningless to allocate a fraction of reducers. Thus, k_i needs to be rounded to an integer. If $k_i > 1$, we can assign $\lfloor k_i \rfloor \geq 1$ reducers to B_i. On the other hand, if $k_i \leq 1$, which means $\lfloor k_i \rfloor = 0$, we must still assign at least one reducer to B_i. The total number of reducers after rounding might be greater than k, in which case reducers have to be responsible for more than one block. Therefore, we need an effective way of assigning reducers to blocks, such that both X and Y are minimized.

If $k_i \leq 1$, we call B_i a *single-reducer block*; otherwise, B_i is a *multi-reducer block*. Let $\mathcal{B}_s$ and $\mathcal{B}_l$ be the set of single-reducer blocks and multi-reducer blocks respectively. Next, we show how to handle single-reducer blocks and multi-reducer blocks separately, such that X and Y are bounded by a constant factor.

$$\mathcal{B}_s = \{B_i \mid W_i \leq \frac{W}{k}\}, \quad \mathcal{B}_l = \{B_i \mid W_i > \frac{W}{k}\}$$

For the sake of convenience, assume that we have ordered the blocks in increasing order of their workload: $W_1 \leq W_2 \leq \ldots \leq W_c \leq \frac{W}{k} < W_{c+1} \leq \ldots \leq W_m$. Let $W_l = \sum_{i=c+1}^{m} B_i$ be the total amount of workload for multi-reducer blocks, and $W_s = \sum_{i=1}^{c} B_i$ be the total amount of workload for single-reducer blocks. Also, let X_s (resp. X_l) be the maximum number of tuples from single-reducer blocks (resp. multi-reducer blocks) received by any reducer; and let Y_s (resp. Y_l) be the maximum number of comparisons from single-reducer blocks (resp. multi-reducer blocks) performed by any reducer. Therefore, $X \leq X_s + X_l$ and $Y \leq Y_s + Y_l$.

Handling multi-reducer blocks

Every block $B_i \in \mathcal{B}_l$ has $k_i \geq 1$ reducers assigned to it, and we will use k_i reducers to distribute B_i via the triangle distribution strategy in Section 4.2. If k_i is fractional, such as $k_i = 3.1$, we will simply use $\lfloor k_i \rfloor$ reducers to handle B_i. Since $\sum_{i=c+1}^{m} k_i \leq k$, every reducer will exclusively handle at most one multi-reducer block.

Example 28: Recall the blocking function h_3 in Example 27: every block is large, since $W_i > \frac{W}{k}$. Instead of using all k reducers to handle every block, we now use $k_i = \frac{W_i}{W}k = \beta$ reducers to handle block B_i. Thus, we have $X = X_l = \frac{\sqrt{2}}{\sqrt{k_i}}|B_i| = \frac{\sqrt{2\beta}}{k}n$, and $Y = Y_l = \frac{W_i}{k_i} = \frac{W}{k}$. Compared to the lower bound, we see that Y is optimal, and X is only $\sqrt{2}$ away from optimal. □

In fact, we can be even more aggressive in assigning reducers to big blocks, by assigning $k_i = \frac{W_i}{W_l}$ (instead of $k_i = \frac{W_i}{W}$) reducers to B_i. This still guarantees that there is at least one reducer for every multi-reducer block, and one reducer handles at most one multi-reducer block, since $\sum_{i=c+1}^{m} k_i = k$. By handling multi-reducer blocks this way, Dis-Dedup achieves the following bounds for X_l and Y_l:

Theorem 8: D*is-Dedup has with high probability* $Y_l \leq (1 + o(1))2Y_{low}$ *and* $X_l \leq (1 + o(1))2X_{low}$ □

Proof: Every multi-reducer block B_i will get assigned $k_i = \lfloor \frac{W_i}{W_l}k \rfloor \geq \frac{W_i}{2W_l}k$ reducers.

From the analysis of the triangle distribution strategy, we know that with high probability Y_l will be an $(1 + o(1))$ factor away from the following quantity:

$$\begin{aligned}\max_{i=c+1}^{m}\left(\frac{W_i}{k_i}\right) &= \max_{i=c+1}^{m}\left(\frac{W_i}{\lfloor \frac{W_i}{W_l}k \rfloor}\right) \\ &\leq \max_{i=c+1}^{m}\left(\frac{2W_i}{\frac{W_i}{W_l}k}\right) = \frac{2W_l}{k} \leq \frac{2W}{k}\end{aligned}$$

As for X_l, we know again that with high probability it will be an $(1+o(1))\sqrt{2}$ factor from:

$$\begin{aligned}\max_{i=c+1}^{m}\left(\frac{|B_i|}{\sqrt{k_i}}\right) &\leq \max_{i=c+1}^{m}\left(\frac{|B_i|}{\sqrt{\frac{W_i}{2W_l}k}}\right) \\ &\leq \frac{\sqrt{2W_l}}{\sqrt{k}} \cdot \max_{i=c+1}^{m}\left(\frac{|B_i|}{\sqrt{W_i}}\right) \\ &\leq \frac{\sqrt{2W_l}}{\sqrt{k}} \cdot \max_{i=c+1}^{m}\left(\frac{|B_i|}{\sqrt{|B_i|(|B_i|-1)/2}}\right) \\ &= (1+o(1))\frac{2\sqrt{W_l}}{\sqrt{k}}\end{aligned}$$

Note that in the proof, we used $\lfloor k_i \rfloor \geq k_i/2$, which has accounted for the utilization rate (which is at least 0.5) when $\lfloor k_i \rfloor$ cannot be arranged in a triangle. Comparing this with the lower bounds in Theorem 7, we conclude the proof. □

Handling single-reducer blocks

For every block $B_i \in \mathcal{B}_s$, since we assign $k_i \leq 1$ reducers to it, we can use one reducer to handle every single-reducer block, just like Naive-Dedup does. However, we must assign single-reducer blocks to reducers to ensure that every reducer has about the same amount of workload.

We first present a *deterministic distribution* strategy for single-reducer blocks, which achieves a constant bound for X_s and Y_s. However, the deterministic strategy requires the mappers to keep the ordering of the sizes of single-reducer blocks in memory, which is very costly. We next consider a *randomized distribution*, which is cheaper to implement, but whose bound for X is dependent on k. Finally, we introduce a *hybrid distribution* strategy that uses the randomized distribution for most of the single-reducer blocks, and the deterministic distribution for only a small subset of the single-reducer blocks. The hybrid distribution requires a small memory footprint, and in the same time achieves constant bounds for both X_s and Y_s.

Deterministic Distribution. In order to allocate the single-reducer blocks evenly, we first order the c single-reducer blocks according to their block sizes, and divide them into $g = \frac{c}{k}$ groups[3], where each group consists of consecutive k blocks in the ordering. Then, we assign to each reducer g blocks, one from each group.

Theorem 9: *The deterministic distribution strategy for single-reducer blocks achieves* $Y_s \leq 2Y_{low}$ *and* $X_s \leq 2X_{low}$. □

Proof: Recall that we started by ordering the blocks in increasing size. Hence, we have that $X_s \leq \sum_{i=1}^{g} |B_{i \cdot k}|$ and $Y_s \leq \sum_{i=1}^{g} W_{i \cdot k}$, where the X_s and Y_s reaches the worst if a reducer happens to get assigned the biggest block from every group.

We first derive the bounds for Y_s. Since $W_1 \leq W_2 \leq \ldots \leq W_c$, we can write the following equation:

$$\begin{aligned} W_s = \sum_{i=1}^{c} W_i &= \sum_{i=1}^{k} W_i + \sum_{i=k+1}^{2k} W_i + \ldots + \sum_{i=(g-1)k+1}^{gk} W_i \\ &\geq \sum_{i=1}^{k} 0 + \sum_{i=k+1}^{2k} W_k + \ldots + \sum_{i=(g-1)k+1}^{gk} W_{(g-1)k} \\ &= k \sum_{i=1}^{g-1} W_{i \cdot k} \end{aligned}$$

Therefore, we have $\sum_{i=1}^{g-1} W_{i \cdot k} \leq \frac{W_s}{k} \leq \frac{W}{k}$. Also, $W_{g \cdot k} = W_c \leq W/k$. Thus, $Y_s \leq \sum_{i=1}^{g-1} W_{i \cdot k} + W_{g \cdot k} \leq 2\frac{W}{k}$.

[3]We assume that $\frac{c}{k}$ is an integer; otherwise, we can conceptually add less than k empty blocks to $\mathcal{B}_s$ to make $\frac{c}{k}$ an integer, and the analysis remains intact.

We now derive the bounds for X_s. We have the following equation due to $|B_1| \leq |B_2| \leq \ldots \leq |B_c|$:

$$\begin{aligned}\sum_{i=1}^{c} |B_i| &= \sum_{i=1}^{k} |B_i| + \sum_{i=k+1}^{2k} |B_i| + \ldots + \sum_{i=(g-1)k+1}^{gk} |B_i| \\ &\geq \sum_{i=1}^{k} 0 + \sum_{i=k+1}^{2k} |B_k| + \ldots + \sum_{i=(g-1)k+1}^{gk} |B_{(g-1)k}| \\ &= k \sum_{i=1}^{g-1} |B_{i \cdot k}|\end{aligned}$$

Therefore, we have $\sum_{i=1}^{g-1} |B_{i\cdot k}| \leq \frac{\sum_{i=1}^{c} |B_i|}{k} \leq \frac{n}{k}$. Also, $|B_{g\cdot k}| = |B_c| \leq \frac{\sqrt{2W}}{\sqrt{k}}$, since we have $W_c = \frac{|B_c|(|B_c|-1)}{2} \leq \frac{W}{k}$. Thus, $X_s \leq \sum_{i=1}^{g-1} |B_{i\cdot k}| + |B_{g\cdot k}| \leq \frac{n}{k} + \frac{\sqrt{2W}}{\sqrt{k}} \leq 2\max(\frac{n}{k}, \frac{\sqrt{2W}}{\sqrt{k}})$

Compare X_s and Y_s with the lower bounds in Theorem 7, we conclude the proof. □

The problem with implementing the deterministic distribution is that there can be a large number of single-reducer blocks, and keeping track of the ordering of all block sizes is an expensive task within each mapper, not to mention the need to actually order all the single-reducer blocks.

Randomized Distribution. This algorithm simply distributes each single-reducer block to a reducer by using a random hash function. In order to analyze the randomized distribution, it suffices to consider the worst-case scenario, which is when we have k single-reducer blocks, each with $|B_i| = n/k$. In this case, the problem becomes a *balls-into-bins* scenario, where we have k balls (blocks) that we distribute independently and uniformly at random into k bins (reducers). It then holds [84] that with high probability each reducer will receive a maximum of $O(\ln(k))$ blocks. Thus:

Theorem 10: *The randomized distribution strategy for single-reducer blocks achieves* $Y_s \leq \ln(k)Y_{low}$ *and* $X_s \leq \ln(k)X_{low}$. □

Proof: Under the worst case analysis, where we have k single-reducer blocks with size $\frac{n}{k}$, each reducer will receive, with high probability, at most $ln(k)$ blocks. We also have the workload of every single block is $W_i = \frac{W}{k}$. Thus $Y_s \leq ln(k)\frac{W}{k} = ln(k)Y_{low}$. Similarly, we have $X_s \leq ln(k)\frac{n}{k} \leq ln(k)X_{low}$. $\square$

The randomized method is efficient in practice, since we do not have to keep track of the single-reducer blocks, but we are adding (in the worst case) an additional $\ln(k)$ factor.

Hybrid Distribution. The hybrid algorithm combines the randomized and deterministic distribution to achieve efficiency and almost optimal distribution. To start, we set a threshold $\tau = \frac{W}{3k\ln(k)}$.[4] For the blocks where $W_i \geq \tau$ (but still $W_i \leq \frac{W}{k}$), we use the deterministic distribution, while for the blocks where $W_i < \tau$ we use the randomized distribution. Observe that now the deterministic option is much cheaper, since we have to keep track of at most $3k\ln(k)$ blocks (which number depends only on the number of reducers, and not n).

