Modeling and Querying Uncertainty in Data Cleaning

by

George Beskales

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, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Data quality problems such as duplicate records, missing values, and violation of integrity constrains frequently appear in real world applications. Such problems cost enterprises billions of dollars annually, and might have unpredictable consequences in mission-critical tasks. The process of data cleaning refers to detecting and correcting errors in data in order to improve the data quality. Numerous efforts have been taken towards improving the effectiveness and the efficiency of the data cleaning.

A major challenge in the data cleaning process is the inherent uncertainty about the cleaning decisions that should be taken by the cleaning algorithms (e.g., deciding whether two records are duplicates or not). Existing data cleaning systems deal with the uncertainty in data cleaning decisions by selecting one alternative, based on some heuristics, while discarding (i.e., destroying) all other alternatives, which results in a false sense of certainty. Furthermore, because of the complex dependencies among cleaning decisions, it is difficult to reverse the process of destroying some alternatives (e.g., when new external information becomes available). In most cases, restarting the data cleaning from scratch is inevitable whenever we need to incorporate new evidence.

To address the uncertainty in the data cleaning process, we propose a new approach, called probabilistic data cleaning, that views data cleaning as a random process whose possible outcomes are possible clean instances (i.e., repairs). Our approach generates multiple possible clean instances to avoid the destructive aspect of current cleaning systems. In this dissertation, we apply this approach in the context of two prominent data cleaning problems: duplicate elimination, and repairing violations of functional dependencies (FDs).

First, we propose a probabilistic cleaning approach for the problem of duplicate elimination. We define a space of possible repairs that can be efficiently generated. To achieve this goal, we concentrate on a family of duplicate detection approaches that are based on parameterized hierarchical clustering algorithms. We propose a novel probabilistic data model that compactly encodes the defined space of possible repairs. We show how to efficiently answer relational queries using the set of possible repairs. We also define new types of queries that reason about the uncertainty in the duplicate elimination process.

Second, in the context of repairing violations of FDs, we propose a novel data cleaning approach that allows sampling from a space of possible repairs. Initially, we contrast the existing definitions of possible repairs, and we propose a new definition of possible repairs that can be sampled efficiently. We present an algorithm that randomly samples from this space, and we present multiple optimizations to improve the performance of the sampling algorithm.

Third, we show how to apply our probabilistic data cleaning approach in scenarios where both data and FDs are unclean (e.g., due to data evolution or inaccurate understanding of the data semantics). We propose a framework that simultaneously modifies the data and the FDs while satisfying multiple objectives, such as consistency of the resulting data with respect to the resulting FDs, (approximate) minimality of changes of data and FDs, and leveraging the trade-off between trusting the data and trusting the FDs. In presence of uncertainty in the relative trust in data versus FDs, we show how to extend our cleaning algorithm to efficiently generate multiple possible repairs, each of which corresponds to a different level of relative trust.

Acknowledgements

First, I would like to thank my advisor, Prof. Ihab Ilyas, for his continues encouragement and guidance throughout my PhD program. Prof. Ilyas has taught me how to be a thorough researcher, paying attention to the smallest details, and be self-motivated.

I am greatly thankful to Prof. Lukasz Golab for providing me with help, enthusiasm and support during the past three years.

I am deeply grateful to my thesis committee members, Prof. Shai Ben-David, Prof. Renée J. Miller, Prof. David Toman, and Prof. Olga Vechtomova. Such outstanding group of researchers has significantly helped shaping my research through their direct guidance and through their own research. I would like to thank my thesis committee members for their invaluable feedback and suggestions that definitely made this dissertation better.

This dissertation would not have been possible without the support of my wife, Alice, who has been there for me, supporting and motivating me all the time. I would like to thank my parents, Anwar and Narguis, for their ongoing and endless sacrifices. They have given everything without expecting anything in return.

Above all, I thank God for all His blessings and mercies. I pray to God to guide me to the right path in this life, and make me a useful member of the community.

Dedication

This is dedicated to my wife, Alice, and my parents Anwar and Narguis. Without your support and unconditional love, this dissertation would not have been possible.

Table of Contents

List of Figures

Chapter 1

Introduction

Data quality is a key requirement for effective data analysis and data processing. In many situations, the quality of business and scientific data is impaired by several sources of noise (e.g., heterogeneity in data formats, imperfection of information extractors, and imprecision of reading devices). Such noise generates many data quality problems (e.g., missing values [29, 64], violated integrity constraints [9, 19, 57], and duplicate records [33, 65]). Errors in data impact the effectiveness of many data querying and analysis tasks, and cost enterprises billions of dollars annually and might have unpredictable consequences in mission-critical tasks [32]. Databases that experience data quality problems are usually referred to as unclean/dirty databases. The process of data cleaning refers to detecting and correcting errors in data. Great efforts have been made to improve the effectiveness and the efficiency of the data cleaning.

Existing data cleaning techniques perform a number of data modifications (e.g., deleting/inserting tuples and modifying tuple attributes) in order to resolve errors found in a given database instance. Data cleaning techniques usually depend on a number of heuristics to choose between multiple plausible alternatives to clean the data. For example, most duplicate elimination systems determine whether two tuples are duplicates or not by measuring the similarity between the tuples, based on some similarity metric, and comparing the obtained similarity value to a predefined threshold. Due to the noise in real-world data and the inaccuracy of the similarity measures, such a process is merely a heuristic that

could result in false decisions. Another heuristic in the context of repairing violations of integrity constraints is striving for minimality of the number of data changes that are performed to bring the database in accordance with the integrity constraints [19, 57]. Again, such criteria do not necessarily lead to correct data cleaning decisions.

The role of the heuristics is mainly overcoming uncertainty in the data cleaning process. This uncertainty is mainly due to the presence of different possible ways to clean the data. Using some heuristics to pick only one alternative effectively *destroys* the other plausible ways to clean the data, which results in a false sense of certainty about the generated data repairs. For example, reporting two moderately similar tuples as duplicates ignores the possibility that the two tuples are not duplicates. Also, when repairing an integrity constraint violation, imposing a minimality constraint over the number of data changes effectively discards all other alternatives that suggest repairing the violation in a non-minimal way. The cost of completely discarding other possible ways to clean the data is twofold:

1. Using a single database repair for answering user queries results in only a subset of all possible query answers. Also, the obtained query answers are not associated with any correctness guarantee, which reduces the usability of the query answers.

2. The current cleaning approach is extremely rigid as the generated single repair is tightly coupled to narrow cleaning specifications. In general, any modifications to such specifications would require restarting the entire cleaning process from scratch.

In this dissertation, our goal is to prevent loss of potentially interesting repairs that is found in existing single-repair data cleaning systems. We extend the data cleaning process to produce multiple data instances that represent possible repairs of the input database. More specifically, we view the data cleaning process as a random process whose possible outcomes represent possible repairs of the data.

We contrast the one-shot, deterministic data cleaning approach and the probabilistic cleaning approach in Figure 1.1. Once we generate all possible repairs, we use probabilistic

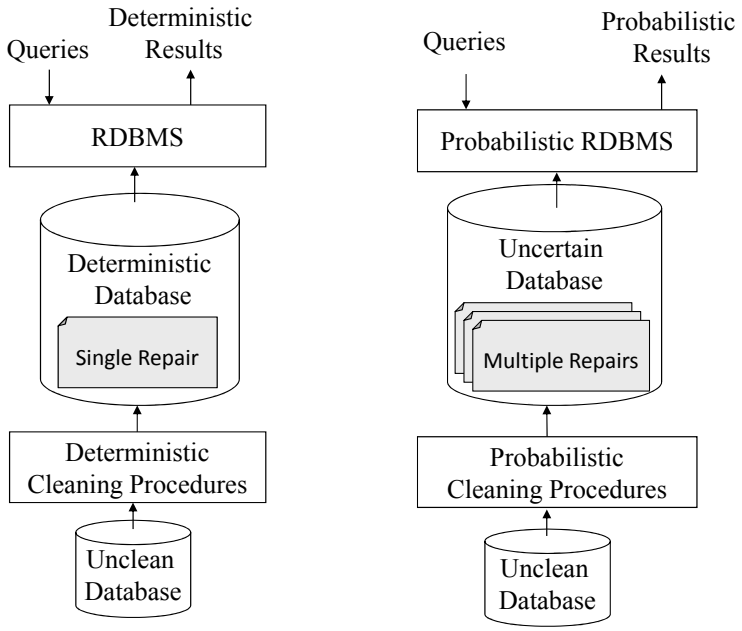

Figure 1.1: One-shot (deterministic) cleaning versus probabilistic cleaning

data management techniques (e.g., [5, 55, 73]) to allow efficient probabilistic query processing. That is, queries are processed against all possible repairs in order to obtain all possible answers based on the possible worlds sematic (refer to Section 2.3.2).

In the following, we list a number of applications that can benefit from probabilistic data cleaning.

- **Capacity planning** A typical example of capacity planning is to find the minimum and maximum possible numbers of distinct entities (e.g., clients) in an unclean database for best-case and worst-case planning. Another example is aggregation queries, where a user might require probabilistically quantified aggregate values (e.g., in the form of confidence intervals), rather than a single value. Such examples can be easily addressed with the help of probabilistic query answering against the set of all possible repairs.

- **Handling multiple cleaning requirements.** In some scenarios, multiple users of a database might have different requirements regarding the data cleaning process. For example, one user might prefer a conservative deduplication strategy (i.e., only merge tuples that are highly similar), while another user prefers a more aggressive deduplication strategy (i.e., merge tuples that have moderate similarity). Because the probabilistic data cleaning generates all possible repairs, it is possible to efficiently extract the repair(s) satisfying a given requirement without invoking the cleaning process from scratch.

- **Interactive data cleaning.** Another possible application is interactive data cleaning, where a human guides the data cleaning system by choosing from multiple alternatives to clean the data. Our data cleaning approach can be used to materialize (parts of) the possible clean instances, and let the user decide which one is the most accurate.

In the following, we summarize the main challenges that arise during implementing our probabilistic data cleaning approach.

- **Identifying the sources of uncertainty.** Each error type requires a specific cleaning procedure that depends on various cleaning decisions (e.g., deciding whether two tuples are duplicate or not, and deciding which attribute should be modified to fix a violation of an integrity constraint). Identifying the cleaning decisions in a given cleaning process and their possible outcomes is necessary for modeling the generative process that enumerates all possible clean instances.

- **Identifying a set of possible clean instances.** The number of all possible clean instances is extremely large for many data quality problems. To provide a practical approach, it is necessary to restrict the space of possible instances to a reasonable subset by pruning off instances that are unlikely to be correct repairs of data.

- **Efficiently generating possible clean instances.** We have to ensure that the computational overhead due to generating multiple clean instances is reasonable, compared to the existing techniques that generate a single clean instance.

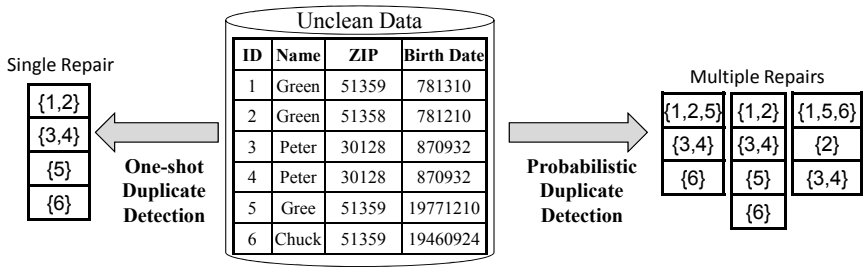

Figure 1.2: One-shot duplicate elimination versus probabilistic duplicate elimination

- **Compactly storing the clean instances.** To reduce the space required for storing the generated clean instances, we identify and remove redundancy in the clean instances. Because the generated instances represent possible repairs of the same data instance, there is usually a significant amount of redundancy in the instances, which reduces the storage requirements.

In this dissertation, we provide three case studies to show how to apply our cleaning approach. Two prominent data quality problems that are frequently found in practice are duplicate records, and violations of functional dependencies (FDs). In Section 1.1, we show how to apply our approach to the problem of duplicate elimination. In Section 1.2, we discuss probabilistic repairing of FD violations. In Section 1.3, we discuss the probabilistic cleaning of both data and FDs.

1.1 Duplicate Records Elimination

Duplicate records are tuples in the database that refer to the same real-world entity. The problem of having duplicate records arises in many scenarios such as data integration, Web data extraction, and manual data entry. The first task towards cleaning duplicate records is to determine groups of tuples that are duplicates. The output of this task represents a clustering (i.e., a partitioning) of the database tuples. The second task of the cleaning process is to consolidate each group of duplicate tuples into one tuple.

For example, in Figure 1.2, we show multiple possible clusterings of the input tuples (represented as sets of tuples IDs). Each cluster is eventually merged into one representative tuple. In these settings, each possible clustering of tuples represents one way to repair the data.

The number of all possible clusterings is exponential in the number of tuples. One method to reduce the number of possible clusterings is by selecting a parameterized clustering algorithm, and only generating the clusterings that are valid outcomes of the selected algorithm for some parameter value. For a certain class of clustering algorithm, namely hierarchical clustering algorithms, it is possible to efficiently obtain the set of possible clusterings by executing the clustering algorithm only once.

In order to compactly store the generated clean instances, we develop a probabilistic data model that stores distinct tuples that appear in the generated instances, and keeps the lineage information of each tuple such as the parameter values generating this tuple. Moreover, we show how to use the generated clean instances to answer user queries probabilistically, and we introduce new query types that reason about the uncertainty in the generated instances. We show how to implement probabilistic query processing using a relational DBMS by rewriting user queries to take into consideration the existence of multiple possible instances.

1.2 Repairing Functional Dependency Violations

Functional dependencies (FDs) represent a prominent class of integrity constraints that are used for capturing data semantics. An FD $X \rightarrow Y$, where X and Y are sets of attributes in a given relation, indicates that any two tuples that have equal values for attributes in X must have equal values for attributes in Y. An example FD in Figure 1.3 is `ZIP`→`State, City`, which indicates that any tuple with the same ZIP code must have the same city and the same state.

Violations of FDs indicate deviations from the data semantics and should be rectified through a data cleaning algorithm. In practice, FDs tend to be violated after integrating heterogeneous data or when extracting data from the Web. Even in a traditional DBMS,

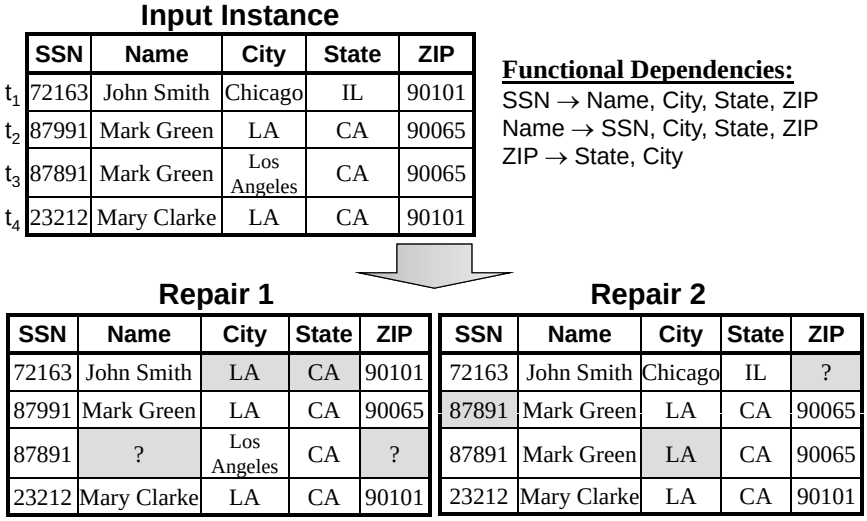

Figure 1.3: An example of an unclean database and possible repairs

unknown FDs may be hidden (i.e., not explicitly captured in schema), or the database administrator may choose not to enforce some FDs for various reasons. For example, Figure 1.3 shows a database instance and a set of FDs, some of which are violated (e.g., tuples t_2 and t_3 violate `ZIP`→`City`, tuples t_2 and t_3 violate `Name`→ `SSN,City`, and tuples t_1 and t_4 violate `ZIP` → `State,City`).

There is often a very large number of ways to modify a table so that it satisfies all the required FDs. One way is to delete a number of offending tuples such that the remaining tuples satisfy all FDs [23, 24]. For example, we can repair the relation instance in Figure 1.3 by deleting t_1 and t_3. However, deleting an entire tuple may result in loss of clean information if only a subset of its attribute is incorrect. Alternatively, we can modify selected attribute values [19, 57]. For example, Figure 1.3 shows two possible repairs obtained by modifying some attributes values (shown as shaded cells). Question marks indicate that a tuple attribute (i.e., a cell) can be modified to one of several values in order to satisfy the FDs. For example, attribute `ZIP` of tuple t_1 in Repair 2 can be changed to any ZIP code as long as it is not equal to 90065 or 90101.