Theorem 11: *The hybrid distribution strategy for single-reducer blocks has with high probability $Y_s \leq (1 + o(1))3Y_{low}$ and $X_s \leq (1 + o(1))3X_{low}$.* $\square$

Proof: For the randomly distributed blocks, we can apply a result from [12] (based on a type of Chernoff bound) on the weighted balls-into-bins problem, and obtain that with high probability the maximum number of comparisons is $Y_s^r \leq (1 + o(1))Y_{low}$. Using the same result, we can show that the maximum input size will be with high probability $X_s^r \leq (1 + o(1))X_{low}$.

For the deterministically distributed blocks, since $\tau \leq \frac{W}{k}$, we can directly apply Theorem 9 to obtain that $Y_s^d \leq 2Y_{low}$ and $X_s^d \leq 2X_{low}$. The desired result is obtained since $X_s \leq X_s^r + X_s^d$ and $Y_s \leq Y_s^r + Y_s^d$. $\square$

Algorithm 10 Dis-Dedup

```
class Mapper
    $BKV2RIDs \leftarrow$ empty dictionary
    method MapperSetup($HM_1$, $HM_2$)
        $W_l \leftarrow \sum_{bkv \in HM_1.keySet()} HM_1[bkv]$
        $S \leftarrow \{1, 2, \ldots, k\}$
        for all $bkv \in HM_1.keySet()$ do
            $k_i \leftarrow \lfloor \frac{HM_1[bkv]}{W_l} k \rfloor$
            $RIDS_i \leftarrow$ select $k_i$ elements from $S$
            $BKV2RIDs[bkv] \leftarrow RIDS_i$
            $S \leftarrow S - RIDS_i$
        $SortedBKVs \leftarrow$ sort all keys in $HM_2$
        $RID \leftarrow 0$
        for all $bkv \in SortedBKVs$ do
            $BKV2RIDs[bkv] \leftarrow RID\%k$
            $RID \leftarrow RID + 1$
    method Map(Tuple $t$, null)
        $bkv \leftarrow h(t)$
        if $BKV2RIDs[bkv] \neq \emptyset$ then
            $rid \leftarrow$ a random number from $[1, k]$
            Emit(Key $rid\#bkv$, $S\#$Tuple $t$)
        else
            $RIDs \leftarrow BKV2RIDs[bkv]$
            $k_i \leftarrow RIDs.size()$
            $l_i \leftarrow$ the large integer s.t. $l_i(l_i - 1)/2 < k_i$
            Int $a \leftarrow$ a random value from $[1, l_i]$
            for all $p \in [1, a)$ do
                $ridIndex \leftarrow$ rid of Reducer $[p, a]$
                $rid \leftarrow RIDs[ridIndex]$
                Emit(Key $rid\#bkv$, $L\#$Tuple $t$)
            $ridIndex \leftarrow$ rid of Reducer $[a, a]$
            $rid \leftarrow RIDs[ridIndex]$
            Emit(Key $rid\#bkv$, $S\#$Tuple $t$)
            for all $q \in (a, l]$ do
                $ridIndex \leftarrow$ rid of Reducer $[a, q]$
                $rid \leftarrow RIDs[ridIndex]$
                Emit(Key $rid\#bkv$, $R\#$Tuple $t$)
class Partitioner
    method Partition(Key $rid\#bkv$, Value $v$, $k$)
        Return $rid$
class Reducer
    method Reduce(Key $rid\#bkv$, Values $[v_1, v_2 \ldots]$)
        same as the reduce function in Algorithm 9
```

Implementation

Dis-Dedup combines the triangle distribution strategy for multi-reducer blocks and the hybrid distribution for single-reducer blocks. However, given a new tuple ingested by a mapper, how does the mapper know whether the tuple belongs to a multi-reducer block or a single-reducer block? In order to make this decision, we need some preprocessing to collect statistics to be loaded into each mapper. In particular, we need to compute the blocking key values of every multi-reducer block $B_i \in \mathcal{B}_l$, and the associated workload W_i. Let HM_1 denote the HashMap data structure that stores the mapping from a blocking key value in the multi-reducer blocks to the size of the block. In addition, we need the blocking key values of those single-reducer blocks $B_i \in \mathcal{B}_l$ that have $W_i \geq \tau$, and the associated workload W_i, in order to implement the hybrid distribution strategy for the single-reducer blocks. Let HM_2 denote the HashMap data structure that stores the mapping from a blocking key value to the size of the block.

To compute HM_1 and HM_2, we use three simple word-count alike MapReduce jobs. The first job takes the original dataset as input, and counts the size of every block B_i. The second job takes as input the result of the first job, and counts the total workload W. The third job takes as input the result of the first job and W, and outputs the blocks for which $W_i > \frac{W}{k}$, namely HM_1, and also the blocks where $\frac{W}{3k\ln(k)} < W_i \leq \frac{W}{k}$, namely HM_2.

Algorithm 10 describes in detail Dis-Dedup. The mapper now has a setup method that needs to be executed, which allocates reducers to blocks in HM_1 and HM_2 (Lines 3-15). To allocate reducers to blocks in HM_1, we first compute the sum of workload of all the multi-reducer blocks, i.e., W_l (Line 4). For every blocking key value in HM_1, we calculate the number of reducers k_i allocated to it, and select k_i reducers from the set of k reducers (Lines 5-10). For every blocking key value in HM_2, we first sort them based on their workload (Line 11), and allocate one reducer to the sorted blocks in a round-robin manner (Lines 12-15). In the actual mapping function, given a new tuple t and its blocking key

[4]The threshold value is chosen as a result of the connection to the weighted balls-into-bins problem. When we throw k balls of weight W/k into k bins uniformly at random, the expected maximum number of balls is $O(\ln(k))$, and so the maximum weight is $O(W\ln(k)/k)$. If we instead throw $k\ln(k)$ balls of weight $W/k\ln(k)$, the expected maximum number remains $O(\ln(k))$, but since the balls are of smaller weight the maximum weight decreases to $O(W/k)$.

value *bkv*, we check if we have a fixed set of reducers allocated to it. If there is no such fixed allocation, *bkv* must the block that is randomly distributed, and thus we randomly choose a reducer to send the tuple (Lines 18-20). If there is a fixed set of reducer $RIDs$ allocated to it, we distribute t according to PJ-Dedup, by arranging reducer $RIDs$ in a triangle (Lines 22-36). The partitioner sends a key-value pair according to the *rid* part in the key (Lines 37-39). The reducer is the same as that of Algorithm 9 (Lines 40-42).

Theorem 12: D*is-Dedup with hybrid distribution for single-reducer blocks achieves with high probability* $X \leq c_x X_{low}$*, and* $Y \leq c_y Y_{low}$*, where* $c_x = 5 + o(1)$ *and* $c_y = 5 + o(1)$*.* □

Theorem 12 is obtained by combining Theorems 8 and 11.

4.4 Deduplication using Multiple Blocking Functions

Since a single blocking function might result in false negatives by failing to assign duplicate tuples to the same block, multiple blocking functions are often used to decrease the likelihood of a false negative. In this section, we study how to distribute the blocks produced by s different blocking functions $h_1, h_2, \ldots, h_s$.

A straightforward strategy to handle s blocking functions would be to apply Dis-Dedup s times (possibly simultaneously). However, this straightforward strategy has two problems: (1) it fails to leverage the independence of the blocking functions, and the tuples from blocks generated by different blocking functions might be sent to same reducer, where they will not be compared. This is similar to PJ-Dedup's failure to leverage the independence of the set of blocks generated by a single blocking function, as shown in Example 27; and (2) a tuple pair might be compared multiple times, if that tuple pair belongs to multiple blocks, each from a different blocking function. We address these two problems by two principles: (1) allocating reducers to blocking functions proportional to their workload; and (2) imposing an ordering of the blocking functions.

Reducer Allocation. We allocate one or more reducers to a blocking function in proportion to the workload of that blocking function, similar to Dis-Dedup discussed in Section 4.3. Let m_j denote the number of blocks generated by a blocking function h_j, B_i^j

denote the i-th block generated by h_j, and $W_i^j = \binom{|B_i^j|}{2}$ denote the workload of B_i^j. Let $W^j = \sum_{i=1}^{m_j} W_i^j$ be the total amount of workload generated by h_j, and $W = \sum_{j=1}^{t} W^j$ be the total workload generated by all t blocking functions. Therefore, the number of reducers B_i^j gets assigned is $\frac{W_i^j}{W^j} \cdot \frac{W^j}{W}k$, where $\frac{W^j}{W}k$ is the number of reducers for handling the blocking function h_j. Thus, the number of reducers assigned to B_i^j is $\frac{W_i^j}{W}$, regardless of which blocking function it originates from.

This means that we can view the blocks from multiple blocking functions as a set of (possibly overlapping) blocks produced by one blocking function, and apply Dis-Dedup as-is. The only modification needed is that in the mapper of Dis-Dedup, instead of applying one hash function to obtain one blocking key value, we apply s hash functions to obtain s blocking key values (we also need to apply the body of the mapping function for every blocking key value). We call the slightly modified version of Dis-Dedup to handle multiple blocking functions Dis-Dedup$^+$.

Theorem 13: *For t blocking functions,* D*is-Dedup*$^+$ *achieves* $X < (5s + o(1))X_{low}$*, and* $Y = (5 + o(1))Y_{low}$. □

Proof: The lower bound for X and Y for s blocking functions is the same lower bound for single blocking function, as stated in Theorem 12, except W now denotes the total number of comparisons for all s blocking functions. The proof of Theorem 13 follows the same line as the proof of Theorem 12. The only difference is when we are dealing with $\mathcal{B}_s$: instead of having the inequality $\sum_{i=1}^{c} |B_i| < n$ for a single blocking function, we now have $\sum_{i=1}^{c} |B_i| < sn$, which leads to an additional factor s in the bound for X. □

The above analysis tells us that the number of comparisons will be optimal, but the input may have to be replicated as many as s times. The reason for this increase is that the blocks may overlap, in which case tuples that belong in multiple blocks may be replicated. The results from Theorem 13 can be carried to a single blocking function that produces a set of overlapping blocks, where $s = \frac{\sum_{i=1}^{m} |B_i|}{n}$.

Blocking Function Ordering. Since a tuple pair can have the same blocking key values according to multiple blocking functions, a tuple pair can occur in multiple blocks.

To avoid producing the same tuple pair more than once, we impose an ordering of the blocking functions, from h_1 to h_s. Every reducer has knowledge of all s blocking functions, and their fixed ordering. Inside every reducer, before a tuple pair t_1, t_2 is compared according to the j^{th} blocking function h_j, it applies all of the the lower-numbered blocking function $h_z, \forall z \in [1, j-1]$, to see if $h_z(t_1) = h_z(t_2)$. If such h_z exists, then the tuple pair comparison according to h_j is skipped in that reducer, since we are sure that there must exist a reducer that can see the same tuple pair according to h_z. In this way, every tuple pair is only compared according to the *lowest numbered blocking function* that puts them in the same block. The blocking function ordering technique assumes that applying blocking functions is much cheaper than applying the comparison function. If this is not true (e.g., the pairs comparison is cheap) the ordering benefit will not be obvious. Based on our discussion with Thomson Reuters and Tamr, which are engaged in large-scale deduplication projects, the assumption of expensive comparison is often true in practice, since it is usually performed by applying multiple classifiers and machine learning models.

Example 29: Suppose two tuples t_1 and t_2 are in the same block according to the blocking functions h_2, h_3, h_5, namely, $h_2(t_1) = h_2(t_2), h_3(t_1) = h_3(t_2)$ and $h_5(t_1) = h_5(t_2)$. In this case, we only compare t_1 and t_2 in the block generated by h_2, and omit the comparison of those two tuples in the blocks generated by h_3 and h_5. □

To recognize which blocking function generated a tuple pair comparison, we augment the key of the mapper from *rid#bkv* to *rid#bkv#hIndex*, where *hIndex* is the blocking function number that generated this *bkv*. In the reduce function, before a tuple pair is compared, we check if there is another blocking function whose index is smaller than the *hIndex* that puts that tuple pair in the same block. If such a blocking function exists, we skip the comparison.

4.5 Experimental Study

In this section, we evaluate the effectiveness of our distribution strategies. All experiments are performed on a cluster of eight machines, running Hadoop 2.6.0 [1]. Each machine has 12 Intel Xeon CPUs at 1.6 GHz, 16MB cache size, 64GB memory, and 4TB hard disk. All machines are connected to the same gigabit switch. One machine is dedicated to serve as the resource manager for YARN and as the HDFS namenode, while the other seven are slave nodes. The HDFS block size is the default 64MB, and all machines serve as data nodes for the DFS. Every node is configured to be able to run 7 reduce tasks at the same time, and thus in total, 49 reduce tasks can run in parallel. Our cluster resembles the cluster that is used in production by Thomson Reuters to perform data deduplication. We also use a real dataset from that company, as described next.

Datasets. We design a *synthetic data* generator, which takes as input the number of tuples n, the number of blocks m, and a parameter θ controlling the distribution of block sizes, following the zipfian distribution. Every generated tuple has two columns $\{A, B\}$. Column A is the blocking key attribute, whose values range from 1 to m. The blocking function h for a tuple t in the synthetic dataset is $h(t) = t[A]$. Thus, tuples with the same value for A belong to the same block. Column B is a randomly generated string of 1000 characters. The comparison between two tuples is the edit distance between the two values for B.

We use two real datasets. The first one is a list of publication records from CITESEERX [5], called CSX. The second one, called OA, is a private dataset from Thomson Reuters, containing a list of organization names and their associated information, such as addresses and phone numbers, extracted from news articles. CSX uses the publication title as the blocking key attribute, while OA uses the organization name. The blocking function for both datasets uses minHash [69]. A minHash signature is generated for each tuple, and tuples with the same signature belong in the same block. For each tuple in CSX or OA, the blocking function extracts 3-grams from the blocking key attribute, and applies a random hash function to the set of 3-grams; the minimum hash value for the set of 3-grams is used as the minHash signature for that tuple. We can generate multiple blocking functions by using different random hash functions. The comparison between two tuples is the edit distance between the two values of the blocking key attributes.

Table 4.2 summarizes the statistics for each dataset we used for performing data deduplication using a single blocking function in Section 4.5.2. We will use up to 20 blocking functions for performing data deduplication using multiple blocking functions in Sec-

[5] http://csxstatic.ist.psu.edu/about

Dataset	# Tuples	AVG block size	MAX block size
CSX	1.4M	1.75	35021
OA	7.4M	1.31	7969
SYNTHETIC	20M	4	3967

Table 4.2: Datasets Statistics

tion 4.5.3.

Algorithms. Dedoop [74, 75] is a state-of-the-art distribution strategy for data deduplication, which has two main drawbacks: (1) it only aims at optimizing Y, with no consideration for X; and (2) it has large memory requirements for mappers and reducers. Dedoop starts by assigning an index to every tuple pair that needs to be compared. For example, for a block with four tuples t_1, t_2, t_3, t_4, tuple pair $\langle t_1, t_2 \rangle$ has index 1, tuple pair $\langle t_1, t_3 \rangle$ has index 2, and so on, and tuple pair $\langle t_3, t_4 \rangle$ has index 6. Dedoop then assigns each reducer an equal number of tuple pairs to compare. If W denotes the total number of tuple pairs, the i^{th} reducer will take care of tuple pairs whose indexes are in $[(i-1)\frac{W}{k}, i\frac{W}{k}]$. All the tuples necessary for a reducer to compare the tuple pairs assigned to it are sent to that reducer. Dedoop always achieves the optimal Y, but could be arbitrarily bad for X. For example, consider a single block of size n. The first reducer is responsible for tuple pairs numbered $[0, \frac{n(n-1)}{2k}]$. Since usually $n \gg k$, the first reducer is responsible for at least n tuple pairs. Since the first n tuple pairs contain all n tuples, the first reducer will have to receive all n tuples. In addition, in order for each mapper to decide which tuples to send to which reducer, and for each reducer to decide which tuple pairs to compare, Dedoop loads into the memory of each mapper and reducer a pre-computed data structure, called *Block Distribution Matrix*, which specifies the number of entities for every block every mapper processes. The size of the block distribution matrix is linear w.r.t. the number of blocks. Indeed, the largest number of blocks reported in evaluating Dedoop is less than 15,000 [75].

We compare the following distribution strategies: (1) Naive-Dedup is the naïve distribution strategy, where every block is assigned to one reducer; (2) PJ-Dedup is the proposed strategy where every block is distributed using the triangle distribution strategy, which is strictly better than applying existing parallel join algorithms [5, 83]; (3) Dis-Dedup is the proposed distribution strategy for a single blocking function, which is theoretically optimal; (4) Dedoop; and (5) Dis-Dedup$^+$, Naive-Dedup$^+$, PJ-Dedup$^+$, Dedoop$^+$ are extensions of Dis-Dedup, Naive-Dedup, PJ-Dedup, and Dedoop$^+$, respectively for multiple blocking functions. Dis-Dedup$^+$ is described in Section 4.4. Naive-Dedup$^+$, PJ-Dedup$^+$, and Dedoop$^+$ employ the same blocking function ordering technique to avoid comparing a tuple pair multiple times.

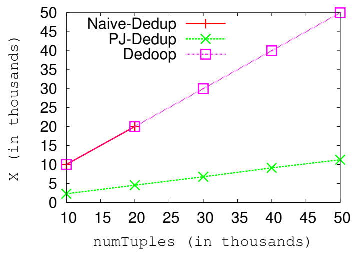
Since the distribution strategies are random, we run each experiment three times, and report the average. The time to compute the statistics for Dis-Dedup and DEDOOP is included in the time of Dis-Dedup and DEDOOP, respectively.

4.5.1 Single Block Deduplication Evaluation

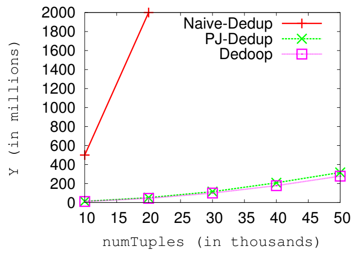
In this section, we compare Naive-Dedup, PJ-Dedup, and DEDOOP when performing a self-join on the synthetic dataset. We omit Dis-Dedup, since it is identical to PJ-Dedup when there is only one block.

Exp-1: Varying number of tuples. Figure 4.3 shows the parameters X, Y, and *time* for Naive-Dedup, PJ-Dedup, and DEDOOP for $k = 45$ reducers. We terminate a job after 6000 seconds. As we can see in Figure 4.3(c), Naive-Dedup exceeds this time limit after $30K$ tuples, since the computation occurs only in a single reducer. In terms of X, Figure 4.3(a) shows that PJ-Dedup achieves the best behavior, while for Naive-Dedup and DEDOOP X is equal to the number of tuples n. Indeed, Naive-Dedup uses only one reducer, and DEDOOP has one reducer that gets assigned the first $\frac{n(n-1)}{2k}$ tuple pairs, which need to access all n tuples. In terms of Y, as depicted in Figure 4.3(b), Naive-Dedup performs the worst. DEDOOP is slightly better than PJ-Dedup, since it distributes the comparisons evenly amongst all reducers, while PJ-Dedup has more work for reducers indexed (p, q) than reducers indexed (p, p).

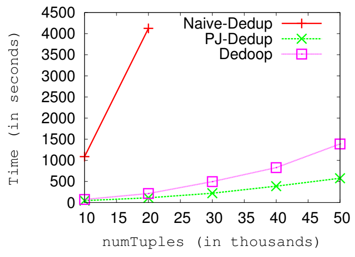
The running time of all three algorithms is depicted in Figure 4.3(c), and it shows that PJ-Dedup achieves the best performance. In fact, comparing Figure 4.3(a) with Figure 4.3(c), we can see that as the number of tuples increases, the gap between PJ-Dedup and DEDOOP in terms of the input size X grows, and so does the running time. This indicates that when Y is similar for PJ-Dedup and DEDOOP, the input size X becomes the differentiating factor.



(a) X



(b) Y



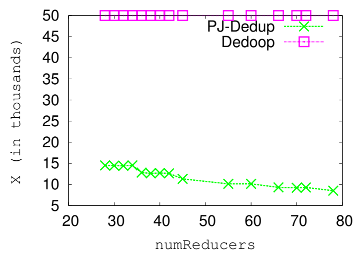
(c) Time

Figure 4.3: Varying number of tuples

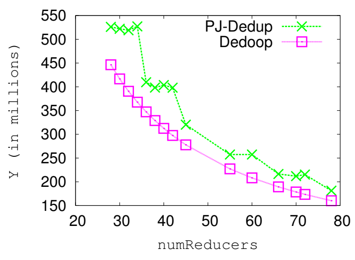
Exp-2: Varying number of reducers. Figure 4.4 shows the effect of the number of reducers on the parameters X, Y, and *time* for both PJ-Dedup and Dedoop. We fix the number of tuples to be $n = 50K$. The first observation is that PJ-Dedup outperforms Dedoop for any k, as shown in Figure 4.4(c). In fact, PJ-Dedup runs 2X faster than Dedoop. Second, both X and Y are decreasing for PJ-Dedup as k increases. However, the decrease is not smooth across the values of k and we can observe a few dips for $k = 36, 45, 55, 66, 78$. This behavior occurs because PJ-Dedup arranges the k reducers in a triangle. Since for certain values of k this is not possible, we choose then the largest subset of reducers that can be arranged into a triangle, and thus waste a small fraction of reducers.

Finally, as we can see from Figure 4.4(c), the running time improves as k increases for $k < 50$, and fluctuates as k increases when $k > 50$. This is because our cluster has a maximum number of 49 reducers being able to run in parallel. If $k > 50$, not all reducers can start at once; some reducers can only start after reducers in the first round finish.

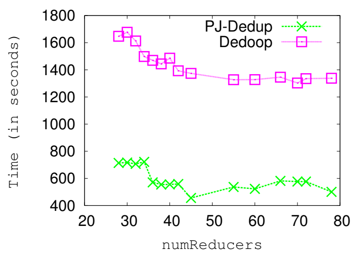
Memory. An important requirement for any MapReduce job is to have sufficient memory for each reducer. The maximum memory requirements for all algorithms are linear with respect to X, since X represents the maximum number of tuples that need to be stored in the heap space of a reducer. As we can see from Figure 4.3(a), PJ-Dedup requires much less memory than Dedoop. Indeed, for Dedoop at least one reducer needs to store all n tuples, while the heap space required for PJ-Dedup is $O(\frac{n}{\sqrt{k}})$, which not



(a) X



(b) Y



(c) Time

Figure 4.4: Varying number of reducers

only is much smaller than $O(n)$, but also decreases as k increases (this can be seen in Figure 4.4(a)).

4.5.2 Single Blocking Function Evaluation

In this section, we compare Naive-Dedup, PJ-Dedup, Dis-Dedup, and Dedoop for deduplication using a single blocking function on all three datasets.

Exp-3: Varying number of blocks. In this experiment, we fix the number of tuples per block to be 4, and then vary the number of blocks, using the synthetic dataset. As shown in Figure 4.5, Dedoop fails (heap space error) after 500,000 blocks. This is because Dedoop keeps in memory of every mapper the blocking distribution matrix, which grows as the number of blocks increases.

Dis-Dedup is identical to Naive-Dedup in this experiment, since all block sizes are the same, and hence no multi-reducer blocks exist. Dis-Dedup is better than PJ-Dedup, since the input size X for Dis-Dedup, which is $\frac{n}{k}$, is smaller than that for PJ-Dedup, which is $\frac{\sqrt{2}n}{\sqrt{k}}$.