Independently of how we choose to repair violations, two cleaning approaches have appeared in previous work. One is to produce a single repair with (approximately) minimum

number of deletions or attribute modifications (e.g., [19, 57]). For instance, we might prefer `Repair 2` in Figure 1.3 because it makes fewer modifications. Another approach, namely consistent query answering, computes answers to selected classes of queries that are valid in every possible "reasonable" repair [9, 23, 24, 42, 79, 80]. In Figure 1.3, a consistent answer of the query that selects all tuples with ZIP code 90101, with respect to the two illustrated repairs, is $\{t_4\}$.

Although consistent query answering acknowledges the existence of multiple possible clean instances, we still consider it as destructive data cleaning. The reason is that consistent query answering discards alternatives that are not completely certain. That is, a tuple that only appears in a subset of all possible repairs cannot appear in any query results; query answers are only derived from tuples that appear in every possible repair (for some definition of possible repairs). Also, in consistent query answering, possible repairs are not materialized. Instead, consistent answers are directly derived from the input, unclean database (e.g., through query rewriting). Such approach is not suitable for several applications such as interactive data cleaning, where a user might need to explore concrete data repairs.

The space of possible FD repairs is very large. It follows that generate all repairs is not feasible. Instead, we aim at finding a meaningful subset of repairs that can generated in an efficient way. In this work, we introduce a space of repairs that change *minimal sets* of tuple attributes (cells). That is, for each repair in the space that changes a set of cells $\mathcal{C}$, there does not exist any other repair that changes a strict subset of $\mathcal{C}$.

In order to explore the space of possible repairs, we develop a randomized cleaning algorithm that generates a sample of clean instances from the described space. Once a sample of database instances is generated, uncertain database management systems such as Monte Carlo Database System (MCDB) [55] can be used for storing the sampled clean instances, and for efficiently answering user queries against the clean instances.

1.3 Repairing Unclean Data and Unclean FDs

Most of the existing work on FD-driven data cleaning assumes that the given FDs are completely correct and modifies the data, in a minimal and non-redundant way, to be consistent with the FDs [15, 19, 27, 57]. However, there are many cases in which the FDs themselves are inaccurate. One cause, for example, is domain evolution, which occurs when the semantics of the data change over time [78]. Examples of domain evolution can be found in the context of database integration and data federation, where the semantics of multiple databases might clash and need readjustment in order to hold globally. Similar problems may arise when organizations merge or split departments, eventually leading to changes in the data semantics. Another notable example is schema evolution of Web data such as Wikipedia (more than 240 schema versions in the last 6 years) [28].

In the following, we show an example of unclean data and unclean (i.e., inaccurate) FD.

Example 1. *Figure 1.4 depicts a relation that holds employee information within an institute. Data are collected over time from various sources (e.g., Payroll records, HR) and thus might contain inconsistencies due to duplicate records. Suppose that we initially assert the FD* `Surname, GivenName` $\rightarrow$ `Income`*. That is, whenever two tuples agree on attributes* `Surname` *and* `GivenName`*, they must agree on* `Income`*. This FD may hold for Western names, in which surname and given name uniquely identify a person in most cases, but not for Chinese names (e.g., tuples* t_6 *and* t_9 *probably refer to different persons).*

The instance in Figure 1.4 violates the given FD due to errors in data and due to using an imprecise FD. One way to clean the data is to first change the FD to `Surname, GivenName, BirthDate` $\rightarrow$ `Income` and then modify attribute `Income` of t_5 (or t_3) to be equal to that of t_3 (respectively, t_5).

We propose a framework that simultaneously repairs the provided FDs, and obtains clean instances of the database. More specifically, a repair represents a pair of a data instance and a set of FDs that are satisfied by the data instance. Clearly, the space of possible repairs in this context is much larger than the space of possible repairs of FD violations when the FDs are fixed. In order to restrict the possible repairs to a reasonable

	GivenName	Surname	BirthDate	Gender	Phone	Income
t_1	Jack	White	5 Jan 1980	Male	923-234-4532	60k
t_2	Sam	McCarthy	19 Jul 1945	Male	989-321-4232	92k
t_3	Danielle	Blake	9 Dec 1970	Female	817-213-1211	120k
t_4	Matthew	Webb	23 Aug 1985	Male	246-481-0992	87k
t_5	Danielle	Blake	9 Dec 1970	Female	817-988-9211	100k
t_6	Hong	Li	27 Oct 1972	Female	591-977-1244	90k
t_7	Jian	Zhang	14 Apr 1990	Male	912-143-4981	55k
t_8	Ning	Wu	3 Nov 1982	Male	313-134-9241	90k
t_9	Hong	Li	8 Mar 1979	Female	498-214-5822	84k
t_{10}	Ning	Wu	8 Nov 1982	Male	323-456-3452	95k

Figure 1.4: An example of an unclean database of persons

subset, we leverage the concepts of minimality of changes and the relative trust between data and FDs, which are described as follows.

- **Minimality of Changes.** The amount of changes made to the data and the FDs in order to obtain a repair should be minimum, based on some metrics that quantify the changes in data and FDs.

- **Relative Trust in Data versus FDs.** We incorporate prior knowledge about which of data and FDs is cleaner in order to concentrate on repairs that are biased towards modifying either the data or the FDs while obtaining a repair. In Example 1, a repair that trusts the FD more than data could change attribute `Income` of tuples t_5, t_6 and t_{10} to be equal to the income of t_3, t_9 and t_8, respectively, while keeping the FD unchanged. A repair that focuses on repairing the FD, while trusting the data, might only change the FD to `Surname, GivenName, Birthdate, Phone` $\rightarrow$ `Income`. We quantify the relative trust in Data versus FDs by imposing a constraint on the amount of allowed data changes. Allowing a relatively small number of data changes reflects higher trust in data, and vice versa.

In order to specify the relative trust in data versus FDs, it is necessary to estimate the amount of errors in data (i.e., the maximum number of allowed changes). Unfortunately,

the amount of errors in data is precisely known in practice. However, it might be easier to specify a range of possible values for the amount of data errors. Our approach enables efficiently generating all pairs of data and FDs repairs that correspond to the given possible values of the relative trust.

1.4 Contributions and Dissertation Outline

In this dissertation, we provide a probabilistic data cleaning approach that leverages uncertainty in the data cleaning process. The outcome of our cleaning approach is a set of possible repairs. We mainly study how to efficiently generate a set of possible repairs from a space of reasonable repairs. We apply the concept of probabilistic data cleaning in the context of two prominent data quality problems: duplicate elimination and violations of functional dependencies. In the following, we provide more details about our contributions.

- **Probabilistic duplicate detection.** We introduce a probabilistic data model for representing the possible repairs generated by any fixed parameterized clustering algorithm. We show how to modify hierarchical clustering algorithms to efficiently generate the possible repairs. We describe how to evaluate relational queries under our model and we propose new query types that reason about uncertainty in the cleaning process. Finally, we show how to integrate our approach into an RDBMS to allow storage of possible repairs and to perform probabilistic query processing efficiently.

- **Sampling repairs of FD violations.** We introduce a novel notion of possible repairs that relaxes the minimality constraint on the number of data changes. We give an algorithm for generating a random sample of repairs from the introduced space. We show how to improve the efficiency of the sampling algorithm by partitioning the input instance into blocks that can be repaired independently. Finally, we describe how to extend our algorithm to prevent certain parts of the data from being changed during the data cleaning (e.g., when completely trusting parts of the database).

- **Probabilistic cleaning of data and FDs.** We introduce cardinality-based metrics to measure changes of FDs. We define minimality criteria based on dominance with respect to changes in data and FDs. We present a cleaning approach that satisfies the objectives described in Section 1.3, given a single value for the relative trust. We show how to extend our algorithm to allow a range of values.

The remainder of the dissertation is organized as follows. In Chapter 2, we list the notations used throughout the dissertation, and we give an overview of the related work. In Chapter 3, we present our approach of probabilistic duplicate elimination. In Chapter 4, we describe our approach to sampling repairs of FD violations. In Chapter 5, we show how to apply our uncertain cleaning approach to simultaneously repair data and FDs when both are unclean. Finally, in Chapter 6, we conclude the dissertation with final remarks and directions for future work.

Chapter 2

Background and Related Work

In this chapter, we provide the necessary preliminaries, and we overview the related work. In Section 2.1, we present definitions and notations that are used throughout the dissertation. In Section 2.2, we discuss data quality problems, we overview previous data cleaning techniques, and we highlight the underlying uncertainty in repairing each type of data errors. In Section 2.3, we give an overview of probabilistic query answering, and the prominent work in this area.

2.1 Preliminaries

A database schema is a set of k relations $R_1, \ldots, R_k$. The schema of a relation R consists of a number of attributes, denoted $(A_1, \ldots, A_{|R|})$, where $|R|$ denotes the number of attributes in relation R. We denote by $Dom(A)$ the domain of an attribute A. A database instance is a set of relation instances $I_1, \ldots, I_k$ of the database relations $R_1, \ldots, R_k$. An instance I of a relation R consisting of attributes $(A_1, \ldots, A_{|R|})$ is a set of tuples, each of which belongs to the domain $Dom(A_1) \times \cdots \times Dom(A_{|R|})$.

We refer to an attribute $A \in R$ of a tuple $t \in I$ as a *cell*, denoted $t[A]$. We denote by $I(t[A])$ the value of a cell $t[A]$ in a relation instance I. When it is clear from the context

which instance we refer to, we omit the mention of instance I and we refer to the value of $t[A]$ in I as simply $t[A]$.

An integrity constraint (IC) is a condition that is specified on a database schema, and restricts the data that can be stored in an instance of the database [68]. We denote by Σ a set of integrity constraints (ICs) that are defined on the database schema. We denote by $|\Sigma|$ the number of ICs in Σ. We say that instance I satisfies Σ, written $I \models \Sigma$, iff the tuples in I do not violate any IC in Σ.

For example, one class of integrity constraints includes functional dependencies (FDs), which are defined as follows. For two attribute sets $X, Y \subseteq R$, a functional dependency $X \rightarrow Y$ holds on an instance I, denoted $I \models X \rightarrow Y$, iff for every two tuples t_1, t_2 in I, $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$.

2.2 Data Cleaning

Quality of data has a significant impact on usability of data and the credibility of the information derived from the data. There are several causes of data quality degradation such as integration of heterogenous data, noisy sensors, and human errors. Forms of data quality problems include missing attribute values [29, 64], violations of integrity constraints [9, 19, 57], existence of duplicate records [33, 65], heterogenous data formats, and syntactic errors in attribute values [64]. The main goal of a data cleaning process is to remove data errors and thus improve the quality of data. In the following, we discuss two prominent data quality problems: duplicate records and violations of integrity constraints.

2.2.1 Duplicate Records

Duplicate records are database tuples that refer to the same real world entity. Duplicate tuples do not have to be identical. However, duplicate tuples are expected to exhibit a high degree of similarity. The process of duplicate elimination (also known as deduplication, record linkage, entity resolution, and object identification) is defined as identifying groups of duplicate tuples, and consolidating the tuples in each group into one tuple.

Several duplicate elimination systems (e.g., [43, 59]) split the process of deduplication into three main tasks:

1. **Tuple matching.** to obtain pairs of tuples that are similar.
2. **Clustering tuples.** to partition the tuples into disjoint clusters, each of which represents a real-world entity.
3. **Coalescing members of each cluster.** to consolidate each cluster of tuples into one representative tuple.

In the following, we describe each task in more details. Matching tuples refer to obtaining pairs of tuples that have high similarity, based on some similarity metric. The output of the tuple matching step can be represented by a weighted undirected graph whose set of nodes represents the set of tuples, and each edge connects two similar tuples. The weight of each edge reflects the degree of similarity between the edge nodes. Different similarity measures have been employed in deduplication systems such as Euclidean distance, edit distance, cosine similarity, Jaccard distance and Q-grams (refer to [33] for a comprehensive survey of similarity metrics). In [17], a new similarity metric has been proposed to take into account the relationships between entities in different database relations. For example, to compute the similarity between two authors, it is beneficial to compute the set of common co-authors. If the number of common co-authors is significant, and the authors' names are similar, most likely the two authors are duplicates.

It is expected that most of the tuple pairs are not similar. Several optimizations have been proposed to leverage such a fact in order to avoid computing the similarity between all pairs of tuples. For example, one optimization method introduced in [63] is to divide data tuples into overlapping *canopies*, where each canopy includes all tuples that have non-zero chance of being duplicates. Canopies are constructed by clustering tuples based on a relatively simple similarity metrics that is cheap to compute. Other optimization techniques were experimentally evaluated in [11].

The second task in the duplicate elimination process is clustering tuples such that each cluster of tuples refers to a single real-world entity. Performing the clustering task is necessary due to the possibility of having conflicts in tuple similarities. That is, the obtained

tuple matchings might not represent a transitive relation (e.g., tuple t_1 is deemed similar to t_2, based on some similarity metric, and tuple t_2 is similar to t_3, while t_1 is not similar to t_3). In general, clustering algorithms aim at producing a clustering that maximizes the similarity between all pairs of tuples belonging to the same cluster (i.e., intra-cluster similarity), and minimizing the similarity between all pairs of tuples belonging to different clusters (i.e., inter-cluster similarity). Current duplicate elimination systems exploit classical clustering algorithms such as greedy agglomerative clustering, star-clustering, cut clustering, Markov clustering and correlation clustering (refer to [50] for a comprehensive comparison between clustering algorithms in the context of duplicate record elimination).

Some specialized clustering algorithms have been designed for the problem of duplicate elimination (e.g., [20]). The algorithm proposed in [20] is based on two clustering criteria: (1) compactness of duplicates (i.e., duplicate tuples should be the k-nearest neighbors of each other), and (2) sparseness of the duplicates' neighborhood (i.e., the space surrounding the duplicate tuples should be relatively empty). The algorithm performs the clustering through a series of SQL queries, which capitalize on the data management capabilities provided by RDBMSs to speed up the clustering.

In [7], a clustering algorithm is proposed to allow specifying a set of hard constraints (e.g., tuples t_1 and t_2 must be clustered together, while t_3 and t_4 must be in different clusters), in addition to soft constraints each of which is associated with a cost. Hard and soft constraints are described using a Datalog-style language, which is a subclass of first-order logic. The authors proposed a randomized approximate algorithm, where the approximation factor is three (in expectation). The core of the algorithm is based on iterative hardening of soft edges (i.e., converting soft edges to hard edges), and using the transitive closure property to deduce other hard edges.

The third step in the process of duplicate elimination is to merge each cluster of tuples into one tuple. Merging is usually done by applying domain-specific rules. For example, in a database of publication records, longer author names are usually used in the representative tuples [43]. Other examples are using the attribute value that appears in the majority of duplicates, and averaging numerical attributes across duplicates.

Examples of Duplicate Elimination Systems

Several end-to-end duplicate elimination systems have been proposed to address different goals. For example, AJAX [43] is an extensible framework that separates the logical and physical aspects of the data cleaning process. The logical view specifies the design of the data cleaning workflow, which consists of multiple cleaning operators, while the physical view specifies the algorithms that implement the cleaning operators. Logical operations include the typical matching, clustering and merging tasks, in addition to other data preparing tasks expressed as SQL queries. Once the logical workflow is specified by the user, the system selects the best algorithm to implement each logical operator in order to reduce the overall cost.

Another open-source deduplication system is Febrl (Freely Extensible Biomedical Record Linkage) [26]. The system mainly focuses on deduplication of electronic medical records. However, Febrl can be used in other domains as well, once the user specifies the details of the deduplication job (e.g., which similarity metrics to use). The system is divided into multiple stages, including standardization of names and addresses, partitioning the data into disjoint blocks, and computing the pairwise similarities between tuples in each block. Unfortunately, Febrl does not cluster the tuples, or merge the duplicate tuples.

Other deduplication systems include Potter's Wheel [69], which is an interactive data cleaning system that integrates data transformation and error detection using spreadsheet-like interface, and IntelliClean [59], which is a rule-based duplicate elimination system.

Uncertainty in Duplicate Elimination

Each task of the deduplication process involves a degree of uncertainty. For example, in the tuple matching task, there exists a large number of similarity metrics (e.g., edit distance, Q-gram distance, Jaro distance [33]) that measure the similarity of two values that belong to the same domain. Choosing the most accurate metric for a given domain is not straightforward. Also, it is difficult to interpret the values resulting from different similarity metrics. This is particularly true for similarity metrics that return a non-normalized (i.e., absolute) values such as the edit distance metric, which computes the number of character

insertions, deletions and updates that makes two strings identical.