Exp-4: Varying block size distribution. In order to test how different algorithms handle block-size skew, we vary the distribution of the block sizes by varying the parameter θ for the synthetic dataset, using $n = 20M$ tuples, and $5M$ blocks. Figure 4.6(a) shows that Dis-Dedup and Naive-Dedup send less tuples than PJ-Dedup, and Figure 4.6(b) shows that

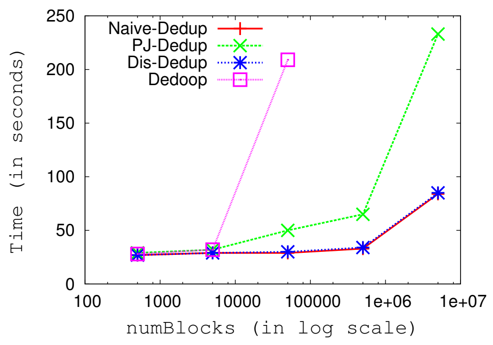
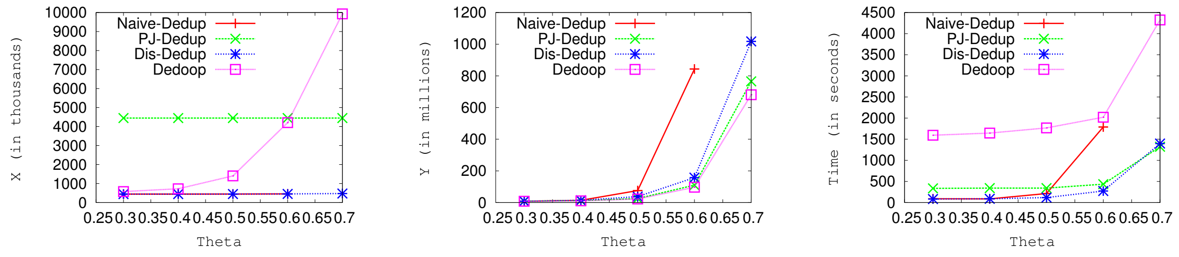


Figure 4.5: Varying number of blocks

the number of comparisons Y increases as the data becomes more skewed across all three algorithms. However, Y for PJ-Dedup grows at the lowest rate, and Y for Dis-Dedup is just a little worse than PJ-Dedup. In terms of running time, Figure 4.6(c) shows that when the data is not skewed ($\theta = 0.3, 0.4$), Naive-Dedup and Dis-Dedup perform the best, since X is now the differentiating factor. When the data becomes more skewed ($\theta = 0.5$), Dis-Dedup starts performing better than Naive-Dedup, which is still better than PJ-Dedup. As θ further increases ($\theta = 0.6, 0.7$), Y becomes the dominating factor in terms of running time. Thus, the running time for Naive-Dedup degrades fast, while Dis-Dedup and PJ-Dedup have similar performance. This observation supports our theoretical analysis that Dis-Dedup can adapt to all levels of skew, while Naive-Dedup and PJ-Dedup perform well only at one end of the spectrum. Note that Dedoop reports heap space error for this synthetic under default memory settings; nevertheless, we increase memory allocation for every mapper and reducer to 6G for Dedoop to compare with other distribution strategies. Observe that Dedoop has the worst running time in Figure 4.6(c) even though the X and Y of Dedoop are not the worst; this is mainly because (1) the number of concurrent mappers and reducers of Dedoop is less than that under the default setting due to the increased memory requirement of each mapper and reducer and (2) each mapper and reducer of Dedoop has an additional initializing cost of processing the block distribution matrix, whose size is linear w.r.t. the number of blocks.

Exp-5: Varying number of reducers. In this experiment, we vary the number of



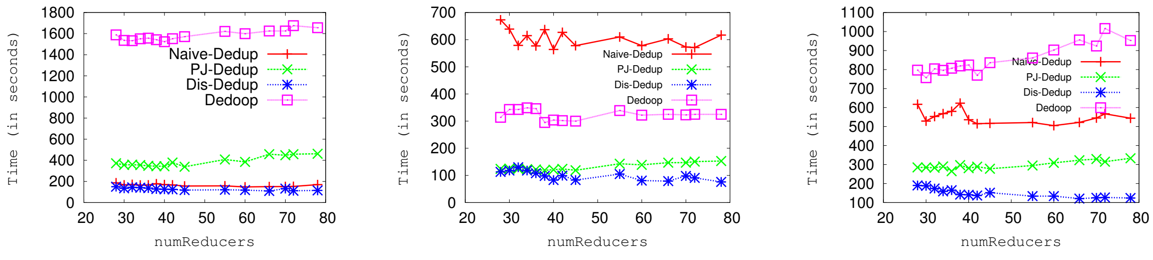
(a) X (b) Y (c) Time

Figure 4.6: Varying block size distribution

reducers k, and compare Naive-Dedup, PJ-Dedup, Dis-Dedup and Dedoop using all three datasets. For the synthetic dataset, we used $n = 20M$, $m = 5M$, and $\theta = 0.5$. Similar to the previous experiment, for Dedoop to run, we increase the memory allocation for every mapper and reducer to 6G for the OA and the synthetic datasets, and to 3G for CSX.

Figure 4.7 shows that Dis-Dedup is consistently the best algorithm for all three datasets, and any number of reducers. Dedoop only performs better than Naive-Dedup on CSX due to its small number of blocks, thus making the initializing cost of processing the block distribution matrix relatively cheap; but Dedoop has the worst performance in other two datasets for the same reasons as explained in Exp-4.

For the synthetic dataset, Naive-Dedup performs better than PJ-Dedup, while for the two real datasets the opposite behavior occurs, since the number of multi-reducer blocks in CSX and OA is bigger than that of the synthetic dataset. Another interesting trend to note here is that as the number of reducers k increases, the difference in running time between Dis-Dedup and PJ-Dedup also grows. The reason for this behavior is that X for PJ-Dedup is a $\sqrt{2k}$ factor away from the bound X_{low}, and thus dependent on k, while X for Dis-Dedup is only a constant factor 2 away.



(a) Synthetic (b) CSX (c) OA

Figure 4.7: Varying reducers on different datasets

4.5.3 Multiple Blocking Functions Evaluation

In this section, we compare the algorithms Naive-Dedup$^+$, PJ-Dedup$^+$, Dis-Dedup$^+$ and Dedoop$^+$, in the case of multiple blocking functions, using the two real datasets.

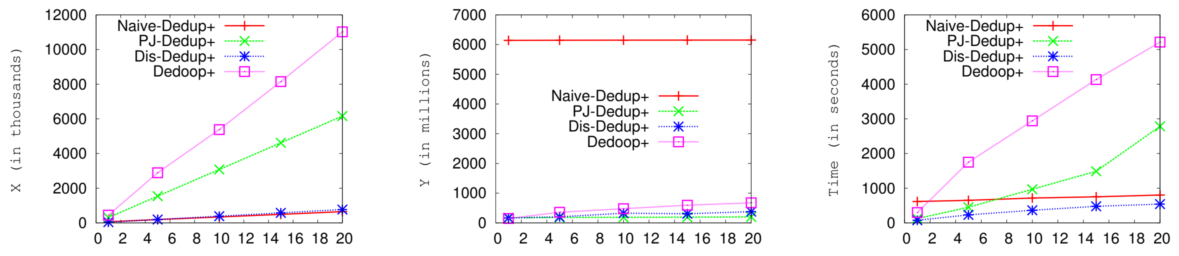
Dataset	1	5	10	15	20
CSX	667.45	710.64	741.52	768.51	784.40
OA	66.16	67.24	67.92	68.00	68.06

Table 4.3: Total number of comparisons W (in millions), for different number of blocking functions

Exp-6: Varying the number of blocking functions. Figure 4.8 shows the comparison using CSX. The input size X of all four algorithms increases linearly w.r.t. the number of blocking functions, as shown in Figure 4.8(a), while the output Y of all three algorithms increases very little, as shown in Figure 4.8(b). This behavior is observed because many tuple pairs generated by a blocking function have already been compared in previous blocking functions, and are thus skipped. Table 4.3 shows the total number of comparisons W for various numbers of blocking functions, indicating that the new tuple pair comparisons generated by 20 blocking functions is not much larger than the comparisons generated by 1 blocking function.

Figure 4.8(c) shows the running time comparison. Dedoop$^+$ performs the worst for multiple blocking functions due to multiple reasons: (1) Dedoop$^+$ has the worst X as shown in Figure 4.8(a); (2) Dedoop$^+$ has higher memory requirement, which limits the number of concurrent mappers and reducers, as explained in Exp-4; and (3) Dedoop$^+$ pays the extra cost of initiating s (≥ 1) MapReduce job to handle s (≥ 1) blocking functions, instead of using one MapReduce job to handle s blocking functions as Naive-Dedup$^+$, PJ-Dedup$^+$, and Dis-Dedup$^+$ do. The reason is again the high memory requirement of Dedoop$^+$; keeping the block distribution matrix produced by all s blocking functions in memory exceeds the memory limit for $s \geq 5$ even after we increase the memory allocation of mappers and reducers to 6G.

Dis-Dedup$^+$ achieves the best performance across any number of blocking functions. As the the number of blocking functions increases, the gap between Naive-Dedup$^+$ and Dis-Dedup$^+$ becomes smaller, while the gap between PJ-Dedup$^+$ and Dis-Dedup$^+$ becomes larger. The reason is that the multi-reducer blocks for s_1 blocking functions may become single-reducer blocks for $s_2 > s_1$ blocking functions as W becomes larger. Therefore, distributing those blocks to one reducer, as Naive-Dedup$^+$ does, instead of distributing them to multiple reducers, as PJ-Dedup$^+$ does, becomes more efficient. Varying number of blocking functions using the OA dataset shows similar results.



(a) X (b) Y (c) Time

Figure 4.8: Varying number of blocking functions

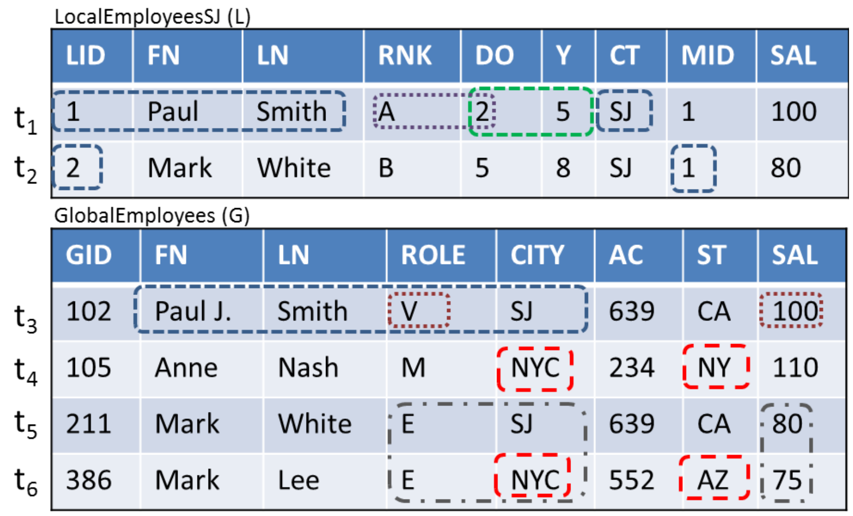
Chapter 5

Holistic Data Cleaning

As discussed in Chapter 2, various types of data quality rules have been proposed for detecting and repairing errors, and great efforts have been made to improve the effectiveness and efficiency of their cleaning algorithms. Currently existing data cleaning techniques often handle one type of errors at a time. While this approach minimizes the complexity of the problem, it does not consider the interaction between different types of rules, and thus can compromise the quality of the repairs due to the lack of end-to-end quality enforcement mechanism, as we show in this chapter.

Example 30: Consider the GlobalEmployees table (G for short) in Figure 5.1. Every tuple specifies an employee in a company with her id (GID), name (FN, LN), role, city, area code (AC), state (ST), and salary (SAL). We consider only two rules for now. The first is a functional dependency (FD) stating that the city values determine the values for the state attribute. We can see that cells in t_4 and t_6 present a violation for this FD: they have the same value for the city, but different states. We highlight the set S_1 of four cells involved in the violation in the figure. The second rule states that among employees having the same role, salaries in NYC should be higher. In this case cells in t_5 and t_6 are violating the rule, since employee Lee (in NYC) is earning less than White (in SJ). The set S_2 of six cells involved in the violation between Lee and White is also highlighted.

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


LocalEmployeesSJ (L)

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

GlobalEmployees (G)

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

Figure 5.1: Local (L) and Global (G) relations for employees data.

Previously proposed data repairing algorithms focus on repairing violations that belong to each class of constraints in isolation, e.g., FD violation repairs [21]. These techniques miss the opportunity of considering the interaction among different classes of constraints violations. For the example above, a desirable repair would update the city attribute for t_6 with a new value, thus only one change in the database would fix the two violations. On the contrary, existing methods would repair the FD by changing one cell in S_1, with an equal chance to pick any of the four by being oblivious to violations in other rules. In particular, most algorithms would change the state value for t_6 to NY or the state for t_4 to AZ. Similarly, rule based approaches, when dealing with application-specific constraints such as the salary constraint above, would change the salaries of t_5 or t_6 in order to satisfy the constraints. None of these choices would fix the mistake for city in t_6, on the contrary, they would add noise to the existing correct data.

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

We have already shown in Chapter 3 that denial constraints (DCs) are more general than existing constraint languages, including FDs and CFDs. Therefore, we aim at designing data repair algorithms for violations of DCs. Cleaning algorithms for DCs have been proposed before [13, 80], but they are limited in *scope*, as they repair numeric values only, in *generality*, as only a subclass of DCs is supported, and in the *cost model*, as they aim at minimizing the distance between original database and repair only. On the contrary, our proposal can repair any value involved in the constraints, we do not have limits on the allowed DCs, and we support multiple quality metrics (including cardinality minimality).

Example 31: The two rules described above can be expressed with the following DCs:

$o_1 : \forall t_\alpha, t_\beta \in G, \neg(t_\alpha.CITY = t_\beta.CITY \wedge t_\alpha.ST \neq t_\beta.ST)$

$o_2 : \forall t_\alpha, t_\beta \in G, \neg(t_\alpha.ROLE = t_\beta.ROLE \wedge t_\alpha.ROLE = \text{NYC} \wedge t_\beta.ROLE \neq \text{NYC} \wedge t_\alpha.SAL \leq t_\beta.SAL)$

The DC in o_1 corresponds to the FD in the global employee table: $CITY \rightarrow ST$ and has the usual semantics: if two tuples have the same value for city, they must have the same value for state, otherwise there is a violation. The DC in o_2 states that every time there are two employees with the same role, one in NYC and one in a different city, there is a violation if the salary of the second is greater than the salary of the first. □

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

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

- We introduce a compilation mechanism to project denial constraints on the current instance and capture the interaction among constraints as overlaps of the violations on the data instance. We compile violations into a Conflict Hypergraph (CH) which generalizes the one previously used in FD repairing [73] and is the first proposal to treat quality rules with different semantics and numerical operators in a unified artifact.

- We present a novel holistic repairing algorithm that repair all violations together w.r.t. one unified objective function. The algorithm is independent of the actual cost model and we present heuristics aiming at cardinality and distance minimality.

- We handle different repair semantics by using a novel concept of Repair Context (RC): a set of expressions abstracting the relationship among attribute values and the heterogeneous requirements to repair them. The RC minimizes the number of cells to be looked at, while guaranteeing soundness.

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

Outline. We formally define the data repair problem in Section 5.1, and we give an overview of the solution in Section 5.2. Technical details of the repair algorithms are discussed in Section 5.3. System optimizations are discussed in Section 5.4, while experiments are reported in Section 5.5.

5.1 Problem Definition

There exist multiple theoretic studies [21] and surveys [14, 45] 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.

Given as input a database I and a set of denial constraints Σ, we aim at finding a *repair* I_r of I such that $I_r \models \Sigma$ (consistency) and their distance *cost*(I_r, I) is minimized. We assume that there are no NULL values in I, and every cell in I contains a single value, so that there is no ambiguity when checking whether a DC is violated. 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. A popular *cost* function from the literature [21, 35] is the following:

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

where t' is the repair for tuple t and $dis_A(I(t[A]), I(t'[A]))$ is a distance between their values for attribute A (an exact match returns 0)[1]. There exist many similarity measurements for structured values (such as strings) and our setting does not depend on a particular approach, while for numeric values we rely on the squared Euclidian distance (i.e., the sum of the square of differences). It has been shown that finding a repair of minimal cost is NP-complete even for FDs only [21]. Moreover, minimizing the above function for DCs and numerical values only it is known to be a *MaxSNP*-hard problem [13]. If we rely on a binary distance between values (0 if they are equal, 1 otherwise), the above cost function corresponds to aiming at computing the repair with the minimal number of changes. The

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

problem of computing such *cardinality-minimal repairs* is known to be NP-hard to be solved exactly, even in the case with FDs only [73].

5.2 Solution Overview

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

5.2.1 System Architecture

The overall system architecture is depicted in Figure 5.2. Our system takes as input a relational database (Data) and a set of denial constraints (DCs), which express the data quality rules that have to be enforced over the input database.

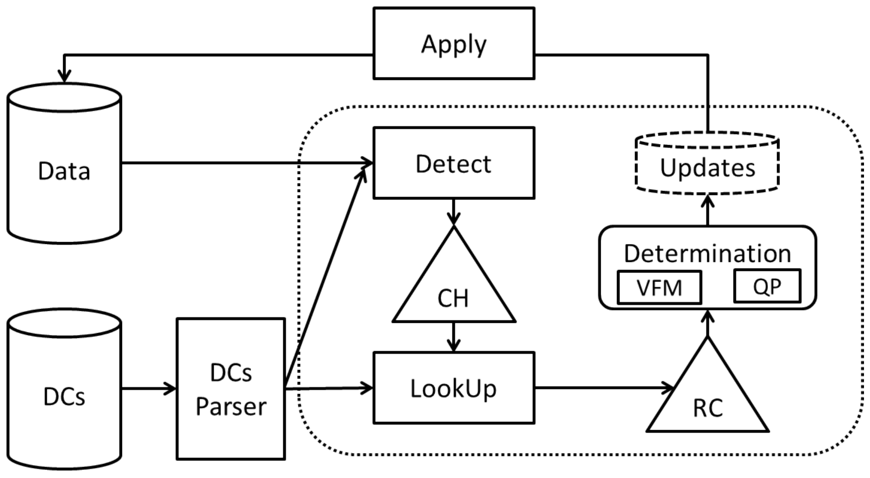


Figure 5.2: Architecture of the system.

Example 32: Consider the LocalEmployee table (L for short) in Figure 5.1. Every tuple represents information stored for an employee of the company in one specific location:

employee local id (LID), name (FN, LN), rank (RNK), number of days off (DO), number of years in the company (Y), city (CT), manager id (MID), and salary (SAL). LocalEmployee table and GlobalEmployee table constitute the input database. We introduce a third DC:

$o_3 : \forall t_\alpha, t_\beta \in L, t_\gamma \in G\ \neg(t_\alpha.LID \neq t_\beta.LID \wedge t_\alpha.LID = t_\beta.MID \wedge t_\alpha.FN \approx t_\gamma.FN \wedge t_\alpha.LN \approx t_\gamma.LN \wedge t_\alpha.CT = t_\gamma.CITY \wedge t_\gamma.ROLE \neq \mathrm{M})$

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

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

Example 33: Given the database in Figure 5.1, the DCs Parser processes constraint o_3 and provides the Detect module the rule to identify a violation spanning ten cells over tuples t_1, t_2, and t_3 as highlighted. Since every cell of this group is a possible error, DCs Parser dictates the LookUp module how to fix the violation if any of the ten cells is considered to be incorrect. For instance, the violation is repaired if "Paul Smith" is not the manager of "Mark White" in L (represented by the repair expression $(t_\alpha.LID \neq t_\beta.MID)$), if the employee in L does not match the one in G because of a different city $(t_\alpha.CT \neq t_\gamma.CITY)$, or if the role for the employee in G is updated to manager $(t_\gamma.ROLE = \mathrm{M})$. □

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

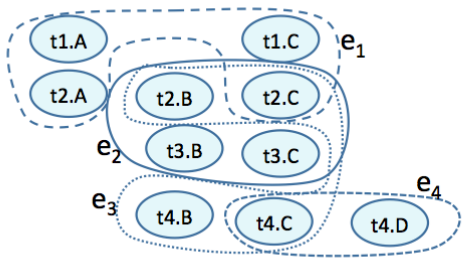
5.2.2 Violations Representation: Conflict Hypergraph

We represent the violations detected by the Detect module in a graph, where the nodes are the violating cells and the edges link cells involved in the same violation. As an edge can cover more than two nodes, we use a *Conflict Hypergraph* (CH) [73]. This is an undirected hypergraph with a set of nodes P representing the cells and a set of annotated hyperedges E representing the relationships among cells violating a constraint. More precisely, a *hyperedge* $(c;\ p_1, \ldots, p_n)$ is a set of violating cells such that one of them must change to repair the constraint, and contains: (a) the constraint c, which induced the conflict on the cells; (b) the list of nodes $p_1, \ldots, p_n$ involved in the conflict.

Example 34: Consider Relation R in Figure 5.3(a) and the following constraints (expressed as FDs and CFDs for readability): $\varphi_1 : A \rightarrow C$, $\varphi_2 : B \rightarrow C$, and $\varphi_3 : R[D = 5] \rightarrow R[C = 5]$. CH is built as in Figure 5.3(b): φ_1 has 1 violation e_1; φ_2 has 2 violations e_2, e_3; φ_3 has 1 violation e_4.

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

(a) Data



(b) CH

Figure 5.3: CH Example.

□

The CH represents the current state of the data w.r.t. the constraints. We rely on this representation to analyze the interactions among violations on the actual database. A hyperedge contains only violating cells: in order to repair it, at least one of its cells must get a new value. Interestingly, we can derive a repair expression for each of the cell involved in a violation, that is, for each variable involved in a predicate of the DC. Given a DC $\neg(P_1 \wedge \ldots \wedge P_m)$ and a set of violating cells (hyperedge) for it $V = \{v_1, \ldots, v_n\}$, for each $v_i \in V$ there is at least one *alternative* repair expression of the form $v_i \psi z$, where z is a constant or a connected cell in V.

A naïve approach to resolve the violations in all hyperedges is to compute the repair by fixing the hyperedges one after the other in isolation. This would lead to a valid repair, but, if there are interacting violations, it would certainly change more cells than the repair with minimal cost. As our goal is to minimize changes in the repair, we can rely on hyperedges for identifying cells that are very likely to be changed. The intuition here is that, in the spirit of [73], by using algorithms such as the Minimum Vertex Cover (MVC), we can identify at the global level what are the minimum number of violating cells to be changed in order to compute a repair.[2] For instance, a possible MVC for the CH in Figure 5.3(b) identifies $t_2[C]$ and $t_4[C]$.

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

5.2.3 Fixing Violation Holistically: Repair Context

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

Example 35: Consider the database and CH in Example 34. Suppose we have $t_2[C]$ and $t_4[C]$ from the MVC as starting points. We start from $t_2[C]$, which is involved in 3 hyperedges. Consider e_1: given $t_2[C]$ to change, the expression $t_2[C] = t_1[C]$ must be satisfied to solve it, thus bringing $t_1[C]$ into frontier. Cell $t_1[C]$ is not involved in

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

other hyperedges, so we stop. Similarly, $t_2[C] = t_3[C]$ must be satisfied to resolve e_2 and $t_3[C]$ is brought into the frontier. For e_3, $t_2[C] = t_4[C]$ is the expression to satisfy, however, $t_4[C]$ is involved also in e_4. We examine e_4 given $t_4[C]$ and we get another expression $t_4[C] = 5$. The resulting RC consists of frontier: $t_1[C], t_2[C], t_3[C], t_4[C]$, and repair expressions: $t_2[C] = t_1[C], t_2[C] = t_3[C], t_2[C] = t_4[C], t_4[C] = 5$.