The process of aggregating attribute similarities to obtain similarity between pairs of tuples is not straightforward either. Several supervised and unsupervised machine learning classifiers have been used in this process such as Naïve Bayes and Support Vector Machine (refer to [33] for a survey of such methods). To overcome the uncertainty in the matching process, several proposals define an *uncertainty region*, in which the matching process cannot make a clear distinction between duplicates and non-duplicate tuples decision [33, 38, 43, 75]. Tuple pairs that belong to the uncertainty region are usually forwarded to a human in form of exceptions.

A related work that provides a probabilistic record-linkage approach has been discussed in [76]. The goal is to integrate two lists of items that possibly contain duplicate references to the same real world entities. The authors presented an XML-based uncertainty model to capture multiple possibilities concerning the output list. However, the proposed algorithm cannot be used for deduplicating a single list that contains several duplicate references, a situation that is frequently seen in practice.

The second task, which clusters tuples into groups of duplicates, has several sources of uncertainty. First, choosing the right clustering algorithm is not straightforward. Hassanzadeh et al. have provided a comprehensive comparison of different clustering algorithm [50]. However, deciding which algorithm should be used heavily depends on many factors such as the performance requirements, quality requirements, and most importantly, the characteristics of the data (e.g., the distribution of duplicates, and the amount of duplicates in data). Another common challenge in the clustering task is identifying the optimal settings of the algorithm parameters that obtain the highest accuracy. For example, greedy agglomerative clustering algorithms use a threshold on the pairwise similarity of clusters to determine when to stop the clustering process [17]. In this case, the user is forced to pick a single parameter value.

The third step of the deduplication process, which is merging clusters of tuples, involves a number of uncertain decisions, such as choosing the right criteria to resolve conflicts in tuple attributes. In [6], Andritsos et al. have proposed an approach to address uncertainty in the merging phase. In this approach, tuples are assumed to be already clustered into

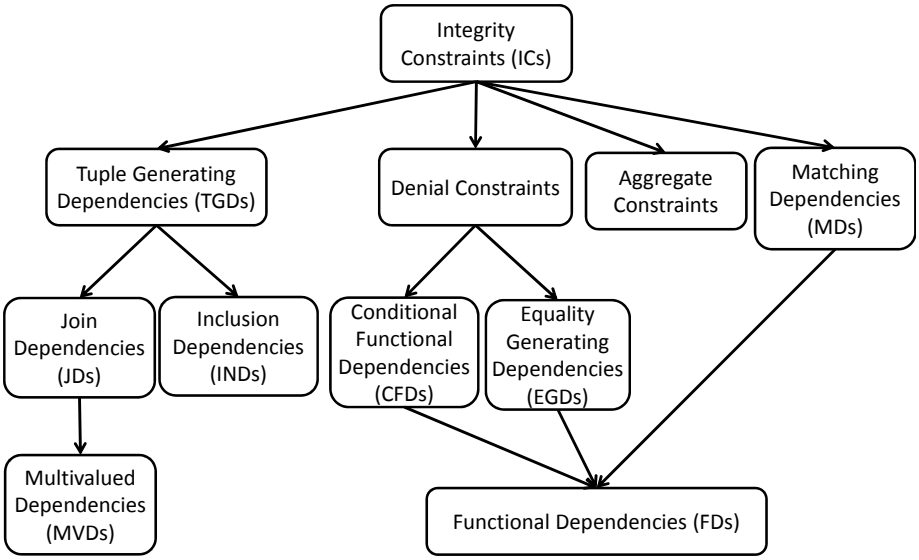

Figure 2.1: Taxonomy of Integrity Constraints

disjoint groups of duplicate records. The representative tuple associated with each cluster is assumed to be one of the cluster members. Furthermore, each tuple within a cluster is associated with the probability of being the representative tuple of the cluster. The authors provided a method to rewrite user queries in order to return all possible answers, along with their probabilities.

2.2.2 Violation of Integrity Constraints

Integrity constraints (ICs) are widely used for representing the semantics of data. Violations of ICs usually indicate drifting from the correct data semantics, which suggest that data is unclean. In this context, the process of data cleaning refers to modifying the data to bring it into accordance with the ICs. Prominent examples of ICs include key constraints, functional dependencies (FDs), inclusion dependencies (INDs), and multivalued dependencies (MVDs) [3, 35]. Such constraints belong to more expressive, larger classes of constraints such as tuple-generating-dependencies (TGDs) [67], and denial constraints [45]. Recently, several extensions to functional dependencies have been proposed such as

conditional functional dependencies (CFDs) [18], and matching dependencies [36]. Other higher-order constraints that include aggregate functions are referred to as aggregate constraints [40, 41]. Figure 2.1 provides taxonomy of several integrity constraints that are frequently used in the data cleaning literature. Each arrow in Figure 2.1 indicates that the target class of constraints is a specialization (i.e., a subclass) of the source class of constraints. For example, multivalued dependencies are a subclass of join dependencies. In the following, we give more details about each class of constraints.

A tuple-generating-dependency (TGDs) indicates that if some tuples in the database satisfy certain equalities, then some other tuples (possibly with some unknown attributes) must also exist in the database instance [67]. Formally, a TGD is defined as follows.

$$\forall \overline{x}\big(\varphi(\overline{x}) \rightarrow \exists \overline{y}\ \psi(\overline{x}, \overline{y})\big)$$

where $\overline{x}$ and $\overline{y}$ are vectors of variables, and φ and ψ are conjunctions of relations in database. A TGD expresses the condition that, if certain tuples satisfy φ, then certain other tuples must be present as well such that ψ is also satisfied.

For example, consider a database consisting of three relations: `Mother(MID,CID)`, which states that `MID` is the mother of `CID`, `Father(FID,CID)`, which states that `FID` is the father of `CID`, and `Sibling(PID1,PID2)` indicating that `PID1` is a sibling of `PID2`. An example TGD is as follows.

$$\forall x, y\big(\texttt{Sibling}(x, y) \rightarrow \exists z(\texttt{Mother}(z, x) \wedge \texttt{Mother}(z, y))\big)$$

A subclass of TGDs in which the formula ψ does not contain any existentially quantified variables is called *full TGDs*. That is, a full TGD is a formula of the form $\forall \overline{x}\big(\varphi(\overline{x}) \rightarrow \psi(\overline{x})\big)$. One example of a full TGD is as follows.

$$\forall x, y, z\big(\texttt{Mother}(x, y) \wedge \texttt{Mother}(x, z) \rightarrow \texttt{Sibling}(y, z)\big)$$

Inclusion dependencies (INDs) is a subclass of TGDs with φ and ψ consisting of a single relation each. Thus, an inclusion dependency has the form $\forall \overline{x}\big(R_1(\overline{x}) \rightarrow \exists \overline{y} R_2(\overline{x}, \overline{y})\big)$.

Another form of an inclusion dependency is $R_1[X] \subseteq R_2[Y]$, where X is a subset of attributes in R_1 and Y is a subset of attributes in R_2. $R_1[X] \subseteq R_2[Y]$ indicates that values of attributes X in an instance of relation R_1 must be a subset of values of attributes Y in an instance of relation R_2. Clearly, the numbers of attributes in X and Y are equal. One example of inclusion dependencies is as follows.

$$\forall x, y\big(\texttt{Sibling}(x, y) \rightarrow \exists z\ \texttt{Mother}(z, x)\big)$$

which can be also be specified in the form of `Sibling(PID1)` $\subseteq$ `Mother(CID)`.

Another subclass of TGDs is join dependencies. Given an instance I of a relation R, we say that I satisfy the join dependency $\bowtie (X_1, \ldots, X_n)$, where $X_i \subseteq R$ for $i \in \{1, \ldots, n\}$, iff $I = \bowtie_{i=1}^{n} \Pi_{X_i}(I)$ (i.e., the instance I is equal to the result of joining the projections of I on X_i for $i \in \{1, \ldots, n\}$). A subclass of join dependencies is the multivalued dependencies (MVDs), which are join dependencies with exactly two sets X_1 and X_2. An MVD, written $X \twoheadrightarrow Y$, that is defined over a relation R represents the following condition: for each pair t_1, t_2 of tuples of an instance I of R such that $t_1[X] = t_2[X]$, there is a tuple t in I where $t[X] = t_1[X] = t_2[X]$, $t[Y] = t_1[Y]$ and $t[R \setminus XY] = t_2[R \setminus XY]$. The MVD $X \twoheadrightarrow Y$ is thus equivalent to the join dependency $\bowtie (XY, R \setminus Y)$.

A large class of integrity constraints is denial constraints [45] (Figure 2.1). Denial constraints indicates that a set of tuples that satisfies certain conditions cannot exist in the database. Formally, a denial constraint is defined as follows.

$$\forall \overline{x}\big(\varphi(\overline{x}) \rightarrow \psi(\overline{x})\big)$$

where φ is a conjunction of relations, and ψ is a Boolean expression consisting of comparison atoms (e.g., $x_i = x_j$, $x_i \neq x_j$, $x_i < x_j$, and $x_i \leq x_j$). An example denial constraint is as follows.

$$\forall x, y\big(\texttt{Mother}(x, y) \rightarrow x \neq y\big)$$

A subclass of denial constraints is equality-generating-dependencies (EGDs), where ψ is restricted to equality comparison. An example of EGDs is as follows.

$$\forall x, y, z\big(\texttt{Mother}(x, y) \wedge \texttt{Mother}(z, y) \rightarrow x = z\big)$$

A prominent subclass of EGDs is functional dependencies (FDs). An FD, written $X \rightarrow Y$, that is defined over a relation R indicates that $t_i[X] = t_j[X] \rightarrow t_i[Y] = t_j[Y]$. For example, one FD is `CID` $\rightarrow$ `MID` that is defined on relation `Mother`. In fact this FD is equivalent to the EGD $\forall x, y, z\big(\texttt{Mother}(x, y) \wedge \texttt{Mother}(z, y) \rightarrow x = z\big)$.

A recent generalization of FDs, named *conditional functional dependencies* (CFDs), has been proposed in [18]. CFDs are regular FDs that are defined only on a subset of tuples that match a certain pattern. More specifically, a CFD is defined as a pair $(X \rightarrow A, t_c)$, where $X \rightarrow A$ is an FD, and t_c is a (pattern) tuple whose attributes are XA. Each attribute of t_c can be either a constant, or a wildcard `'_'`. An instance tuple t *matches* t_c on X, written $t[X] \asymp t_c[X]$, iff $\forall B \in X(t_c[B] = t[B] \vee t_c[B] = \texttt{'\_'})$. CFDs are divided into two variants: variable CFDs, where $t_c[A] = \texttt{'\_'}$, and constant CFDs, where $t_c[A]$ is a constant. A variable CFD $(X \rightarrow A, t_c)$ indicates that for any two tuples t_1, t_2, $t_1[X] = t_2[X] \asymp t_c[X] \rightarrow t_1[A] = t_2[A]$. A constant CFD $(X \rightarrow A, t_c)$ indicates that for each tuple t, $t[X] \asymp t_c[X] \rightarrow t[A] = t_c[A]$.

For example, consider a relation `Address(StreetNumber, StreetName, City, Country, PostalCode)`. A constant CFD defined over relation `Address` is (`PostalCode` $\rightarrow$ `City`, `(N2L3G1, Waterloo)`), which indicates that all tuples with `PostalCode = N2L3G1`, attribute `City` must be equal to `Waterloo`. An example of a variable CFD on relation `Address` is (`Country, PostalCode` $\rightarrow$ `StreetName, (UK, '_', '_')`), which indicates that for pairs of tuples with `Country = UK` and have equal values of `PostalCode`, attribute `StreetName` must be equal.

CFDs represent a subclass of denial constraints. For example, the CFD (`Country, PostalCode` $\rightarrow$ `StreetName, (UK, '_', '_')`) can be rewritten as follows.

$$\begin{aligned}&\forall \overline{x}, \overline{y}(\texttt{Address}(x_1, x_2, x_3, x_4, x_5) \wedge \texttt{Address}(y_1, y_2, y_3, y_4, y_5)\\ &\rightarrow x_4 \neq \texttt{UK} \ \vee \ y_4 \neq \texttt{UK} \ \vee \ x_5 \neq y_5 \ \vee \ x_2 = y_2)\end{aligned}$$

Another extension of functional dependencies has been proposed in [36], named *matching dependencies* (MDs), where the equality constraints that appear at the left-hand-side

and the right-hand-side of FDs can be replaced by similarity constraints. That is, a matching dependency is defined as a pair $(X \rightarrow Y, < ls, rs >)$, where X and Y are sets of attributes, and ls and rs are real values. A database instance satisfies $(X \rightarrow Y, < ls, rs >)$ iff for all two tuples with similarity of X above ls, similarity of Y is above rs. Similarity of attribute sets X and Y are computed based on some predefined similarity metrics. An example of an MD defined on relation `Address` is `(StreetName → City, <0.8,0.7>)`, which indicates that whenever two tuples have similar values of attribute `StreetName` (with a similarity score above 0.8), they must have similar values for attribute `City` (with a similarity score above 0.7).

Note that matching dependencies can be used for expressing duplicate detection rules (refer to Section 2.2.1). That is, a matching dependency $(X \rightarrow R, < ls, 1.0 >)$ can be used for specifying which tuples are duplicates based on pairwise similarity of attributes X. However, matching dependencies cannot be used for specifying how tuples should be clustered together in presence of conflicts in similarities (cf. Section 2.2.1).

Repairing FD Violations

We focus on repairing violations of functional dependencies. Given a database instance that violates a set of FDs, there are mainly two methods to repair the database: deleting tuples from the database [23, 24], or altering tuple attribute values [19, 57]. Note that inserting new tuples cannot resolve FD violations (or, more generally, any denial constraint violations). Altering tuples is preferred over deleting tuples as it minimizes the amount of lost information. That is, deleting tuples might result in removing attributes that are not involved in any violation.

Several approaches aim at repairing violations of FDs by changing the minimum number of tuple attributes (cells) [19, 57]. This problem is proved to be NP-hard, and several heuristics have been proposed to obtain a repair efficiently. For example, in [19], an iterative algorithm is proposed to fix violations of functional dependencies (FDs) and inclusion dependencies (INDs). The proposed algorithm fixes FD violations by repeatedly selecting an FD violation, and modifying the right-hand-side attribute of the violating tuples to be equal. In order to guarantee termination, the algorithm memorizes the sets of cells

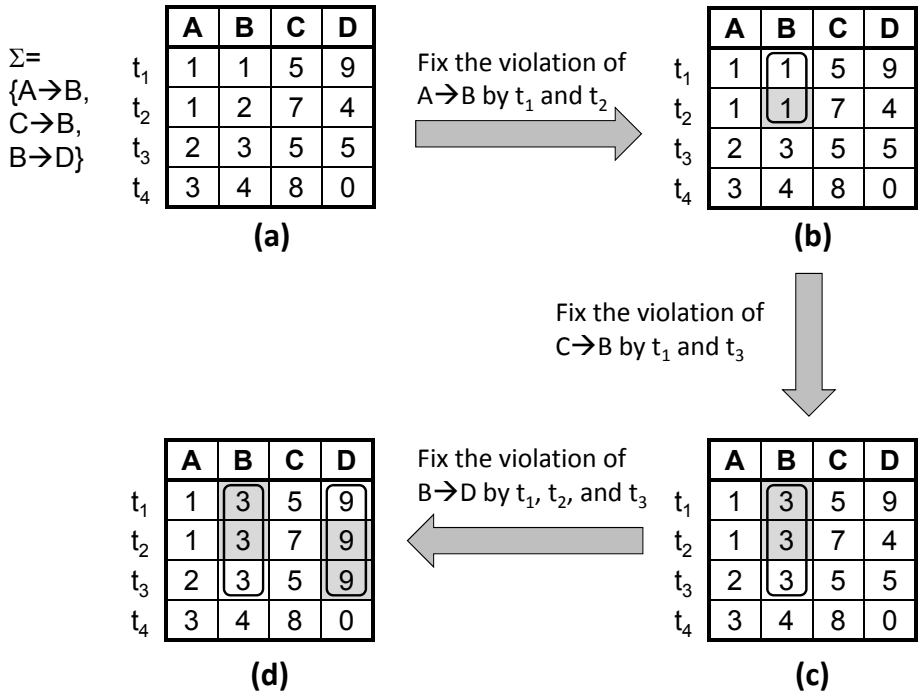

Figure 2.2: An example illustrating execution of the cleaning algorithm in [19]

that have been equated, and it ensures that their values remain equal throughout the algorithm execution. This is achieved by defining and maintaining an equivalence relation on the database cells such that cells belonging to the same equivalence class must be equal. Initially, each cell in the database is assigned to a separate equivalence class. When repairing a violation of an FD $X \rightarrow A$ involving two tuples t_1 and t_2, the algorithm merges the equivalences classes of $t_1[A]$ and $t_2[A]$, and changes the values of all cells in the resulting equivalence class to be equal.