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

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

Example 36: Consider again the L relation in Figure 5.1. Two DCs are defined to check the number of extra days off assigned to each employee:

$o_4 : \forall t_\alpha \in L, \neg(t_\alpha.RNK = \text{A} \wedge t_\alpha.DO < 3)$

$o_5 : \forall t_\alpha \in L, \neg(t_\alpha.Y > 4 \wedge t_\alpha.DO < 4)$

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

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

5.3 Holistic Data Cleaning Algorithm

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

5.3.1 The Iterative Algorithm

Given a database and a set of DCs, we rely on Algorithm 11 to perform holistic data cleaning. It starts by computing violations, the CH, and the MVC over it. These steps bootstrap the outer loop (lines 5–26), which is repeated until the current database is clean (lines 19–22) or a termination condition is met (lines 23–26). Cells in the MVC are ranked in order to favor those involved in more violations and are repaired in the inner loop (lines 8–18). In this loop, the RC for the cell is created with the LookUp procedure. When the RC is completed, the Determination step assigns the values to the cells that have a constant assignments in the repair expressions (e.g., $t_1[A] = 5$). Cells that do not have assignments with constants (e.g., $t_1[A] \neq 1$), keep their value and their repair is delayed to the next outer loop iteration. If the updates lead to a new database without violations, then it can be returned as a repair, otherwise the outer loop is executed again. If no new cells have been involved w.r.t. the previous loop, then the termination condition is triggered and the cells without assignments are updated with new fresh values in the post processing final step.

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

Algorithm 11 `Holistic Repair`

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

the inner loop and a new version of the CH is created. This CH has new cells involved in violations and therefore the termination condition is not met.

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

Example 37: Consider again only rules o_1 and o_2 in the running example. After the first inner loop iteration, the RC contains an assignment $t_6[\text{CITY}] \neq$ "NYC", which is not enforced by the determination step and therefore the database does not change. The HC is created again (line 19) and it still has violations for o_1 and o_2. The cells involved in the two violations go into *tempCells* and *sizeAfter* is set to 9. A new outer loop iteration sets *sizeBefore* to 9, the inner loop does not change the data, and it gets again the same graph at line 19. As *sizeBefore* = *sizeAfter*, it exits the outer loop and the post processing assigns $t_6[\text{CITY}] =$ "FV". □

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

Though Algorithm 11 is sound, it may not generate the optimal repair, such as the repair with the minimal number of changes, as illustrated in the following example.

Example 38: Consider again Example 34. We showed a repair with four changes obtained with our algorithm, but there exists a cardinality minimal repair with only three changes: $t_1[C] = 3, t_2[C] = 3, t_4[D] = NV$. □

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

5.3.2 Detect: Identifying Violations

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

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

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

5.3.3 LookUp: Building the Repair Context

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

predicate in dcs	$=$	$\neq$	$>$	$>=$	$<$	$<=$	$\approx_t$
predicate in repair exps	$\neq$	$=$	$<=$	$<$	$>=$	$>$	$\neq_t$

Table 5.1: Table of conversion of the predicates in a DC for their repair. Predicate $\neq_t$ states that the distance between two strings must be greater than t.

As $p \in A_\phi(c)$, then it is required that $r : p\phi^c c$, where ϕ^c is the predicate converted as described in Table 5.1. Variable c can be a constant or another cell. For denial constraints, we defined a function *DC.Repair(e,c)*, based on the above rule, which automatically generates a repair expression for a hyperedge e and a cell c. We first show an example of its output when constants are involved in the predicate and then we discuss the case with variables.

Example 39: Consider the constraint o_2 from the example. We show below two examples of repair expressions for it.

$$\begin{aligned}&\mathit{DC.Repair}((o_2; t_5[\text{ROLE}], t_5[\text{CITY}], t_5[\text{SAL}], \ldots, t_6[\text{SAL}]),\\ &\qquad t_5[\text{ROLE}]) = \{t_5[\text{ROLE}] \neq \text{“}E\text{”}\}\\ &\mathit{DC.Repair}((o_2; t_5[\text{ROLE}], t_5[\text{CITY}], t_5[\text{SAL}], \ldots, t_6[\text{SAL}]),\\ &\qquad t_6[\text{SAL}]) = \{t_6[\text{SAL}] \geq 80\}\end{aligned}$$

In the first repair expression the new value for t_5[ROLE] must be different from “E” to solve the violation. The repair expression does not state that its new value should be different from the active domain of ROLE (i.e., $t_5[\text{ROLE}] \neq \{\text{“E”}, \text{“S”}, \text{“M”}\}$), because in the next iteration of the outer loop it is possible that another repair expression imposes t_5[ROLE] to be equal to a constant already in the active domain (e.g., a MD used for entity resolution). If there is no other expression suggesting values for t_5[ROLE], in a following step the termination condition will be reached and the post-process will assign a fresh value to t_5[ROLE]. □

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

Algorithm 12 LookUp

```
Require: Cell cell, Hyperedge edge, Repair Context rc
Ensure: updated rc
1: exps $\leftarrow$ $Denial.repair(edge, cell)$
2: frontier $\leftarrow$ $exps.getFrontier()$
3: for all cell $\in$ frontier do
4:     edges $\leftarrow$ $cell.getEdges()$
5:     for all edge $\in$ edges do
6:         exps $\leftarrow$ exps $\cup$ LookUp(cell,edge,rc).$getExps()$
7: rc.$update(exps)$
```

Algorithm 12 describes how, given a cell c and a hyperedge e, LookUp processes recursively in order to move from a single repair expression for c to a *Repair Context*.

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

Example 40: Consider the constraint o_3 from Example 32 and an additional DC o_6 : $\forall t_\alpha \in G, \neg(t_\alpha.ROLE = \text{V} \wedge t_\alpha.SAL < 200)$, that is, a vice-president cannot earn less than 200. Given t_3[ROLE] as input, LookUp processes the two edges over it and collects the repair expressions t_3[ROLE] $\neq$ "V" from o_6 and t_3[ROLE] $=$ "M" from o_3. □

5.3.4 Determination: Finding Valid Assignments

Given the set of repair expressions collected in the RC, the Determination function returns an assignment for the frontier in the RC. The process for the determination is depicted in Algorithm 13: Given an RC and a starting cell, we first choose a (maximal) subset of the repair expressions that is satisfiable, then we compute the value for the cells in the frontier aiming at minimizing the cost function, and update the database accordingly later.

Algorithm 13 DETERMINATION

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

In Algorithm 13, we have two determination procedures. One is Value Frequency Map (VFM), which deals with string typed expressions. The other is quadratic programming (QP), which deals with numerical typed expressions[3].

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

Example 41: Consider the example in Figure 5.1 and two new DCs:

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

$o_7 : \forall t_\alpha \in L, \neg(t_\alpha.RNK = \text{B} \wedge t_\alpha.DO > 4)$

$o_8 : \forall t_\alpha \in L, \neg(t_\alpha.Y > 7 \wedge t_\alpha.DO < 6)$

That is, an employee after 8 years should have at least 6 extra days off, and an employee of rank "B" cannot have more than 4 days. Given $t_2[\text{DO}]$ as input by the MVC, LookUp processes the two edges over it and collects the repair expressions $t_2[\text{DO}] \leq 4$ from o_7 and $t_2[\text{DO}] \geq 6$ from c_8. The satisfiability test fails $(x \leq 4 \wedge x \geq 6)$ and the *computeSatisfiable* function starts removing expressions from the RC, in order to maximize the set of satisfiable constraints. In this case, it removes o_7 from the RC and sets $t_2[\text{DO}] = 6$ to satisfy o_8. Violation for o_7 is pushed to the outer loop, and, as in the new MVC there are no new cells involved, the post processing step updates $t_2[\text{RNK}]$ to a fresh value. □

2) Function getAssignments. After getting the maximum number of solvable expressions, the following step aims at computing an optimal repair according to the cost model at hand. We therefore distinguish between string typed expressions and numerical typed expressions for both cost models: cardinality minimality and distance minimality

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

Example 42: Consider a schema $R(A, B)$ with 5 tuples $t_1 = R(a, b), t_2 = R(a, b), t_3 = R(a, cde), t_4 = R(a, cdf), t_5 = R(a, cdg)$. R has an $FD : A \rightarrow B$. Suppose now we have an RC with set of expressions $t_1[B] = t_2[B] = t_3[B] = t_4[B] = t_5[B]$. VFM is created with $b \rightarrow 2, cde \rightarrow 1, cdf \rightarrow 1, cdg \rightarrow 1$. So value b is chosen. □

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

Example 43: Consider the same database as Example 42. String cardinality minimality is not necessarily string distance minimality. Now VFM is created as follows: $b \rightarrow 12, cde \rightarrow 10, cdf \rightarrow 10, cdg \rightarrow 10$. So any of cde, cdf, cdg can be chosen. □

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

Example 44: Consider a schema $R(A, B, C)$ with a tuple $t_1 = R(0, 3, 2)$ and the two repair expressions: $r_1 : R[A] < R[B]$ and $r_2 : R[B] < R[C]$. To find valid assignments, we want to minimize the quadratic objective function $(x-0)^2 + (y-3)^2 + (z-2)^2$ with two linear constraints $x < y$ and $y < z$, where x, y, z will be new values for $t_1[A], t_1[B], t_1[C]$. We get solution $x = 1, y = 2, z = 3$ with the value of objective function being 3. $\square$

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

Example 45: Consider the same database as in Example 44. Numerical distance minimality is not necessary numerical cardinality minimum. It can be easily spotted that $x = 0, y = 1, z = 2$ whose squared distance is 4 only has one change, while $x = 1, y = 2, z = 3$ whose squared is 3 has three changes. $\square$

5.4 Optimizations and Extensions

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

Detection Optimization. Violation detection for DCs checks every possible grounding of predicates in denial constraints. Thus improving the execution times for violation detection implies reducing the number of groundings to be checked. We face the issue by verifying predicates in a order based on their selectivity. Before enumerating all grounding combinations, predicates with constants are applied first to rule out impossible groundings. Then, if there is an equality predicate without constants, the database is partitioned according to two attributes in the equality predicate, so that grounding from two different partitions need not to be checked. Consider for example o_3. The predicate $(t_\gamma.ROLE \neq \mathrm{M})$ is applied first to rule out grounding with attribute $t_\gamma.ROLE$ equals M. Then predicate $(t_\alpha.LID = t_\beta.MID)$ is chosen to partition the database, so groundings with values of attributes $t_\alpha.LID$ and $t_\beta.MID$ not being in the same partition will not be checked.

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

Custom Repair Strategy. The default repair strategy can easily be personalized with a user interface for the LookUp module. For example, a user might want to enforce the increase of the salary for the NYC employee whenever there is a violatino of the rule o_2. We have shown how repair expressions can be obtained automatically for DCs. In general, the *Repair* function can be provided for any new kind of constraints that is plugged to the system. In case the function is not provided, the system would only detect violating cells with the Detect module. The iterative algorithm will try to fix the violation with repair expressions from other interacting constraints or, if it is not possible, it will delay its repair until the post-processing step.

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

5.5 Experimental Study

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

5.5.1 Experimental Settings

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

- HOSP, is from US Department of Health & Human Services [5]: HOSP has 100K tuples with 19 attributes and we designed 9 FDs for it.
- CLIENT [13], has 100K tuples, 6 attributes over 2 relations, and 2 DCs involving numerical values.
- EMP is a synthetic data that has two relations and 17 attributes in total. We designed 6 DCs for it, including a DC that involves three tuples from the two relations. For the complete list of DCs, please refer to the paper [31].

Errors in the datasets have been produced by introducing noise with a certain rate, that is, the ratio of the number of dirty cells to the total number of cells in the dataset. An error rate $e\%$ indicates that for each cell, there is a $e\%$ probability we are going to change that

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

[5]http://www.hospitalcompare.hhs.gov/

cell. In particular, we update the cells containing strings by randomly picking a character in the string, and change it to “X”, while cells with numerical values are updated with randomly changing a value from an interval.[6]

Algorithms. The techniques presented in the paper have been implemented in Java. As our holistic cleaning algorithm is modular with respect to the cost function that the user wants to minimize, we implemented the two semantics discussed in Section 5.3. In particular we tested the *getAssigment* function both for cardinality minimality (*RC-C*) and for the minimization of the distance (*RC-D*).