Figure 2.2 shows an example of executing the cleaning algorithm in [19]. Cells that are changed during the algorithm execution are shaded. Equivalence classes are shown as rectangles (we omit rectangles of singleton classes). Figure 2.2(a) shows the input instance and the given FDs. Initially, the defined equivalence relation places every cell in a separate (singleton) equivalence class. The algorithm selects the violation involving FD $A \rightarrow B$ and tuples t_1 and t_2. To fix this violation, the algorithm merges the equivalence classes of the

right-hand-side cells $t_1[B]$ and $t_2[B]$, and assigns the value 1 to all cells in the resulting equivalence class. The resulting instance is shown in Figure 2.2(b). The next violation involving $C \rightarrow B$ and tuples t_1 and t_3 is resolved in Figure 2.2(c), and finally the violation of $B \rightarrow D$ by tuples t_1, t_2 and t_3 is resolved in Figure 2.2(d). Note that, after changing a cell, the list of FD violations can possibly change (i.e., new violations might appear, and existing violations might disappear). For example, the third violation involving $B \rightarrow D$ and tuples t_1, t_2, t_3 did not initially exist; it was created as a result of changing $t_1[B]$ and $t_2[B]$.

In general, the number of equivalence class merges is less than the number of tuples multiplied by the number of attributes that appear in the right-hand-sides of FDs. This guarantees that the algorithm finds a repair in polynomial time. The resulting repair, however, might contain a number of cell changes that is not the minimum across all possible repairs. This is due to using a step-wise greedy algorithm. In fact, we show in Section 4.5 that, for some data instances and FD sets, cells that are changed by the algorithm can be reverted back to their original value without causing FD violations.

The approach proposed in [27] extends the algorithm in [19] to repair violations of conditional functional dependencies (CFDs).

In [57], an algorithm was proposed to obtain a single repair with approximately the minimum number of cell changes, where the approximation factor depends only on the set of FDs and the relation schema. The algorithm is based on representing FD violations as a *hyper-graph*, which is a generalization of a graph, where an edge can connect any number of vertices. Vertices of the hyper-graph are cells of the instance, and each hyper-edge is a set of cells involved in a violation. The algorithm obtains an approximate minimum vertex cover, and modifies the cells in the vertex cover to repair all violations. A second step of the algorithm is to identify newly generated violations (due to changing cells), and performs a bounded number of cell changes to repair them.

We show an example of executing the algorithm in [57] in Figure 2.3. Hyper-edges are shown as dotted shapes in the figure. For clarity of presentation, we show two hyper-edges in Figure 2.3(a), and one hyper-edge in Figure 2.3(b). In this example, each hyper-edge consists of four cells. Assume that the obtained vertex cover consists of cells $t_2[B]$ and

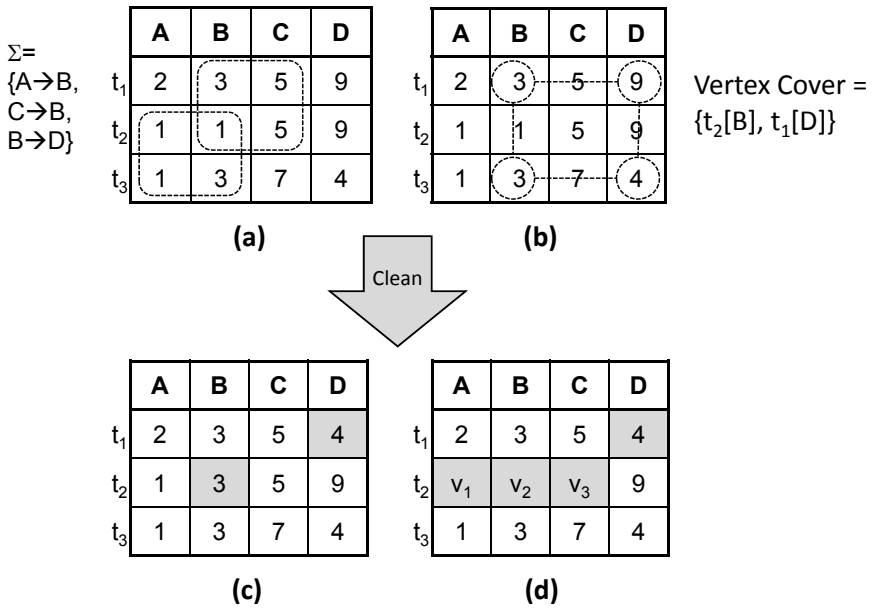

Figure 2.3: An example of execution of the algorithm in [57]. (a), (b) the initial conflict hyper-graph, (c) the instance after the first step (d) the instance after the second step

$t_1[D]$. In the first step of the algorithm, all cells in the vertex cover are changed as shown in Figure 2.3(c). Changing $t_2[B]$ to 3 creates two new violations of $B \rightarrow D$ by t_1 and t_2, and by t_2 and t_3. The algorithm fixes the newly generated violations by modifying attributes that appear in the left-hand-sides of FDs until no violation can be found. To guarantee termination of the algorithm, modified cells are assigned to variables that can only be substituted by constants that do not appear in the data instance. Furthermore, distinct variables cannot be assigned to equal constants (we provide more details in Section 4.2.1). Figure 2.3(d) shows the data instance after the second step of the algorithm. The final instance represents a number of ground instances in which v_1 cannot be equal to 1 or 2, v_2 cannot be equal to 3, and v_3 cannot be equal to 5 or 7.

Consistent Query Answering

A related line of research is consistent query answering. Given a set of possible repairs of FD violations, consistent query answering computes results of selected classes of queries that

are valid in every possible repair [9, 23, 24, 42, 79, 80]. There are two main approaches for consistent query answering. The first approach rewrites user queries based on the defined integrity constraints, and use the unmodified input instance to answer the query (e.g., [9, 42]). A second approach is to construct a condensed representation of all repairs that allows obtaining consistent answers [79, 80]. A restricted class of queries can be answered efficiently while harder classes are answered using approximate methods (e.g., [60]). Unfortunately, consistent query answering adopts an aggressive criteria for pruning possible query answers (i.e., answers that are not completely trusted are not produced). It is not trivial to modify consistent query answering to obtain query results that are partially trusted (i.e., answers that are correct in a subset of possible repairs). Moreover, materializing possible clean instances of the database is not the main focus of consistent query answering, which is an important task in several applications such as interactive data cleaning, and data exploration.

A probabilistic approach to the problem of repairing FD violations has been proposed in [47]. The authors introduced a probabilistic data model to represent possible repairs of inconsistent databases. Unfortunately, the proposed model imposes a strong constraint on the set of FDs defined over the database. Specifically, any attribute that appears in the right-hand-side of an FD cannot appear in the left-hand-side of another FD. This simplifying assumption enables independent repairing of FD violations. In fact such an assumption allows obtaining optimal repairs (i.e, that have the minimum number of cell changes) in polynomial time.

Repairing Unclean Data and Unclean FDs

In data cleaning scenarios where the given FDs are not completely accurate (e.g., due to domain evolution or data integration), we should consider changing the FDs at the same time as changing the data. An approach to simultaneously repair data and FDs have been proposed by Chiang and Miller in [22]. Given an input instance I and a set of FDs Σ, the authors proposed a technique to obtain a single repair (Σ', I') that is close to (Σ, I). A unified cost model is proposed to measure the distance between any repair (Σ', I') and the inputs (Σ, I). The proposed algorithm aims at obtaining a single repair with the minimum

cost. Because obtaining an optimal repair is intractable, the proposed algorithm depends on several heuristics to obtain a repair that is not necessarily optimal.

In the extreme case where the data is completely trusted, the cleaning problem boils down to tuning the given set of FDs to fit the input database instance. That is, we start with an initial set of FDs that are not accurate, and we modify this set in a minimal, not-trivial way to be satisfied by the database instance. Tuning a set of FDs is related to the problem of discovering FDs, where the goal is to compute all non-trivial FDs that are satisfied by a database instance [51, 58, 61, 81]). However, in the problem of FD discovery, we start with an empty set of FDs, and we discover new non-trivial FDs that are satisfied by the given database instance.

2.3 Probabilistic Data Management

In this section, we give an overview of probabilistic data management. This line of research is focused on capturing uncertainty in data and efficiently answering queries against uncertain data. In Section 2.3.1, we discuss various data models for capturing uncertainty in data, and in Section 2.3.2, we present current techniques for querying uncertain databases.

2.3.1 Uncertain Data Models

In general, uncertainty in data arises due to various reasons such as imprecision in reading devices, errors in data entry, and incorrect data integration. For example, to compensate imprecision in temperature sensors, a single reading could be replaced by a range of possible values, along with a probability distribution over this range [31].

Several database models that extend the relational model have been proposed to enable representing uncertain data. In general, an uncertain data model represent a number of possible database instances, which are denoted *possible worlds*. For example, assume that some tuples in a database have uncertain existence (i.e., the database may or may not contain such tuples). Additionally, assume that the existence of a tuple is independent of the existence of other tuples. A simple model to capture uncertainty in tuple existence is

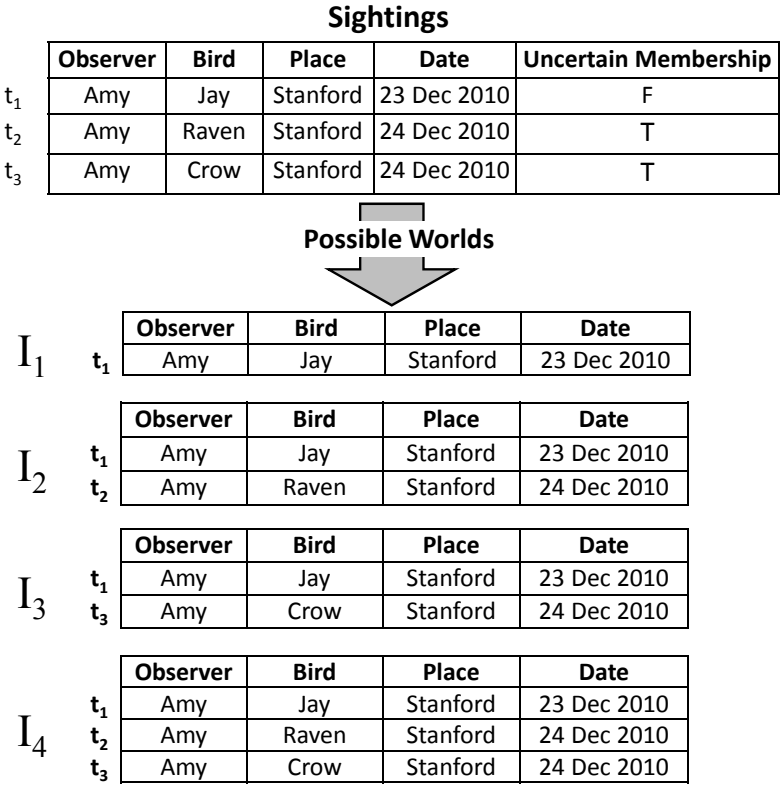

Figure 2.4: An example of an uncertain data model and the corresponding possible worlds

to extend each relation in the database by appending a special Boolean attribute, named `Uncertain Membership`. This attribute is equal to true in tuples with uncertain existence, and equal to false in tuples with certain existence.

Consider the example in Figure 2.4 of a relation `Sightings` that stores bird sightings according to various observers. Attributes of `Sightings` are `Observer`, `Date`, `Place`, `Bird`, in addition to the special attribute `Uncertain Membership`. Figure 2.4 depicts the corresponding possible worlds I_1, I_2, I_3 and I_4, each of which represents a possible instance of relation `Sightings`.

Some uncertain data models have limitations that prevent capturing specific sets of possible worlds. For example, consider the example in Figure 2.4. Assume that tuples t_2 and t_3 are mutually exclusive (i.e., Amy saw a raven or a crow, but not both). This

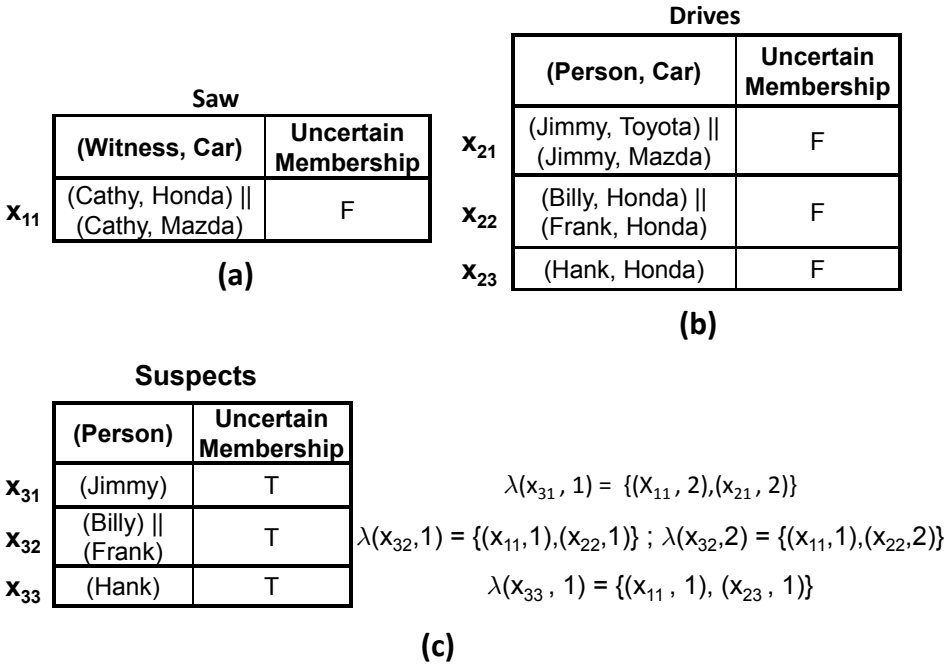

Figure 2.5: ULDB model (a),(b) base relations `Saw` and `Drives` (c) relation `Suspects` resulting from query $\Pi_{Person}(\texttt{Saw} \bowtie \texttt{Drives})$

constraint cannot be captured by the uncertain model in the figure due to assuming that tuples are independent. Several models have been proposed in [71] with different degrees of expressiveness. In general, uncertain data models that are unable to represent any set of possible worlds are said to be *incomplete*. On the other hand, uncertain data models that can represent any set of possible worlds are referred to as *complete* data models [71].

Despite the shortcomings of incomplete models, they are adopted in several systems as they are more intuitive and easier to maintain and query [71]. An incomplete model can be sufficient for a particular system if it is *closed* under all operations that are performed in that system. A model is closed under a given operation if the result of applying this operator on inputs represented by the model can also be represented by the model. For example, the model in Figure 2.4 is closed under selection operation, while it is not closed under join (due to the tuple independence assumption). By definition, complete models are closed under all operations.

More complicated models have been introduced to capture sophisticated dependencies among the database tuples. For example, an uncertain data model, named ULDB, was introduced in [13] that is based on capturing the *lineage information* of tuples (a.k.a., provenance information). ULDB model is proved to be complete, and hence closed under all relational operations. In this model, a relation is defined as a set of *x-tuples*. Each x-tuple is a set of mutually exclusive tuples. Also, existence of an x-tuple can either be certain or uncertain, which is captured by a special attribute `Uncertain Membership`. The lineage information is represented as a function λ that links each tuple in an x-tuple to its base tuples.

For example, Figure 2.5 depicts two relations: `Saw` and `Drives`, and the relation `Suspects` resulting from query $\Pi_{Person}(\texttt{Saw} \bowtie \texttt{Drives})$. The x-tuple x_{32} in relation `Suspects` consists of two tuples. The first tuple `Billy` results from joining the tuple `(Cathy,Honda)` in x-tuple x_{11} with tuple `(Billy,Honda)` in x-tuple x_{22}. Similarly, the second tuple `Frank` is the result of joining `(Cathy,Honda)` and `(Frank,Honda)`. Thus, the lineage of the first tuple in x-tuple x_{32}, denoted as $\lambda(x_{32}, 1)$, consists of $(x_{11}, 1)$ and $(x_{22}, 1)$, and the lineage of the second tuple, denoted as $\lambda(x_{32}, 2)$, consists of $(x_{11}, 1)$ and $(x_{22}, 2)$. Furthermore, the existence of x-tuple x_{32} is uncertain, and hence attribute `Uncertain Membership` is equal to true, because x-tuple x_{11} could be equal to `(Cathy, Mazda)`.

To see how lineage can capture dependencies between tuples, consider a possible world that contains $(x_{31}, 1)$ and $(x_{32}, 1)$ (i.e., the first tuple in x-tuple x_{31}, and the first tuple in x-tuple x_{32}). In absence of any linage information about these tuples, we cannot rule out this possible world. On the other hand, if we know that their linage information contains inconsistent tuples, i.e., $(x_{11}, 2)$ and $(x_{11}, 1)$, we can correctly conclude that this possible world cannot exist.

One extension to the ULDB model allows quantifying the uncertainty about the existence of each tuple by associating a probability to each possible tuple within an x-tuple.

Other directions for modeling uncertainty in data are inspired by the machine learning literature. For example, Bayesian Networks have been used in [72] to model the correlation between tuples existence. In the proposed model, each tuple with uncertain membership is represented as a node in the Bayesian Network graph and the dependencies between tuples

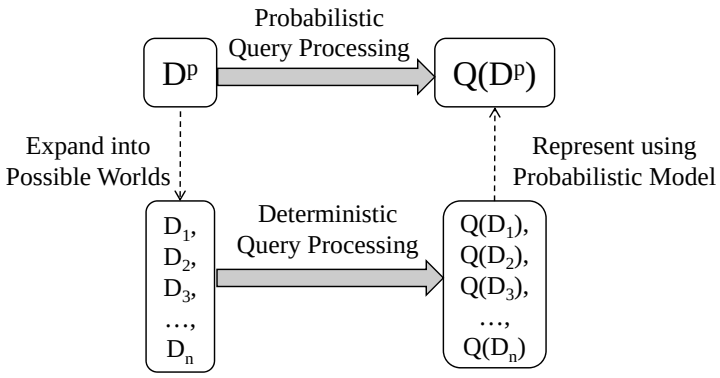

Figure 2.6: Probabilistic query processing using possible worlds semantic

are captured by the graph edges along with the conditional probability tables.

2.3.2 Probabilistic Query Processing

Most approaches for probabilistic data processing rely on the possible worlds semantic to define query answers in the presence of uncertain data. Figure 2.6 shows how query results are interpreted under such a semantic. A query Q that is applied against a probabilistic database D^p can be answered by expanding D^p into all possible worlds $\{D_1, ..., D_n\}$. The query Q is then applied to the individual possible worlds using the semantics of deterministic query processing, resulting in $Q(D_1)$ through $Q(D_n)$. The resulting possible worlds are then captured by the probabilistic data model, which represents $Q(D^p)$. Clearly, processing queries in this way is prohibitively expensive due to the large number of possible worlds. Thus, current approaches aim at performing the query processing directly against the probabilistic database D^p to obtain $Q(D^p)$.

In [30], Dalvi and Suciu defined a class of queries, called *safe queries*, that can be answered in polynomial time. The uncertainty model used in [30] consists of tuples with probabilistic memberships (i.e., each tuple is associated with a value representing its probability of being in the database). Additionally, existence of different tuples in the database is assumed to be independent. The authors focused on a class of queries that involve selection, projection and join operations (called SPJ queries for short). Each relational

operator is extended to compute the membership probability of the resulting tuples. Computation of tuples membership probabilities is based on the independence assumption of tuples existence. For example, joining two tuples produces an output tuple whose membership probability is the product of membership probabilities of the joined tuples. The projection operator (with duplicate elimination) obtains the probability of an output tuple by computing the probability of the disjunction of all its corresponding base tuples in the input relation. Selection does not alter the membership probability of the resulting tuples.

In general, the above model is not closed under projection and join operations as it fails to capture dependencies between tuples. For example, if a tuple t_1 joins with another tuple t_2 to form t_{12} and tuple t_1 joins with tuple t_3 to form t_{13}, existence of tuple t_{12} is not independent from existence of t_{13}, and thus the two tuples cannot be represented using this model.

A certain subclass of query plans, called safe plans, do not introduce dependencies among tuples in intermediate results. Hence, we can correctly compute the probabilities of output tuples while assuming independence among tuples. For example, consider Figure 2.7 that shows two plans to answer the query $\Pi_{Person}(\texttt{Saw} \bowtie \texttt{Drives})$. The left-hand-side plan performs a join operation between the relations `Saw` and `Drives`. The fact that the resulting tuples are not independent is not captured because of the limitations of the uncertainty model, and hence the successive projection operation results in incorrect membership probability of the query result. On the other hand, the right-hand-side plan first computes the relation $\Pi_{car}(\texttt{Saw})$. The results are then joined with the relation `Drives` and the join results are projected on the attribute `Person`. The latter plan does not produce intermediate relations with dependent tuples, and thus the computed marginal probabilities are correct.

Safe queries are those that have at least one safe plan to obtain query results. For example, if the attribute `Witness` of the relation `Saw` is included as a join key in Figure 2.7, no safe plan can be found to answer the query. For unsafe queries, approximation techniques such as Monte Carlo simulation [56] are used.

Benjelloun et al. introduced in [13] a probabilistic data processing system, called Trio, that is based on the lineage-based probabilistic data model ULDB (Section 2.3.1). The

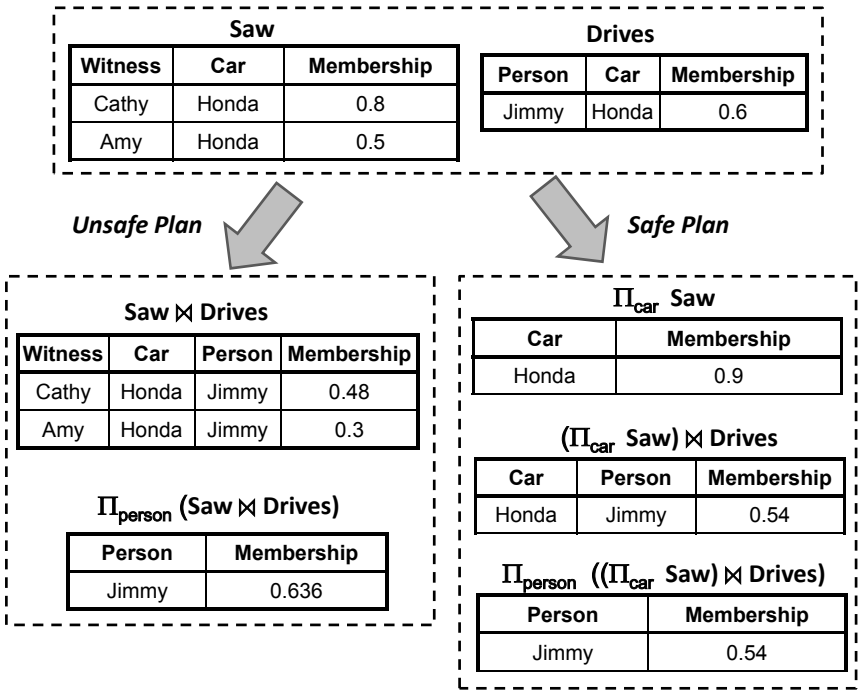

Figure 2.7: An example of safe and unsafe plans for query $\Pi_{Person}(\texttt{Saw} \bowtie \texttt{Drives})$ [30]

lineage information of the stored tuples allows tracking the sources the tuples, and providing correct membership probabilities. The authors described how to update the lineage of database tuples as query operators are applied over the database relations. For example, in Figure 2.5, joining x-tuple x_{11}, `(Cathy, Honda)||(Cathy, Mazda)`, and x-tuple x_{22}, `(Billy, Honda)||(Frank, Honda)`, on attribute `Car` results in an x-tuple `(Cathy, Honda, Billy)||(Cathy, Honda, Frank)`. The lineage of the first tuple in the resulting x-tuple is conjunction $(x_{11}, 1) \wedge (x_{22}, 1)$, and the lineage of the second tuple is conjunction $(x_{11}, 1) \wedge (x_{22}, 2)$. Other operators such as projection with duplicate elimination result in lineage with disjunctions of base tuples. Computation of tuples probabilities is decoupled from obtaining the output relation in order to allow deferring probability computation until requested by the user.

In [55], Jampani et al. have introduced a probabilistic query processing system, named MCDB, that is based on Monte Carlo simulation. The authors rely on Monte Carlo sim-

ulation to enable efficient answering of virtually any query types that were not possible in other systems, such as aggregate queries and complex nested queries. In MCDB, relations are divided into normal (deterministic) relations, and random relations whose tuples have uncertain existence in the relation, and uncertain attributes. Each random relation is split into blocks such that tuples belonging to different blocks are independent (i.e., have independent existence, and independent attribute values). Random relations are associated with value generating (VG) functions that are responsible for generating a number of possible instances of each block at query time. That is, each call to a VG function generates one instance of a given block. Implementations of VG functions range from standard probability functions (e.g., Gaussian distribution, Gamma distribution, and Multivariate distributions), to complex functions written in C language.

A simple strategy to answer a query using Monte Carlo simulation is to materialize a number of possible instances, say N, of each random relation, compute the query answers for each possible world, and aggregate the generated tuples to count the number of possible worlds in which each tuple exists. In MCDB, the authors proposed several query optimization techniques to significantly reduce the cost of query answering compared to this simple strategy. For example, for each uncertain tuple t, different versions of t in the possible instances are *bundled* together and represented internally as an array of tuples $t[1], \ldots, t[N]$. The user query is applied only once against tuple bundles, and thus the query optimization is performed only once instead of N times. Also, each tuple bundle is processed at the same time by a given query operator, which can save computation cycles. For example, if all versions in a bundle have the same value for attribute A, a selection predicate based on A can either accept or reject the entire bundle using a single comparison. Finally, MCDB materializes possible instances of random relations at query time only when necessary, and thus avoids costly materialization whenever possible.

Chapter 3

Modeling Uncertainty in Duplicate Elimination

In this chapter, we present our approach for probabilistic duplicate elimination [16]. In Section 3.1, we define multiple spaces of possible repairs. Our cleaning approach is given in Section 3.2. In Section 3.3, we discuss how to support relational queries. In Section 3.4, we discuss implementing our probabilistic data model inside relational DBMSs and we present new query types that are supported by our system. In Section 3.5, we describe how to model uncertainty in merging duplicate tuples. An experimental evaluation is given in Section 3.6.

3.1 Spaces of Possible Repairs

In this section, we define the space of all possible repairs of a given database instance that contains duplicate tuples. We also describe multiple approaches to limit the space size for efficient processing.

In the context of duplicate detection, a repair represents a clustering of the input tuples, where each cluster contains tuples that refer to the same real-world entity. We formally define a possible repair as follows.

Definition 1. Repair of Duplicate Tuples. *Given a relation instance I with duplicate tuples, a repair X is a set of disjoint tuple clusters $\{C_1, \ldots, C_m\}$ such that $\bigcup_{i=1}^{m} C_i = I$.*

That is, a repair X is a partition of tuples in I. By coalescing each cluster of tuples into a representative tuple, we obtain a clean (duplicate-free) instance I'. We assume in this section that coalescing members of clusters is performed in a deterministic way, and we discuss in Section 3.5 extending our approach to consider uncertainty in coalescing tuples.

Repairs have clear analogy to the concept of *possible worlds* in uncertain databases [4, 30, 53]. Possible worlds are all possible database instances originating from tuple and/or attribute uncertainty. However, in our settings, the repairs emerge from uncertainty in deciding whether a set of tuples are duplicates or not.

In general, there are two key problems when dealing with the space of all possible repairs. First, the number of possible repairs is as large as the number of possible clusterings of tuples in I, which is exponential in the number of tuples (by correspondence to the problem of set partitioning [10]). Second, quantifying the confidence in each possible repair by, for example, imposing a probability distribution on the space of possible repairs, is not clear without understanding the underlying process that generates the repairs.

There are multiple ways to constrain the space of possible repairs such as imposing hard constraints to rule out impossible repairs, or filtering the repairs that do not meet specific requirements (e.g., pairwise distance among clustered tuples must be larger than a given threshold). In our work, we consider the subset of all possible repairs that are valid output of a parameterized clustering algorithm. In other words, given a parameterized clustering algorithm, we limit the space of possible repairs to those generated by the algorithm using different parameter settings. This approach has two effects that are described as follows.

1. Limiting the space of possible repairs improves the efficiency of generating and querying the repairs, and reduces the space required to store the repairs.

2. By assuming (or learning) a probability distribution on the values of the algorithm parameters, we can induce a probability distribution on the space of possible repairs, which allows for a richer set of probabilistic queries (e.g., finding the most probable

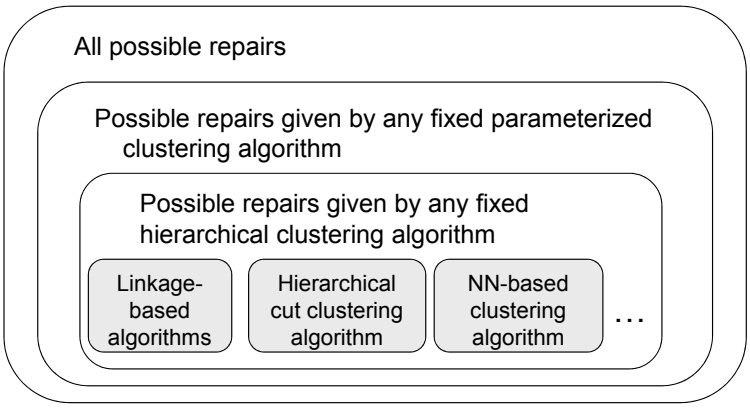

Figure 3.1: Constraining the space of possible repairs

repair, finding the probability of clustering two specific tuples together, or finding the marginal probability of a predicate).

Constraining parameterized algorithms to a specific class of algorithms can further reduce the time required for generating the repairs. For hierarchical clustering algorithms, the size of the space of possible repairs is linear in the number of tuples in the unclean relation (we give more details in Section 3.2.2). Moreover, a hierarchical clustering algorithm can be modified in a simple way to efficiently generate the possible repairs by a single run of the algorithm.

Figure 3.1 depicts the containment relationship between the space of all possible repairs, and the possible repairs generated by any given parameterized algorithm. Figure 3.1 also shows examples of hierarchical clustering methods that we discuss in Section 3.2.2.

3.2 Modeling Possible Repairs

In this Section, we provide a probabilistic data model to represent a set of possible repairs. We focus on modeling the space of possible repairs generated by any fixed parameterized clustering algorithm.

There are multiple approaches that can be adopted to model the possible repairs. In the following, we present two extremes within the spectrum of possible representations.

- The first representation is the triple $(I, \mathcal{A}, \mathcal{P})$, where I denotes the unclean relation instance, $\mathcal{A}$ denotes a fixed parameterized clustering algorithm, and $\mathcal{P}$ denotes a set of possible parameter settings for the algorithm $\mathcal{A}$. This approach is a compact representation that does not materialize any possible repairs, and thus no construction cost is incurred.
- The second representation is the set of all possible clean instances $\{I_1, \ldots, I_{|\mathcal{P}|}\}$ that can be generated by the algorithm $\mathcal{A}$ using all possible parameter settings in $\mathcal{P}$.

Other representations between these two extremes involve (partial) materialization of possible repairs and storing views that aggregate these repairs. For example, a possible representation is to associate each pair of tuples with the relative frequency of repairs in which both tuples belong to the same cluster (i.e., declared as duplicates). The problem of finding a suitable view of the possible repairs is analogous to the problem of selecting which materialized views to build in relational databases. Choosing a suitable view depends on several factors such as the cost of materializing the view and the types of queries that can be answered using that view as we illustrate in Example 2.

Example 2. *Consider two sets of possible repairs, denoted A and B, that involve the base tuples* $\{t_1, t_2, t_3\}$ *as shown in Figure 3.2. For all pairs of tuples, the relative frequency of repairs in which the two tuples are clustered together is the same with respect to both sets of repairs. These frequencies are shown in the symmetric matrix in Figure 3.2.*

The view consisting of the pair-wise clustering frequencies depicted in Figure 3.2 can be used to efficiently answer some queries (e.g., is there any repair in which t_1 and t_2 are clustered together). However, Example 2 shows that the proposed view is a lossy representation of repairs. That is, this view cannot be used to restore the encoded set of possible repairs. Therefore, some queries might be impossible to answer using such a representation. For example, finding the relative frequency of repairs in which t_1, t_2 and t_3 are clustered together is not possible using the view in Figure 3.2.

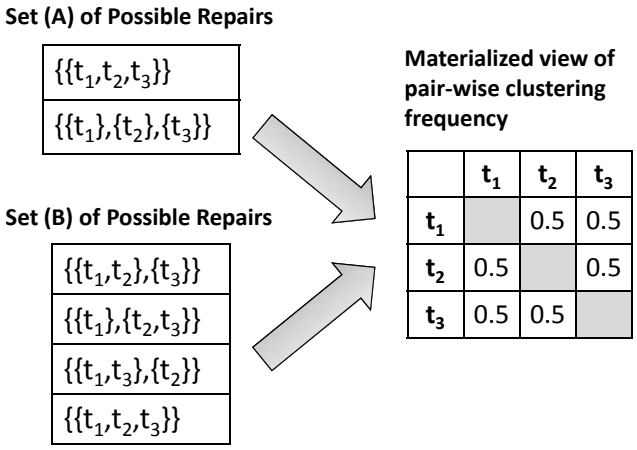

Figure 3.2: Two sets of possible repairs represented by the same matrix of pair-wise clustering frequencies

We summarize our proposed desiderata regarding modeling the possible repairs as follows.