We implemented also the following algorithms in Java: the FD repair algorithms from [15] (*Sample*), [21] (*Greedy*), [73] (*VC*) for HOSP; and the DC repair algorithm from [13] (*MWSC*) for CLIENT. As there is no available algorithm able to repair all the DCs in EMP, we compare our approach against a sequence of applications of other algorithms (*Sequence*). In particular, we ran a combination of three algorithms: *Greedy* for DCs o_1, *MWSC* for o_2, o_4, o_5, o_6, and a simple, ad-hoc algorithm to repair o_3 as it is not supported by any of the existing algorithms. In particular, for o_3 we implemented a simplified version of our Algorithm 11, without MVC and with violations fixed one after the other without looking at their interactions. As there are six alternative orderings, we executed all of them for each test and picked the results from the combination with the best performance. For $\approx_t$ we used string edit distance with $t = 3$: two strings were considered similar if the minimum number of single-character insertions, deletions and substitutions needed to convert a string into the other was smaller than 4.

Metrics. We measure performance with different metrics, depending on the constraints involved in the scenario and on the cost model at hand. The number of changes in the repair is the most natural measure for cardinality minimality, while we use the cost function in Section 5.1 to measure the distance between the original instance and its repair. Moreover, as the ground truth for these datasets is available, to get a better insight about repair quality we measured also precision (P, corrected changes in the repair), recall (R, coverage of the errors introduced with $e\%$), and F-measure ($F = 2 \times (P \times R)\ (P + R)$). Finally, we measure the execution times needed to obtain a repair.

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

As in [15], we count as *correct changes* the values in the repair that coincide with the values in the ground truth, but we count as a fraction (0.5) the number of *partially correct changes*: changes in the repair which fix dirty values, but their updates do not reflect the values in the ground truth. It is evident that fresh values will always be part of the partially correct changes.

All experiments were conducted on a Win7 machine with a 3.4GHz Intel CPU and 4GB of RAM. Gurobi Optimizer 5.0 has been used as the external QP tool [62] and all computations were executed in memory. Each experiment was run 6 times, and the results for the best execution are reported. We decided to pick the best results instead of the average in order to favor *Sample*, which is based on a sampling of the possible repairs and has no guarantee that the best repair is computed first.

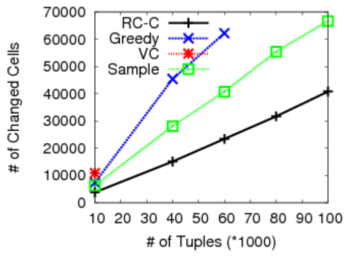
5.5.2 Experimental Results

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

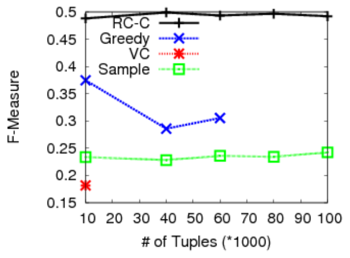
Exp-1: FDs only. In the first set of experiments we show that the holistic approach has benefits even when the constraints are all of the same kind, in this case FDs. As in this example all the alternative approaches consider some kind of cardinality minimality as a goal, we ran our algorithm with the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

Figure 5.4 reports results on the quality of the repairs generated for the HOSP data with four systems. Our system clearly outperforms all alternatives in every quality measure. This verifies that holistic repairs are more accurate than alternative fixes. The low values for the F-measure are expected: even if the precision is very high (about 0.9 for our approach on 5% error rate), recall is always low because many randomly introduced error cannot be detected. Consider for example R(A,B), with an FD: $A \rightarrow B$, and two tuples R(1,2), R(1,3). An error introduced for a value in A does not trigger a violation, as there is not match in the left hand side of the FD, thus the erroneous value cannot be repaired.

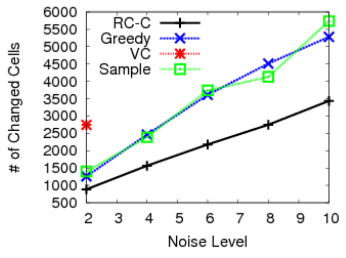
Figure 5.4(c) shows the number of cells changed to repair input instances (of size 10K tuples) with increasing amounts of errors. The number of errors increases when $e\%$ increases for all approaches; however, *RC-C* benefits of the holism among the violations and is less sensitive to this parameter.



(a) # changes



(b) F-Measure



(c) % errors

Figure 5.4: Compare data repairing on HOSP

Execution times are reported in Figure 5.5, we set a timeout of 10 minutes and do not report executions over this limit. We can notice that our solution competes with the fastest algorithm and scales nicely up to large databases. We can also notice that *VC* does not scale to large instances due to the large size of their hypergraph, while our optimizations effectively reduces the number of hyperedges in *RC-C*.

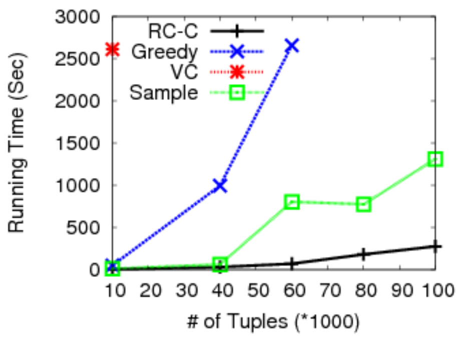
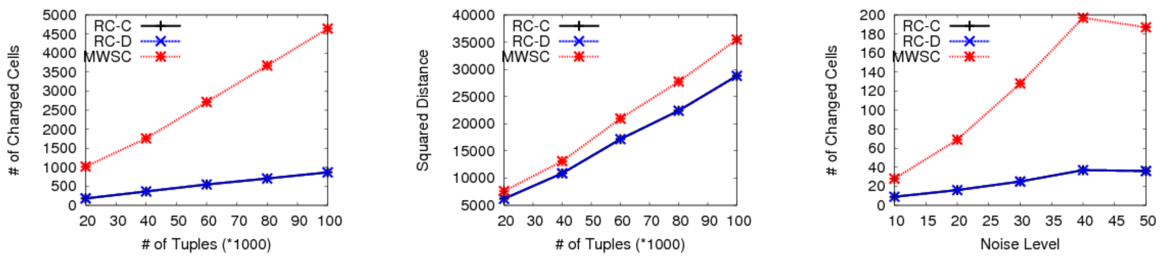


Figure 5.5: HOSP Exec. time

Exp-2: DCs with numerical values. In the experiment for CLIENT data, we compare our solution against the state-of-the-art for the repair of DCs with numerical values (*MWSC*) [13]. As *MWSC* aims at minimizing the distance in the repair, we ran the two versions of our algorithm (*RC-C* and *RC-D*).

Figure 5.6 shows that *RC-C* and *RC-D* provide more precise repairs, both in terms of number of changes and distance, respectively. As in Exp-1, the holistic approach shows significant improvements over the state-of-the-art even with constraints of the same kind only, especially in terms of cardinality minimality. This can be observed also with data with increasing amount of errors in Figure 5.6(c). Notice that *RC-C* and *RC-D* have very similar performances for this example. This is due to the fact that the dataset was designed for *MWSC*, which supports only local DCs. For this special class the cardinality minimization heuristic is not needed in order to obtain minimality.

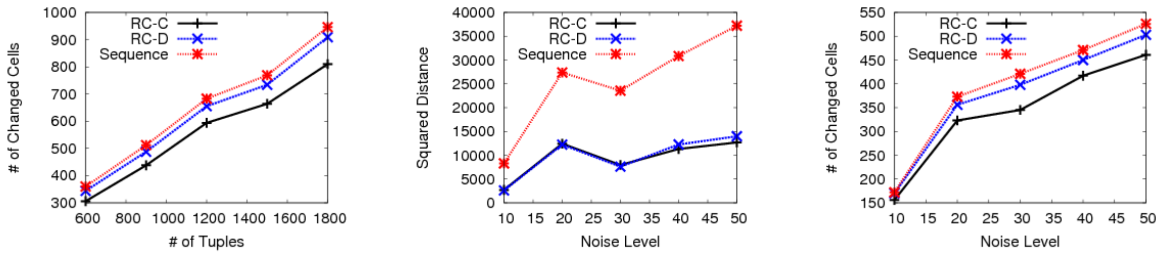


(a) # changes (b) Distance (c) % errors

Figure 5.6: Compare data repairing on CLIENT

In terms of execution time, the overheads for *RC-C* and *RC-D* are really small and the execution times for them are comparable to *MWSC*.

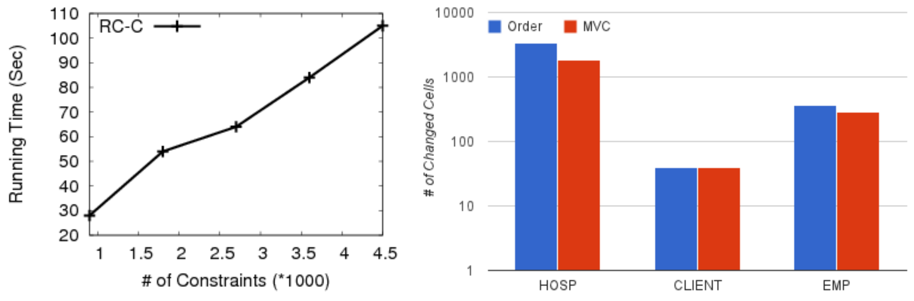
Exp-3: Heterogeneous DCs. In the experiments for the EMP dataset, we compare *RC-C* and *RC-D* against *Sequence*. In this dataset we have more complex DCs and, as expected, Figures 5.7(a) and 5.7(c) show that *RC-C* performs best in terms of cardinality minimality. Figure 5.7(b) reports that both *RC-C* and *RC-D* perform significantly better than *Sequence* in terms of Distance cost. We observe that all approaches had low precision in this experiment: this is expected when numerical values are involved, as it is very difficult for an algorithm to repair a violation with exactly the correct value. Imagine an example with value x violating $x > 200$ and an original, correct value equals to 250; in order to minimize the distance from the input, value x is assigned 201 and there is a significant distance w.r.t. the true value.



(a) # changes (b) Distance (c) % errors

Figure 5.7: Compare data repairing on Emp

The three algorithms have the same time performances. This is not surprising, as they share the detection of the violations which is by far the most expensive operation due to the presence of a constraint involving three tuples. The cubic complexity for the detection of the violations clearly dominates the computation.



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

Figure 5.8: Results varying the number of constraints and the ordering criteria in Algorithm 1.

Exp-4: Number of Rules. In order to test the scalability of our approach w.r.t. the number of constraints, we generated DCs for the HOSP dataset and tested the performance of the system. New rules have been generated as follows: randomly take one FD c from the original constraints for HOSP, one of its tuples t from the ground truth, and create a CFD c', such that all the attributes in c must coincide with the values in t (e.g., c' : Hosp[Provider#=10018] $\rightarrow$ Hosp[Hospital="C. E. FOUNDATION"]). We then generated an instance of 5k tuples with 5% error rate and computed a repair for every new set of DCs. For each execution, we increased the number of constraints as input. The results in Figure 5.8(a) verifies that the execution times increase linearly with the number of constraints.

Exp-5: MVC contribution. In order to show the benefits of MVC on the quality of repair, we compared the use of MVC to identify conflicting cells versus a simple ordering based on the number of violations a cell is involved (Order). For the experiment we used datasets with 10k tuples, 5% error rate and *RC-C*. Results are reported in Figure 5.8(b). For the hospital dataset the number of changes is almost the double with the simple ordering (3382 vs 1833), while the difference is smaller for the other two experiments because they show fewer interactions between violations.

Chapter 6

Conclusion and Future Work

In this chapter, we conclude the dissertation and discuss the future work.

6.1 Conclusion

In this dissertation, we tackled various challenges associated with the three steps of qualitative data cleaning, namely, the rule mining step, the error detection step, and the error repair step.

We showed how to automatically mine data quality rules expressed in the formalism of denial constraints (DCs). We choose DCs as the formal integrity constraint language for capturing data quality rules because it is able to capture many real-life data quality rules, and at the same time it allows for efficient discovery algorithm, namely, the FASTDC algorithm as we have described. The main insight of FASTDC is that it transforms the problem of discovering DCs to the problem of searching for minimal set covers of the evidence set, a data structure built from the input data, and hence avoiding enumerating all candidate DCs. FASTDC also contains several novel pruning optimizations, some of which leverage the axioms we developed for DCs. We also show how FASTDC can be adapted to discover approximate DCs from a dirty dataset. As the number of discovered DCs can be large and some of them might not be correct due to overfitting, we provide an interestingness measure, which includes succinctness and coverage, to rank the discovered DCs.