- The model should be a *lossless* representation of the possible repairs in order to allow answering queries that require a complete knowledge about these repairs. In other words, we have to ensure that the possible repairs can be restored from the model.
- The model should allow efficient answering of a set of important queries types (e.g., selection, projection and join queries, which are frequently used in practice).
- The model should provide materialization of the results of costly operations (e.g., clustering procedures) that are required by most queries.
- The model should have a small space complexity to allow efficient construction, storage and retrieval of the possible repairs, in addition to efficient query processing.

In Section 3.2.1, we describe our proposed model that addresses the aforementioned requirements. In Section 3.2.2, we show how to efficiently obtain the possible repairs for the class of hierarchical clustering algorithms.

3.2.1 Algorithm-Dependent Model

In this section, we introduce a probabilistic data model to encode the space of possible repairs generated by any fixed parameterized clustering algorithm $\mathcal{A}(I, \tau)$ with a single parameter τ. We denote by the range $[\tau^l, \tau^u]$ all valid values that can be assigned to τ.

We view the algorithm parameter τ as a random variable such that the probability of a parameter value v is equal to the probability that the clustering $\mathcal{A}(I, \tau)$ has the highest quality (based on some quality metric) among all other clusterings of I generated by $\mathcal{A}$ at $\tau \neq v$. We denote by f_τ the probability density function of τ defined over $[\tau^l, \tau^u]$. In Section 6.2.3, we discuss possible directions for learning the probability distribution function f_τ.

The set of possible repairs $\mathcal{X}$ is defined as $\{\mathcal{A}(I, v) : v \in [\tau^l, \tau^u]\}$. The set $\mathcal{X}$ defines a probability space created by drawing random 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 $X \in \mathcal{X}$ of I, denoted $\Pr(X)$, is derived as follows.

$$\Pr(X) = \int_{\tau^l}^{\tau^u} f_\tau(v) \cdot h(\mathcal{A}(I, v), X)\ dv \tag{3.1}$$

where $h(A, B)$ is an indicator function that is equal to 1 if $A = B$, and 0 otherwise.

In the following, we define an uncertain clean relation (a *U-clean relation* for short) that encodes the possible repairs $\mathcal{X}$ of an unclean instance I of relation R that are generated by a parameterized clustering algorithm $\mathcal{A}$.

Definition 2. ***U-Clean Relation.*** *A U-clean relation, denoted I^c, is a set of c-tuples where each c-tuple is a representative tuple of a cluster of tuples in I. Attributes of I^c include all attributes of R, in addition to two special attributes: C and P. Attribute C of a c-tuple is the set of tuples identifiers in I that are clustered together to form this c-tuple. Attribute P represents the parameter values of the clustering algorithm $\mathcal{A}$ that lead to clustering tuples in C.*

The parameter settings P is represented as one or more intervals within the range of the algorithm parameter τ. We interpret each c-tuple t as a propositional variable, and each repair $X \in \mathcal{X}$ as a truth assignment for all c-tuples in I^c such that $t = \texttt{True}$ if tuples in attribute C of t form a cluster in X, and $t = \texttt{False}$ otherwise. Note that it is possible to have overlapping clusters represented by different c-tuples in I^c since I^c encapsulates more than one possible repair of I.

Figure 3.3 illustrates our model of possible repairs for two unclean relations `Person` and `Vehicle`. U-clean relations $\texttt{Person}^c$ and $\texttt{Vehicle}^c$ are created by clustering algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ using parameters τ_1 and τ_2, respectively. For brevity, we omit some attributes from $\texttt{Person}^c$ and $\texttt{Vehicle}^c$ (shown as dotted columns in Figure 3.3). Parameters τ_1 and τ_2 are defined on the real interval $[0, 10]$ with uniform distributions. We provide more details of the construction process in Section 3.2.2. Relations $\texttt{Person}^c$ and $\texttt{Vehicle}^c$ capture all repairs of the base relations corresponding to possible parameters values. For example, if $\tau_1 \in [1, 3)$, 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-tuples in $\texttt{Person}^c$ whose parameter settings contain the interval $[1, 3)$. Moreover, the U-clean relations allow for identifying the parameter settings of the clustering algorithm that lead to generating a specific cluster of tuples. For example, the cluster $\{\texttt{P1}, \texttt{P2}, \texttt{P5}\}$ is generated by algorithm $\mathcal{A}_1$ if the value of parameter τ_1 belongs to the range $[3, 10]$.

3.2.2 Constructing U-clean Relations

Hierarchical clustering algorithms cluster tuples of an input instance I in a hierarchy, which represents a set of possible clusterings starting from a clustering containing each tuple in a separate cluster, to a clustering containing all tuples in one cluster (e.g., Figure 3.4). The algorithms use specific criteria, usually involves a parameter of the algorithm, to determine which clustering to return.

Hierarchical clustering algorithms are widely used in duplicate detection. Examples include link-based algorithms (e.g., single-linkage, average-linkage and complete-linkage) [54], hierarchical cut clustering [39], and CURE [48]. Other algorithms can be altered to allow producing hierarchical clustering of tuples such as the fuzzy duplicate detection

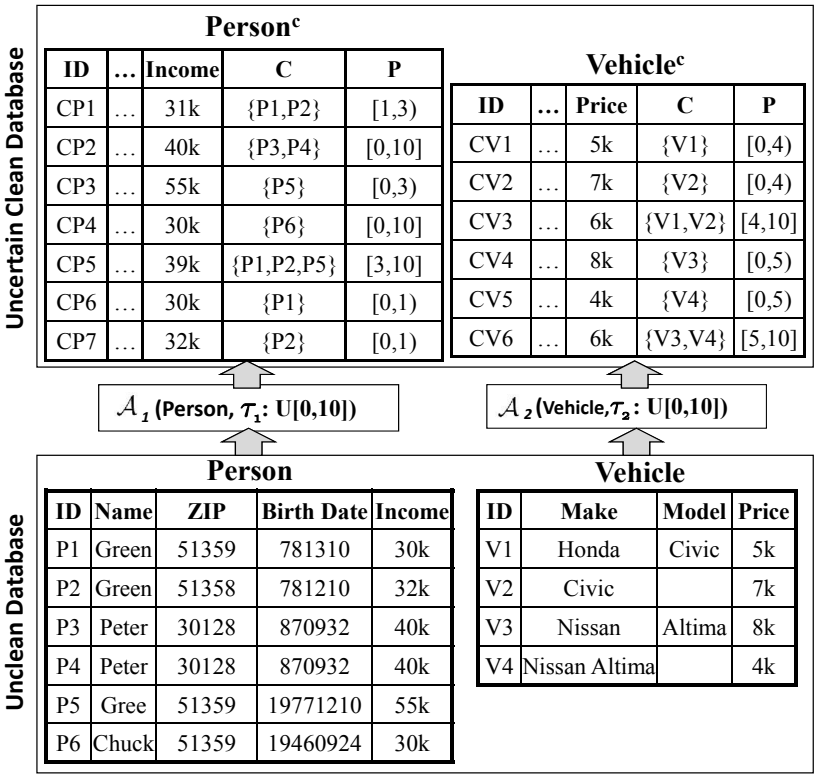

Figure 3.3: An example illustrating the U-clean model

framework introduced in [20], as we show later in this section. Hierarchical clustering is also used as a basis for other duplicate detection algorithms such as collective entity resolution [17], and deduplication under aggregate constraints [21].

Due to the nature of hierarchical clustering algorithms, only simple modifications are necessary to allow constructing U-clean relations as we discuss in the following case studies.

Case Study 1: Link-based Hierarchical Clustering Algorithms

Given an input unclean instance I consisting of n tuples, a hierarchical linkage-based clustering algorithm generally rely on two parameters: (1) a distance function $dist(C_i, C_j)$, where C_i and C_j are two disjoint clusters, and (2) a stopping condition (e.g., terminate the clustering when the distance between all pairs of clusters is greater than a given threshold

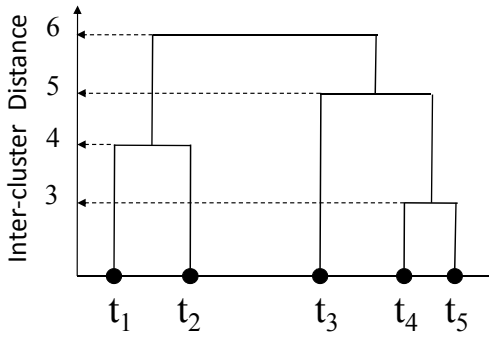

Figure 3.4: An example of link-based hierarchical clustering

τ). Clusters are merged iteratively starting from all singleton clusters. At each iteration, the function *dist* is used to pick the closest two clusters to link. If the value of *dist* of such a pair is below threshold τ, the two clusters are merged by creating a parent cluster in the hierarchy composed of the union of the two original clusters. If the distance between the closest clusters is greater than τ, the algorithm terminates and return the obtained clustering.

The distance between two tuples is determined through various functions such as Euclidian distance, Edit Distance, and Q-grams [33]. The distance between two clusters is an aggregate of the pair-wise distances. For example, in single-linkage [54], *dist* returns the distance between the two closest tuples in the two clusters, while in complete-linkage, *dist* is the distance between the two furthest tuples in the two clusters.

Figure 3.4 gives an example of the hierarchy generated by a linkage-based algorithm for the instance $I = \{t_1, \ldots, t_5\}$. The parameter τ represents a threshold on inter-cluster distances which are represented by the Y-axis. Different repairs are generated when applying the algorithm with different values of τ. For example, for $\tau \in [0, 3)$, the produced repair is $\{\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}, \{t_5\}\}$, while $\tau \in [3, 4)$ produces the repair $\{\{t_1\}, \{t_2\}, \{t_3\}, \{t_4, t_5\}\}$.

We modify the link-based clustering algorithm to build U-clean relations as described in Algorithm 1. The algorithm performs clustering of tuples similar to the conventional greedy agglomerative clustering algorithm. However, we additionally create and store all c-tuples corresponding to the clusters linked at distances within the range $[\tau^l, \tau^u]$.

Initially, the algorithm creates a singleton cluster and its corresponding c-tuple for each tuple in I (lines 1-5). The initial parameter settings of created c-tuples are the entire range $[\tau^l, \tau^u]$. The algorithm incrementally merges the closest clusters C_i and C_j, and creates a c-tuple corresponding to the new cluster (lines 6-10). Additionally, we update the c-tuples corresponding to C_i and C_j as shown in lines 11-15. The algorithm terminates when the distance between the closest clusters exceeds τ^u or when all tuples are clustered together.

Algorithm 1 `U_Cluster`(I, τ^l, τ^u)

Require: I: the unclean instance
Require: τ^l: Minimum threshold value
Require: τ^u: Maximum threshold value
1: Define a new singleton cluster C_i for each tuple $t_i \in I$
2: $\mathcal{C} \leftarrow \{C_1, \ldots, C_{|I|}\}$
3: **for each** $t_i \in I$ **do**
4: Add the c-tuple $(t_i[A_1], \ldots, t_i[A_m], C_i, [\tau^l, \tau^u])$ to I^c
5: **end for**
6: **while** ($|\mathcal{C}| > 1$ and distance between the closest pair of clusters (C_i, C_j) in $\mathcal{C}$ is less than or equal to τ^u) **do**
7: $C_k \leftarrow C_i \cup C_j$
8: Replace C_i and C_j in $\mathcal{C}$ with C_k
9: $t_k \leftarrow$ `get_representative_tuple`(C_k) {See Section 3.2.3}
10: Add the c-tuple $(t_k[A_1], \ldots, t_k[A_m], C_k, [dist(C_i, C_j), \tau^u])$ to I^c
11: **if** $(dist(C_i, C_j) < \tau^l)$ **then**
12: Remove the c-tuples corresponding to C_i and C_j from I^c
13: **else**
14: Set the upper bounds of the parameter settings of the c-tuples corresponding to C_i and C_j in I^c to $dist(C_i, C_j)$
15: **end if**
16: **end while**
17: **return** I^c

Case Study 2:NN-Based Clustering

In [20], a duplicate detection algorithm based on the nearest neighbor (NN) techniques is introduced. The algorithm introduced in [20] declares a set of tuples as duplicates whenever they represent a *compact set* that has *sparse neighborhood*. A set of tuples S is compact if $\forall t \in S$, the distance between t and any other tuple in S is less than the distance between t and any tuple not in S (i.e., tuples in S are mutual k nearest neighbors where $k = |S| - 1$). The neighborhood growth of a tuple t, denoted $ng(t)$, is defined as the number of tuples with distance to t smaller than double the distance between t and its nearest neighbor. A set S has a sparse neighborhood if its aggregated neighborhood growth $F_{r \in S}\ ng(r)$ is less than a threshold τ, where F is an aggregate function such as max or $average$.

Although the NN-based clustering algorithm in [20] is not presented as a hierarchical clustering algorithm, it can be used for producing hierarchical clusterings. Compact sets are arranged in a hierarchy due to the fact that compact sets are nested (i.e., any two different compact sets are either disjoint or the smaller compact set is a subset of the larger compact set). That is, $\forall S_i, S_j, i \neq j (S_i \cap S_j = \phi \vee S_i \subset S_j \vee S_j \subset S_i)$. We verify this fact by contradiction. Assume that $\exists S_i, S_j, i \neq j (S_i \cap S_j \neq \phi \wedge S_i \setminus S_j \neq \phi \wedge S_j \setminus S_i \neq \phi)$ (the operator $\setminus$ denotes the set difference). Let k_i and k_j denote the cardinality of S_i and S_j, respectively, and assume, without loss of generality, that $k_i < k_j$. According to the definition of compact sets, the $k_i - 1$ nearest neighbors of a tuple $t \in S_i$ are equal to $S_i - \{t\}$, and similarly, the $k_j - 1$ NNs of $t \in S_j$ are equal to $S_j - \{t\}$. For $t \in (S_i \cap S_j)$, the $k_i - 1$ NNs of t are not contained in the set of the $k_j - 1$ NNs of t because $S_j \setminus S_i \neq \phi$. This contradicts the fact that $k_i - 1$ NNs of t must be contained in the $k_j - 1$ NNs of t, assuming that the k-NNs of t are uniquely defined (i.e., ties in distance are broken in a deterministic way). Thus, the nesting property of the compact sets is correct.

If the used aggregation function F is max (or any other monotone function with respect to the size of a compact set), increasing τ results in a monotonic decrease of the number of clusters by merging two or more compact sets into one compact set. The reason is that for any two compact sets S_i, S_j such that $S_i \subset S_j$, $\max_{r \in S_i}\ ng(r) \leq \max_{r \in S_j}\ ng(r)$. Thus, the NN-based clustering algorithm effectively constructs a hierarchy of compact sets where neighborhood spareness is used as the stopping condition.

In order to allow efficient construction of U-clean relations , we modify the NN-based

clustering algorithm similar to the link-based algorithms. We construct compact sets incrementally, starting with singleton compact sets until reaching compact sets of the maximum size allowed (using the same technique in [20]). Each compact set with aggregated neighborhood growth above τ^l and below τ^u is stored in I^c. For any two compact sets S_i, S_j such that $S_i \subset S_j$ and they have the same neighborhood growth, we only store S_j in I^c in order to comply with a property of the algorithm in [20], which is to report the *largest* compact sets that satisfy the sparse neighborhood criterion. Parameter settings are maintained for each c-tuple similar to the linkage-based algorithms.

Time and Space Complexity

In general, our modified hierarchical clustering algorithms have the same asymptotic complexity of the unmodified algorithms. The reason is that we only add a constant amount of work to each iteration which is constructing c-tuples and updating their parameter settings (e.g., lines 9-15 in Algorithm 1).

The space complexity of a U-clean relation I^c is $O(n)$, where n is the number of tuples in the input instance I. Hierarchical clustering arranges tuples in the form of an N-ary tree. The leaf nodes in the tree are the tuples in the unclean relation instance I, while the internal nodes are clusters of tuples that contain two or more tuples. Let n be the size of I, and n' be the number of clusters containing two or more tuples (the number of internal nodes). The maximum value of n' occurs when the tree is binary, in which n' is equal to $n-1$. Thus, the total number of nodes in the clustering hierarchy is less than or equal to $n' + n = 2n - 1$. The size of I^c is equal to the number of the possible clusters, which is bounded by $2n - 1$. It follows that the size of I^c is linear in the number of tuples in I.

The number of repairs encoded by I^c is less than or equal to $n' + 1 = n$. The reason is that, besides the initial repair where each tuples is in a singleton cluster, each internal node indicates merging multiple clusters together, resulting in a new repair.

3.2.3 Representative Tuples of Clusters

Tuples in the same cluster within a repair indicate duplicate references to the same real-world entity. In order to obtain a clean instance, we need to resolve potential conflicts

between attributes of duplicate tuples. We assume that conflicts in attribute values of the cluster tuples are resolved deterministically using a user defined merging procedure (line 9 in Algorithm 1), which can be decided based on the data semantics. For example, conflicting values of attributes `Income` and `Price` in Figure 3.3 are resolved by using their average as a representative value.

Note that deterministically resolving conflicts in attribute values of tuples that belong to the same cluster may lead to loss of information and introduce errors in the generated repairs. The technique proposed in [6] tackled this problem by modeling uncertainty in merging tuples. The authors assume that a representative tuple for a cluster is a random variable whose possible outcomes are all members of the cluster. We see uncertainty in the merging operation as another level of uncertainty that can be combined in our framework. For the sake of clarity, we focus on uncertainty in clustering tuples in the following sections, and we describe how to extend our approach to handle uncertain merging in Section 3.5.

3.3 Query Processing

We define relational queries over U-clean relations using the concept of the *possible worlds semantic* [14, 30, 72] (refer to Section 2.3.2). According to the possible worlds semantic, queries are conceptually answered against individual clean instances of the unclean database that are encoded in the U-clean relations, and the resulting answers are re-encoded in a U-clean relation. Furthermore, the marginal probability of a query answer is equal to the sum of probabilities of possible worlds (clean instances) in which such answer is true. For example, consider a selection query that reports persons with `Income` greater than 35k considering all repairs encoded by $\texttt{Person}^c$ in Figure 3.3. One qualified tuple is `CP3`. This tuple is valid only for repairs generated at the parameter settings $\tau_1 \in [0, 3)$. Therefore, the probability that tuple `CP3` belongs to the query result is equivalent to the probability that τ_1 is within $[0, 3)$, which is 0.3 (assuming that τ_1 is uniformly distributed over the range $[0, 10]$).

In the following, we describe how to support multiple query types under our model such as selection, projection, join and aggregation.

3.3.1 SPJ Queries

In this section, we define the selection, projection and join (SPJ) operators over U-clean relations.

Model closure under SPJ queries is important in order to allow query decomposition (i.e., applying operators to the output of other operators). To make our model closed under SPJ operations, we extend the definition of attribute C in U-clean relations to be a composition of multiple clusters, and extend attribute P to be a composition of multiple parameter settings of one or more clustering algorithms. Similar methods are proposed in [30], where each tuple is associated with a complex probabilistic event.

We interpret attributes C and P of a c-tuple t as propositional variables that are true for repairs containing t and false for all other repairs. For example, consider c-tuple $CP2$ in Figure 3.3. The value $\{P3, P4\}$ of attribute C represents a propositional variable that is true iff the two tuples $P3$ and $P4$ are clustered together. Similarly, the value $[0, 10]$ of attribute P represents a variable that is true iff the parameter τ_1 belongs to the interval $[0, 10]$.

For U-clean relations resulting from SPJ queries, we define attributes C and P as *propositional formulae* in DNF over attributes C and P of the base U-clean relations, respectively. For example, consider joining two c-tuples $CP2$ and $CV3$ in Figure 3.3. Attribute C of the resulting c-tuple is $\{P3, P4\} \wedge \{V1, V2\}$, and attribute P is $\tau_1 \in [0, 10] \wedge \tau_2 \in [4, 10]$.

Note that the propositional formulae of attributes C and P of a c-tuple t are identical formulae defined on different variables, which are the clusters and the parameter settings of the base c-tuples. That is, a DNF formula of attribute C of a c-tuple can be converted to the DNF formula of attribute P of the same c-tuple by replacing every cluster in C with the corresponding parameter settings (e.g., replacing $\{P3, P4\}$ with $\tau_1 \in [0, 10]$ and replacing $\{V1, V2\}$ with $\tau_2 \in [4, 10]$ in the previous example).

SPJ operators that are applied to U-clean relations are *conceptually* processed against all clean instances represented by the input U-clean relations, and the resulting instances are re-encoded into an output U-clean relation. We add a superscript u to the operators

symbols to emphasize awareness of the *uncertainty* encoded in the U-clean relations. In the following, we show how to efficiently evaluate SPJ queries without an exhaustive processing of individual repairs (similar to the concept of *intensional query evaluation* [30]).

Selection

We define the selection operator over U-clean relations, denoted σ^u, as follows: $\sigma^u_p(I^c) = \{t : t \in I^c \wedge p(t) = True\}$, where p is the selection predicate defined over attributes in R. That is, a selection query $\sigma^u_p(I^c)$ results in a U-clean relation containing the c-tuples in I^c that satisfy the predicate p. The operator σ^u does not change attributes C or P of the resulting c-tuples.

For example, Figure 3.5(a) shows the result of a selection query against $\texttt{Person}^c$ in Figure 3.3, where we are interested in finding persons with income greater than 35k. The query produces three c-tuples that are identical to the input c-tuples `CP2`, `CP3`, and `CP5`.

Projection

We define the projection operator Π^u over a U-clean relation as follows. The expression $\Pi^u_{A_1,\ldots,A_k}(I^c)$ returns a U-clean relation that encodes projections of all clean instances represented by I^c on attributes $A_1, \ldots, A_k$ that belong to R. The schema of the resulting U-clean relation is $(A_1, \ldots, A_k, C, P)$. Under bag semantics, duplicate c-tuples are retained. Hence, attributes C and P of the projected c-tuples remain unchanged. Under set semantics, c-tuples with identical values with respect to attributes $A_1, \ldots, A_k$ are reduced to one c-tuple with attributes C and P computed as follows. Let $t' \in \Pi^u_{A_1,\ldots,A_k}(I^c)$, where t' is a projected c-tuple corresponding to duplicate c-tuples $\{t_1, \ldots, t_r\} \subseteq I^c$. Attribute C of t' is equal to $\bigvee_{i=1}^{r} t_i[C]$ and attribute P of t' is equal to $\bigvee_{i=1}^{r} t_i[P]$.

For example, Figure 3.5(b) shows the results of a projection query (under set semantics) posed against Relation $\texttt{Vehicle}^c$ in Figure 3.3, where we are interested in finding the distinct car prices. The only duplicate c-tuples with respect to attribute `Price` are `CV3` and `CV6`.

SELECT ID, Income
FROM Person^c
WHERE Income>35k

ID	Income	C	P
CP2	40k	{P3,P4}	[0,10]
CP3	55k	{P5}	[0,3)
CP5	39k	{P1,P2,P5}	[3,10]

(a)

SELECT DISTINCT Price
FROM Vehicle^c

Price	C	P
4k	{V4}	[0,5)
5k	{V1}	[0,4)
6k	{V1,V2}∨{V3,V4}	[4,10] ∨ [5,10]
7k	{V2}	[0,4)
8k	{V3}	[0,5)

(b)

SELECT Income, Price
FROM Person^c , Vehicle^c
WHERE Income/10 >= Price

Income	Price	C	P
40k	4k	{P3,P4} ^ {V4}	τ_1:[0, 10] ^ τ_2:[0,5)
55k	5k	{P5} ^{V1}	τ_1:[0, 3) ^ τ_2:[0,4)
55k	4k	{P5} ^ {V4}	τ_1:[0, 3) ^ τ_2:[0,5)

(c)

Figure 3.5: Relational queries (a) selection (b) projection (c) join

Join

We define the join operator $\bowtie^u$ over two U-clean relations as follows. The expression $(I_i^c \bowtie_p^u I_j^c)$ results in a U-clean relation that contains all pairs of c-tuples in I_i^c and I_j^c that satisfy the join predicate p that is defined on attributes in R_i and R_j (the schemas of input instances I_i and I_j, respectively). The schema of the resulting relation is $(A_{i1}, \ldots, A_{im}, A_{j1}, \ldots, A_{jp}, C, P\}$,where $R_i = (A_{i1}, \ldots, A_{im})$ and $R_j = (A_{j1}, \ldots, A_{jp})$. We compute attributes of the resulting c-tuples as follows. Let t_{ij} be the result of joining $t_i \in I_i^c$ and $t_j \in I_j^c$. For attribute $A \in R_i$, $t_{ij}[A] = t_i[A]$, and similarly for $A \in R_j$, $t_{ij}[A] = t_j[A]$. Furthermore, $t_{ij}[C] = t_i[C] \wedge t_j[C]$ and $t_{ij}[P] = t_i[P] \wedge t_j[P]$.

For example, Figure 3.5(c) shows the results of a join query on Relations `Person`^{c} and `Vehicle`^{c} in Figure 3.3. The query finds which car is likely to be purchased by each person by joining a person with a car if 10% of the person's income is greater than or equal to the car's price. Note that the parameter settings of c-tuples in the join results involve two

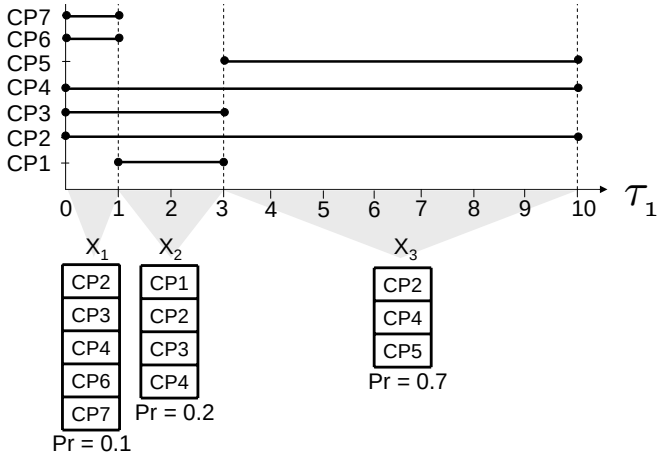

Figure 3.6: Distribution of possible repairs in instance $\texttt{Person}^c$

parameters: τ_1 and τ_2. Therefore, we precede each interval in attribute P with the referred parameter to avoid ambiguous settings.

3.3.2 Aggregation Queries

The aggregation query $Agg(I^c, expr)$ uses the function Agg (such as sum, $count$, and min) to aggregate the value of the expression $expr$ over all c-tuples in I^c. The expression $expr$ is defined on attributes in R (the schema of the unclean instance I). Examples of $expr$ include a single attribute in R, or a function defined on one or more attributes. The result of an aggregation query against one clean database instance is a single scaler value. However, in our settings, I^c encodes multiple possible clean instances. Hence, the answer of an aggregation query over I^c is a probability distribution over possible answers, each of which is obtained from one or more clean possible instances. To simplify the discussion, we assume that the aggregate query involves a base U-clean relation I^c that is generated by a clustering algorithm $\mathcal{A}$ using a single parameter τ. We discuss at the end of this section how to answer aggregation queries over U-clean relations resulting from SPJ queries.

For example, consider the aggregation query $average(\texttt{Person}^c, \texttt{Income})$, where we are interested in finding the average of persons' incomes, given the possible repairs repre-

sented by $\texttt{Person}^c$ in Figure 3.3. Figure 3.6 shows the possible repairs of Relation Person, which are $\{CP2, CP3, CP4, CP6, CP7\}$, $\{CP1, CP2, CP3, CP4\}$ and $\{CP2, CP4, CP5\}$ whose probabilities are 0.1, 0.2 and 0.7, respectively. The aggregate value for the three repairs are 37.4k, 39k and 36.33k, respectively. Hence, the query answer is the following discrete probability distribution: $\Pr(average = 37.4k) = 0.1$, $\Pr(average = 39k) = 0.2$, $\Pr(average = 36.33k) = 0.7$.

A straightforward algorithm to answer aggregation queries over a U-clean relation I^c is described as follows.

1. Identify the distinct end points $e_1, \ldots, e_q$ (in ascending order) that appear in attribute P of all c-tuples in I^c. Define V_i to be the interval $[e_i, e_{i+1}]$ for $1 \leq i \leq q-1$.
2. For each interval V_i :
   (a) Obtain the corresponding repair $X_i = \{t : t \in I^c \wedge V_i \subseteq t[P]\}$.
   (b) Evaluate function Agg over X_i.
   (c) Compute the probability of X_i: $\int_{V_i} f_\tau(x)dx$.
3. Compute the probability of each value of Agg by summing the probabilities of the repairs corresponding to such a value.

For example, for the aggregation query $average(\texttt{Person}^c, \texttt{Income})$, we extract the end points in attribute P of $\texttt{Person}^c$, which are $\{0, 1, 3, 10\}$ as shown in Figure 3.6. The corresponding intervals $[0, 1]$, $[1, 3]$, and $[3, 10]$ represent the repairs X_1, X_2 and X_3, respectively. We compute the aggregate value corresponding to each repair by evaluating function Agg over the c-tuples in this repair. Finally, we report each aggregate value along with the sum of probabilities of its corresponding repairs. The complexity of the described algorithm is $O(n^2)$ due to evaluating the function Agg over individual repairs (recall that the number of repairs is $O(n)$ and the size of a repair is $O(n)$ as shown in Section 3.2.2). In the remainder of this section, we show how to reduce the complexity to $O(n \log n)$.

We employ a method to incrementally evaluate the aggregate function Agg, which is based on the concept of *partial aggregate states* [1, 62]. This method is based on defining

a *state* for a given subset of the items that summarizes the data of the items. States of disjoint subsets are aggregated to obtain the state of their union. Finally, the state of the set of all items is used for computing the value of the aggregate function.

More specifically, for each aggregate function Agg, three functions have to be defined [62]: `init_state` that initializes the states of singleton and empty sets, `merge_states` that merges states of disjoint sets to obtain the state of their union, and `finalize_state` that obtains the aggregate value corresponding to a state. For example, for aggregate function *average*, a state represents a pair $(sum, count)$. The initialization of the empty set returns the state $(0, 0)$, while initialization of a set with a single item v returns the state $(v, 1)$. The merging of two states $(sum1, count1)$ and $(sum2, count2)$ returns the state $(sum1 + sum2, count1 + count2)$. The finalization function returns the value of $sum/count$. To compute the aggregate value of a repair X, we can compute the state of the c-tuples that belong to X, and then we invoke function `finalize_state` to obtain the aggregate value.

We define a B-tree index, denoted IND, over the parameter space $[\tau^l, \tau^u]$ such that each interval V_i is represented as a leaf node in IND (denoted as V_i as well). Each leaf node V_i represents a distinct possible repair $X_i = \{t : t \in I^c \wedge V_i \subseteq t[P]\}$. We associate each node l in IND with a local state, denoted $l.state$. We construct IND such that the state of tuples in X_i corresponding to V_i results from merging the local state of V_i and the local states of all ancestor nodes of V_i in IND.

Algorithms 2 and 3 outline our procedure to obtain the probability distribution of the aggregate value. Initially, the entire parameter space $[\tau^l, \tau^u]$ is covered by one node in the index, named *root*. The local state of *root* is initialized to the state of the empty set (e.g., $(0, 0)$ in case of the function *average*). For each c-tuple in I^c, the procedure `Update_Index` is invoked. `Update_Index` recursively traverses the index IND starting from the root node. For each node l, if the associated parameter range is completely covered by the interval P, we update the local state of l, otherwise, if l is an internal node, we recursively process its children nodes. If l is a leaf node, we split it into multiple nodes such that one of the new nodes is contained in the interval P (and thus we update its local state accordingly), and the other node(s) are disjoint from P (and thus their local states are not changed). Whenever a node is split (as it becomes full, or due to the condition at line 7

Algorithm 2 `Aggregate`($I^c, expr,$ `init_state, merge_states, finalize_state`)

Require: I^c: An input U-clean relation

Require: $expr$: An expression over attributes of R

Require: `init_state`, `merge_states`, `finalize_state`: Functions for manipulating states of nodes

Define an index IND over the space of the clustering algorithm parameter $[\tau^l, \tau^u]$
Initialize IND to have one node $root$ covering the entire parameter space
$root.state \leftarrow$ `init_state`(ϕ)
Define a set D (initially empty)
Define a state $tuple\_state$
for each $t \in I^c$ **do**
  $tuple\_state \leftarrow$ `init_state`$(\{expr(t)\})$
  `Update_Index`($root$, $t[P]$, $tuple\_state$, `merge_states`)
end for
for each node $l \in IND$, using pre-order traversal **do**
  if $l \neq root$ **then**
    $l.state \leftarrow$ `merge_states`$(l.state, l.parent.state)$
  end if
  if l is a leaf node **then**
    $Agg\_value \leftarrow$ `finalize_state`$(l.state)$
    $Prob \leftarrow \int_l f_\tau(x)dx.$
    Add $(Agg\_value, Prob)$ to D
  end if
end for
Merge pairs in D with the same Agg_value and sum up their $Prob$
return D

Algorithm 3 `Update_Index`$(l, P, tuple\_state,$ `merge_states`$)$

Require: l: a node in an index

Require: P: parameter interval to be updated

Require: *tuple_state*: a new state to be merged within the interval P

Require: `merge_states`: A function to merge multiple states

1: **if** the range of node l is entirely contained in P **then**
2: $l.state \leftarrow$ `merge_states`$(l.state, tuple\_state)$
3: **else if** l is an internal node and l intersects with P **then**
4: **for each** child node l' of l **do**
5: `Update_Index`$(l', P, tuple\_state,$ `merge_states`$)$
6: **end for**
7: **else if** l is an leaf node and the range of l intersects with P **then**
8: Split l into multiple nodes such that only one new leaf node l' is contained in P and the other node(s) are disjoint from P (note: this might trigger splitting ancestor nodes and/or creating a new root)
9: Set the states of all new leaf nodes to the state of the old leaf node l
10: $l'.state \leftarrow$ `merge_states`$(l'.state, tuple\_state)$
11: **end if**

in Algorithm 3), the local states of the new nodes are the same as the original node. If a new root is introduced, its local state is set to `init_state`(ϕ).