We presented a distribution strategy that distributes the error detection workload evenly to a cluster of machines in a parallel shared-nothing computing environment. The distribution strategy was described in the context of detecting duplicate records; however, it is readily applicable to any error detection workload that requires a tuple pairwise comparison. Our proposed distribution strategy aims at minimizing the maximum computation cost and the maximum communication cost, which are the two main types of cost one needs to consider in a shared-nothing environment. Both costs of our proposed strategy are guaranteed to be within a small constant factor from the lower bounds. We also showed how our strategy can be used to evenly distribute a data deduplication workload when blocking techniques are used to avoid exhaustively compare all tuple pairs.

We proposed a holistic data cleaning technique, which accumulates evidences from a broad spectrum of data quality rules, and suggests possible data updates in a holistic manner. Compared with previous piece-meal data repairing approaches, the holistic approach produces data updates with higher accuracy because it compiles different errors into one representation, namely, the conflict hypergraph, and aims at suggesting data updates that can fix as many errors as possible.

6.2 Future Work

As discussed in Chapter 1, error detection techniques can be generally classified into either qualitative or quantitative. This dissertation focused on qualitative error detection techniques. In the future, we plan to investigate techniques to detect quantitative errors.

Data is increasingly generated by sensors, mobile devices, and automated processes in a variety of domains, including manufacturing, transportation, health care, and smart homes. Because the majority of this data reflects normal case operation, it is generally not productive for analysts to exhaustively explore every value. Instead, a natural approach is to look for unusual values and outliers, namely, quantitative errors. Example applications include equipment monitoring (e.g., detecting faulty manufacturing components), intrusion detection (e.g., building break-ins), environmental observation (e.g., occupancy and HVAC control), and customer service (e.g., finding customers experiencing poor service). We plan to investigate techniques to detect quantitative errors in two directions.

First, we plan to investigate *contextual outliers*; sometimes a data point may not be an outlier compared with the rest of the data set but is an outlier when compared with a subgroup of the data. In our discussions with data analysts in multiple large enterprises, these contextual outliers are useful in two important application scenarios: data exploration and targeted error explanation and diagnosis.

Second, we also intend to discover errors in time series and sequence data generated by sensors, machine logs, and other automated processes (also known as the Internet of Things). In particular, we would like to discover abnormal state transition by first identifying the underlying states the sensor readings represents and then flagging unlikely state transitions.

References

[1] Apache hadoop. http://hadoop.apache.org.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Data profiling: A tutorial. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1747–1751, 2017.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Foto N Afrati and Phokion G Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *Proc. 12th Int. Conf. on Database Theory*, pages 31–41, 2009.

[5] Foto N Afrati and Jeffrey D Ullman. Optimizing joins in a map-reduce environment. In *Proc. 13th Int. Conf. on Extending Database Technology*, pages 99–110, 2010.

[6] Charu C. Aggarwal. *Outlier Analysis*. Springer, 2013.

[7] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 207–216, 1993.

[8] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[9] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 586–597, 2002.

[10] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 68–79, 1999.

[11] Marianne Baudinet, Jan Chomicki, and Pierre Wolper. Constraint-generating dependencies. *J. Comput. Syst. Sci.*, 59(1):94–115, 1999.

[12] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In Richard Hull and Martin Grohe, editors, *Proc. 33rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 212–223. ACM, 2014.

[13] Leopoldo Bertossi, Loreto Bravo, Enrico Franconi, and Andrei Lopatenko. Complexity and approximation of fixing numerical attributes in databases under integrity constraints. In *Proc. 10th Int. Workshop on Database Programming Languages*, 2005.

[14] Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[15] George Beskales, Ihab F Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *Proc. VLDB Endowment*, 3(1-2):197–207, 2010.

[16] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. On the relative trust between inconsistent data and inaccurate constraints. pages 541–552, 2013.

[17] George Beskales, Ihab F Ilyas, Lukasz Golab, and Artur Galiullin. Sampling from repairs of conditional functional dependency violations. *VLDB J.*, 23(1):103–128, 2014.

[18] George Beskales, Mohamed A Soliman, Ihab F Ilyas, and Shai Ben-David. Modeling and querying possible repairs in duplicate detection. volume 2, pages 598–609. VLDB Endowment, 2009.

[19] Mikhail Bilenko, Beena Kamath, and Raymond J Mooney. Adaptive blocking: Learning to scale up record linkage. In *Proc. 2006 IEEE Int. Conf. on Data Mining*, pages 87–96, 2006.

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

[21] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2005.

[22] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proc. 23rd Int. Conf. on Data Engineering*, pages 746–755, 2007.

[23] Anup Chalamalla, Ihab F Ilyas, Mourad Ouzzani, and Paolo Papotti. Descriptive and prescriptive data cleaning. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 445–456, 2014.

[24] Fei Chiang and Renée J. Miller. Discovering data quality rules. *Proc. VLDB Endowment*, 1(1):1166–1177, 2008.

[25] Fei Chiang and Renée J. Miller. A unified model for data and constraint repair. In *Proc. 27th Int. Conf. on Data Engineering*, pages 446–457, 2011.

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

[27] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. on Knowl. and Data Eng.*, 24(9):1537–1555, September 2012.

[28] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 63–78, 2015.

[29] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data deduplication. *Proc. VLDB Endowment*, 9(11):864–875, 2016.

[30] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *Proc. VLDB Endowment*, pages 1498–1509, 2013.

[31] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *Proc. 29th Int. Conf. on Data Engineering*, pages 458–469, 2013.

[32] Xu Chu, Ihab F. Ilyas, Paolo Papotti, and Yin Ye. Ruleminer: Data quality rules discovery. In *Proc. 30th Int. Conf. on Data Engineering*, pages 1222–1225, 2014.

[33] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. pages 1247–1261, 2015.

[34] Samuel Clemens. 7 facts about data quality. *InsightSquared*, 2012.

[35] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *Proc. VLDB Endowment*, pages 315–326, 2007.

[36] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. NADEEF: a commodity data cleaning system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 541–552, 2013.

[37] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., 2003.

[38] Tamraparni Dasu, Theodore Johnson, S. Muthukrishnan, and Vladislav Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 240–251, 2002.

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

[40] 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.

[41] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. 18th Int. Conf. on Very Large Data Bases*, pages 27–40, 1992.

[42] Xin Luna Dong and Felix Naumann. Data fusion: resolving data conflicts for integration. *Proc. VLDB Endowment*, 2(2):1654–1655, 2009.

[43] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. and Data Eng.*, 19(1):1–16, 2007.

[44] Grace Fan, Wenfei Fan, and Floris Geerts. Detecting errors in numeric attributes. In *Web-Age Information Management*, pages 125–137. Springer, 2014.

[45] Wenfei Fan and Floris Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[46] Wenfei Fan, Floris Geerts, and Xibei Jia. A revival of integrity constraints for data cleaning. *Proc. VLDB Endowment*, 1(2):1522–1523, 2008.

[47] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. and Data Eng.*, 23(5):683–698, 2011.

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

[49] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *Proc. VLDB Endowment*, 2(1):407–418, 2009.

[50] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Interaction between record matching and data repairing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 469–480. ACM, 2011.

[51] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2), 2012.

[52] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 371–380, 2001.

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

[54] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The llunatic data-cleaning framework. *Proc. VLDB Endowment*, 6(9):625–636, 2013.

[55] Lise Getoor and Ashwin Machanavajjhala. Entity resolution: theory, practice & open challenges. *Proc. VLDB Endowment*, 5(12):2018–2019, 2012.

[56] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F Naughton, Narasimhan Rampalli, Jude Shavlik, and Xiaojin Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 601–612, 2014.

[57] Lukasz Golab, Howard J. Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. *Proc. VLDB Endowment*, 1(1):376–390, 2008.

[58] Daniel M. Gordon, Greg Kuperberg, and Oren Patashnik. New constructions for covering designs. *J. COMBIN. DESIGNS*, 3(269–284), 1995.

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

[60] Sergio Greco, Cristian Molinaro, and Francesca Spezzano. Incomplete data and data dependencies in relational databases. *Synthesis Lectures on Data Management*, 2012.

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

[62] Gurobi. Gurobi optimizer reference manual, 2012.

[63] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, et al. Demonstration of the myria big data management service. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 881–884. ACM, 2014.

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

[65] Mauricio A Hernández and Salvatore J Stolfo. The merge/purge problem for large databases. *ACM SIGMOD Rec.*, 24(2):127–138, 1995.

[66] Thomas N Herzog, Fritz J Scheuren, and William E Winkler. *Data Quality and Record Linkage Techniques.* Springer Science & Business Media, 2007.

[67] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[68] Ihab F. Ilyas and Xu Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[69] Piotr Indyk. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, 38(1):84–90, 2001.

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

[71] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Human Factors in Computing Systems (CHI)*, 2011.

[72] K. Kerr, T. Norris, and R. Stockdale. Data quality information and decision making: a healthcare case study. In *ACIS*, pages 5–7, 2007.

[73] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proc. 12th Int. Conf. on Database Theory*, 2009.

[74] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: efficient deduplication with hadoop. *Proc. VLDB Endowment*, 5(12):1878–1881, 2012.

[75] Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. In *Proc. 28th Int. Conf. on Data Engineering*, pages 618–629, 2012.

[76] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493, 2010.

[77] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. Metric functional dependencies. In *Proceedings of the 25th International Conference on Data Engineering*, pages 1275–1278, 2009.

[78] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. Record linkage: similarity measures and algorithms. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 802–803, 2006.

[79] Andrei Lopatenko and Leopoldo 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.

[80] Andrei Lopatenko and Loreto Bravo. Efficient approximation algorithms for repairing inconsistent databases. In *Proc. 23rd Int. Conf. on Data Engineering*, pages 216–225, 2007.

[81] Alexandra Meliou, Wolfgang Gatterbauer, Suman Nath, and Dan Suciu. Tracing data errors with view-conditioned causality. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 505–516, 2011.

[82] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Synthesis Lectures on Data Management. 2010.

[83] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 949–960. ACM, 2011.

[84] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In *RANDOM*, pages 159–170, 1998.

[85] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

[86] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 381–390, 2001.

[87] J. Schönheim. On coverings. *Pacific Journal of Mathematics*, 14:1405–1411, 1964.

[88] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. Data curation at scale: The data tamer system. In *Proc. 6th Biennial Conf. on Innovative Data Systems Research*, 2013.

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

[90] Jeffrey D. Ullman. Designing good mapreduce algorithms. *XRDS*, 19(1):30–34, September 2012.

[91] Norases Vesdapunt, Kedar Bellare, and Nilesh Dalvi. Crowdsourcing algorithms for entity resolution. *Proc. VLDB Endowment*, 7(12), 2014.

[92] Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J. Miller. Continuous data cleaning. In *Proc. 30th Int. Conf. on Data Engineering*, pages 244–255, 2014.

[93] Jiannan Wang, Tim Kraska, Michael J Franklin, and Jianhua Feng. Crowder: Crowdsourcing entity resolution. *Proc. VLDB Endowment*, 5(11):1483–1494, 2012.

[94] Jiannan Wang, Guoliang Li, Tim Kraska, Michael J Franklin, and Jianhua Feng. Leveraging transitive relations for crowdsourced joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 229–240, 2013.

[95] Jiannan Wang and Nan Tang. Towards dependable data repairing with fixing rules. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 457–468. ACM, 2014.

[96] Eugene Wu and Samuel Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endowment*, 6(8):553–564, 2013.

[97] Eugene Wu, Samuel Madden, and Michael Stonebraker. A demonstration of dbwipes: clean as you query. *Proc. VLDB Endowment*, 5(12):1894–1897, 2012.

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

[99] Mohamed Yakout, Ahmed K Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F Ilyas. Guided data repair. *Proc. VLDB Endowment*, 4(5):279–289, 2011.

[100] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *HotCloud*, volume 10, page 10, 2010.