Once all c-tuples are consumed (after line 9 in Algorithm 2), we traverse the index IND in pre-order, and we repeatedly merge the local state of each node with the state of its parent to compute the global state of the node. For each leaf node, we use the computed global state to compute the aggregate value of the corresponding repair. We group the obtained aggregate values and we sum up the probabilities of the corresponding repairs to obtain the probability distribution of the possible aggregate values. We prove in this section that our algorithm has a complexity of $O(n \log n)$, where n is the number of c-tuples in I^c.

Figure 3.7 shows an example to illustrate this procedure for the aggregation query $average($`Person`$^c,$ `Income`$)$. We start with a node covering the parameter range $[0, 10]$,

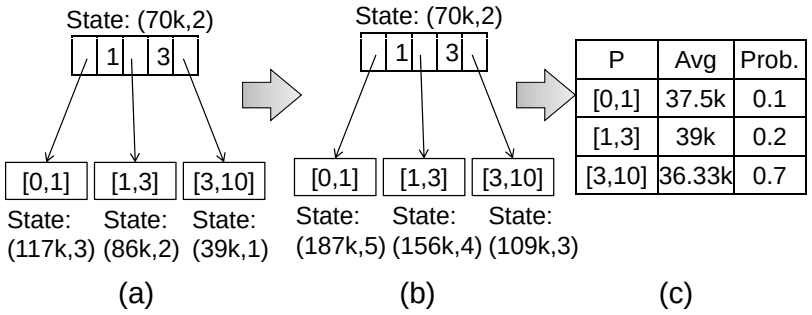

Figure 3.7: An example of an aggregation query (a) index IND after line 9 in Alg. 2 (b) index IND after line 19 in Alg. 2 (c) the probability distribution of the aggregate values

which is initialized to state $(0, 0)$. After reading the first c-tuple `CP1`, we split the node $[0, 10]$ into three leaf nodes $[0, 1]$, $[1, 3]$, and $[3, 10]$ associated with the states (0,0), (31k,1) and (0,0), respectively. At the same time, a new root is introduced with the state $(0, 0)$. When reading the next c-tuple `CP2`, we update the state of the root node to (40k,1). We repeat the same process when reading the remaining c-tuples. The final B-tree is shown in Figure 3.7(a). Then, we merge the state of each node with the states of its ancestors (Figure 3.7(b)). The final probability distribution is derived from the states of leaf nodes (Figure 3.7(c)).

Complexity Analysis

In the following, we prove that our algorithm has a complexity in $O(n \log n)$, where n is the number of c-tuples in I^c.

We divide the procedure of constructing the probability distribution into two steps: (1) constructing IND and updating the nodes states, and (2) obtaining the aggregate values of intervals represented by the leaf nodes of IND and computing the probability distribution.

The first step builds a B-tree by repeatedly inserting parameter ranges of c-tuples in I^c according to Algorithm `Update_Index`. The n intervals of the parameter settings of all c-tuples consist of at most $2n$ distinct end point (e_i's). Thus, the B-tree that contains all distinct intervals V_i's will contain at most $2n - 1$ leaf nodes. As a result, the space complexity of the B-tree is $O(n)$. We prove that insertion of each interval costs $O(\log n)$ as

follows. Let f denote the B-tree fan-out degree, and *height* denote the number of levels in the B-tree, where the root level is 1, and the leaf level is equal to *height*. Assume that we need to insert an interval that spans s contiguous leaf nodes. Out of the s leaf nodes, at most two nodes (i.e., the left-most and right-most nodes) can be split into multiple nodes. For the remaining $s-2$ nodes, we only need to update their states. If $s \geq 2f+2$, then there exists a parent node l_p such that all children of l_p are among the s leaf nodes, and thus we only need to update the local state of l_p instead of the states of the f children nodes. In general, the number of updated nodes is reduced to at most $\lfloor (s-f-2)/f \rfloor + f + 2$ by updating nodes at level $height-1$ instead of their children at level *height*. This observation guarantees that the number of updated or split nodes at level *height* is less than or equal to $f+2$. By continuously applying the same observation at higher levels, we conclude that the maximum number of updated nodes at each level is $f+2$. Therefore, the total number of scanned nodes is less than or equal to $height \cdot (f+2)$ nodes, which is in $O(\log n)$.

The second step involves traversal of the B-tree, which has linear complexity in n. Sorting and grouping pairs of aggregate values and their probabilities are done in $O(n \log n)$. Hence, we conclude that the complexity of finding the probability distribution of the aggregate values is $O(n \log n)$.

Aggregate Queries Over SPJ Results

U-clean relations resulting from SPJ queries involve a number of parameters equal to the number of joined base U-clean relations, denoted by d. The attribute P of each c-tuple is represented as a DNF over single parameter settings where each clause is a conjunction of d parameter settings each of which referring to one of the d parameters. Therefore, we view each clause in attribute P as a hyper-rectangle in a d dimensional space. Consequently, attribute P is viewed as a union of multiple d-dimensional hyper-rectangles.

We extend our technique to answer aggregate queries by replacing the B-tree index with a multidimensional index, namely UB-tree [12]. Also, Algorithm `Update_Index` is modified such that its argument P is a union of multiple hyper-rectangles. The conditions at line 1,3 and 7 are changed to be tested against *any* hyper-rectangle in P. Splitting of a leaf node in line 8 must be performed such that each intersecting hyper-rectangle in P has a new leaf node with the exact range of P.

The complexity of our technique in this case is polynomial in the number of distinct hyper-rectangles appearing in parameter settings of c-tuples (denoted g), and exponential in the number of parameters d. The reason is that the number of leaf nodes in IND (the number of possible repairs) is at most $(2g-1)^d$, as we show next.

In general, parameter settings of a c-tuple involve d parameters and are represented as a union of multiple hyper-rectangles, each of which is d dimensional. We divide the parameter space into a number of disjoint hyper-rectangles as follows. For each parameter $\tau_i, 1 \leq i \leq d$, we extract the end points of all hyper-rectangles in I^c with respect to τ_i. The resulting points divide the space of τ_i into at most $2g-1$ intervals, and thus the space of all parameters is partitioned into at most $(2g-1)^d$ disjoint hyper-rectangles, corresponding to at most $(2g-1)^d$ possible repairs. Consequently, the number of leaf nodes of the index IND cannot exceed $(2g-1)^d$, and thus the complexity of our algorithm is polynomial in g and exponential in d.

3.4 Implementation in RDBMS

In this section, we show how to implement U-clean relations and query processing inside relational database systems. We also propose new queries that reason about the uncertainty of repairing.

3.4.1 Implementing U-clean Relations

We implement attributes C and P in a relational database as abstract data types (ADTs). Attribute C is encoded as a set of (ORed) clauses, each of which is a set of (ANDed) clusters. Attribute P of a U-clean relation I^c is encoded as an array of hyper-rectangles in the d-dimensional space, where d is the number of parameters of the used clustering algorithms. Each hyper-rectangle is represented as d one-dimensional intervals.

Executing SPJ queries requires manipulation of attributes C and P according to the discussion in Section 3.3.1. The selection and projection (under bag semantics) operators do not alter the values of C and P, and hence no modifications are necessary to these

operators in relational DBMSs. On the other hand, the join operator modifies attributes C and P to be the conjunctions of the joined c-tuples attributes. C and P of the results are computed through functions $ConjC(C_1, C_2)$ and $ConjP(P_1, P_2)$, where C_1 and C_2 are tuple clusters, and P_1 and P_2 are parameter settings of clustering algorithm(s). We implement the functions $ConjC$ and $ConjP$ such that they return the conjunction of their inputs in DNF.

In our implementation, we do not provide native support to projection with duplicate elimination (i.e., using the Distinct keyword). However, we realize projection with duplicate elimination through group-by queries. We implement two functions $DisjC(C_1, \ldots, C_n)$ and $DisjP(P_1, \ldots, P_n)$ to obtain the disjunction of clusters $C_1, \ldots, C_n$ and parameter settings $P_1, \ldots, P_n$, respectively. Performing projection with duplicate elimination of a U-clean relation `UR` on a set of attributes `A1,...,Ak` is equivalent to the following SQL query:

```
SELECT A1,...,Ak, DisjC(C), DisjP(P)
FROM UR
GROUP BY A1,...,Ak
```

This query effectively projects `UR` on attributes `A1,...,Ak` and computes the disjunctions of attributes C and P of the duplicate c-tuples.

In the following, we give a list of operations that reason about the possible repairs encoded by a U-clean relation to allow new probabilistic query types that are described in Section 3.4.2.

- $Contains(P, x)$ returns `True` iff the parameter settings P contains a given parameter setting x.
- $ContainsBaseTuples(C, S)$ returns `True` iff a set of base tuples identifiers S is contained in a cluster C.
- $Prob(P, f_{\tau_1}, \ldots, f_{\tau_d})$ computes the probability that a c-tuple with parameter settings P belongs to a random repair. $f_{\tau_1}, \ldots, f_{\tau_d}$ are the probability distribution functions of the clustering algorithms parameters $\tau_1, \ldots, \tau_d$ that appear in P.

- $MostProbParam(\texttt{UR}, f_{\tau_1}, \ldots, f_{\tau_d})$ computes the parameter setting of the most probable repair of a U-clean relation UR, given the probability distribution functions of parameters $f_{\tau_1}, \ldots, f_{\tau_d}$.

We describe how to efficiently implement the functions $Prob$ and $MostProbParam$ as follows.

Implementing Function $Prob$

$Prob$ determines the membership probability of a c-tuple, given its parameter settings P, denoted as $\Pr(P)$. For a base U-clean relation that involve a single clustering algorithm parameter τ with probability distribution function f_τ, this probability is equal to $\int_P f_\tau(x)dx$. For U-clean relations resulting from SPJ queries, attribute P involves d parameters and is represented as a union of several hyper-rectangles, each of which is d dimensional. Let g denotes the number of distinct hyper-rectangles that appear in attribute P for all c-tuples. We first divide the parameters space into a number of disjoint hyper-rectangles, denoted $\{L_1, L_2, \ldots\}$, as follows. For each parameter $\tau_i, 1 \leq i \leq d$, we extract the distinct end points with respect to τ_i of the hyper-rectangles that appear in I^c. The resulting points divide the space of τ_i into at most $2g-1$ intervals, and thus the space of all parameters is partitioned into at most $(2g-1)^d$ disjoint hyper-rectangles $\{L_1, L_2, \ldots\}$. The probability of L_j is defined as follows.

$$\Pr(L_j) = \prod_{i=1}^{d} \int_{L_j.\tau_i^l}^{L_j.\tau_i^u} f_{\tau_i}(x)dx \tag{3.2}$$

where $L_j.\tau_i^l$ and $L_j.\tau_i^u$ indicate the lower and upper values of parameter τ_i in cell L_j, respectively. Clearly, for each hyper-rectangle H that appear in attribute P of a c-tuple, a hyper-rectangle L_j can only be either contained in H or disjoint from H. Additionally, H is completely covered by one or more hyper-rectangles in $\{L_1, L_2, \ldots\}$. Thus, we compute $\Pr(P)$ as follows.

$$\Pr(P) = \sum_j con(L_j, P) \Pr(L_j) \tag{3.3}$$

where $con(L_j, P)$ is an indicator function that returns 1 if L_j is contained in any hyper-rectangle in P, and 0 otherwise.

Implementing Function $MostProbParam$

For base U-clean relations, determining the most probable repair can be done efficiently by scanning c-tuples in I^c, and extracting all end points of their parameter settings. Distinct end points split the parameter space $[\tau^l, \tau^u]$ into multiple intervals $V_1, \ldots, V_m$ corresponding to the possible repairs. For example, in Figure 3.6, the possible repairs are X_1, X_2, and X_3 corresponding to the intervals $[0, 1]$, $[1, 3]$, and $[3, 10]$, respectively. The probability of each repair is computed based on its corresponding parameter settings. Function $MostProbParam$ returns the interval of the repair with the highest probability (e.g., $[3, 10]$ in Figure 3.6). The overall complexity of the function $MostProbParam$ is $O(n \log n)$ (mainly, due to sorting the end points of parameter settings).

For U-clean relations resulting from SPJ queries, we use the following technique to compute $MostProbParam$. We denote by g the number of distinct hyper-rectangles in parameter setting of all c-tuples. We partition the parameters space into at most $(2g-1)^d$ disjoint hyper-rectangle using the end points of all hyper-rectangles appearing in c-tuples (in the same way described for implementing $Prob$). We construct a set of parameter settings, denoted $\mathcal{Z}$ such that each item in $\mathcal{Z}$ is a subset of hyper-rectangles in $\{L_1, L_2, \ldots\}$ corresponding to a unique repair. Initially, $\mathcal{Z}$ contains one set Z_0, which is the set of all hyper-rectangles $\{L_1, L_2, \ldots\}$. For each c-tuple t in I^c, we split each set Z_i in $\mathcal{Z}$ into two sets Z_{i1}, Z_{i2} such that Z_{i1} (respectively, Z_{i2}) corresponds to repairs containing (respectively, not containing) t. That is, $Z_{i1} = \{L_j : L_j \in Z_i \wedge L_j \subseteq t[P]\}$ and $Z_{i2} = \{L_j : L_j \in Z_i \wedge L_j \nsubseteq t[P]\}$. After scanning all c-tuples, we compute the probability of each set (i.e., repair) Z_i, which is equal to the sum of probabilities of hyper-rectangles in Z_i. Once the highest probability is identified, we return the corresponding set Z_i.

For example, Figure 3.8(a) shows a U-clean relation resulting from a join query. The set $\mathcal{Z}$ initially contains one set Z_0 containing the four cells depicted in Figure 3.8(b). Scanning the first c-tuples does not cause splitting of Z_0 as all cells are contained in the parameter settings $\tau_1 : [0, 10], \tau_2 : [0, 5]$. Scanning the seconds c-tuple results in splitting

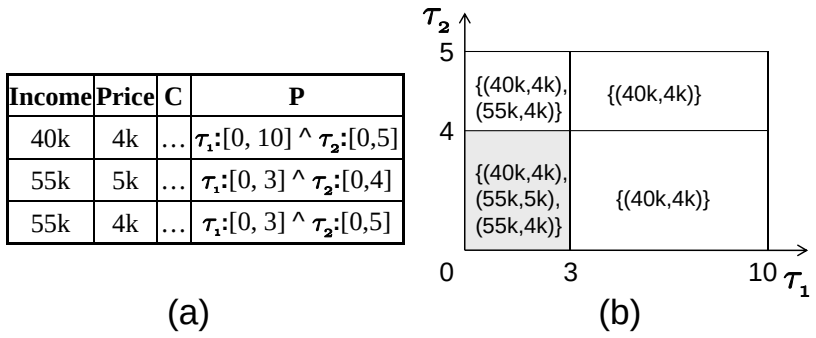

Figure 3.8: (a) Results of a join query (b) the corresponding possible clean instances

Z_0 into two sets such that the first set contains the shaded cell, and the second set contains the unshaded cells. After scanning the third c-tuple, the set $\mathcal{Z}$ contains three sets. The probability of each set is then computed and the most probable one (which covers τ_1 : $[3, 10], \tau_2 : [0, 5]$) is returned.

3.4.2 Other Query Types

In this section, we describe multiple meta-queries that are defined over our uncertainty model. Specifically, the queries we describe in this section explicitly use attributes P and C to reason about the possible repairs modeled by U-clean relations.

Extracting Possible Clean Instances

Any clean instance encoded in a U-clean relation can be constructed efficiently given the required parameters values of the clustering algorithm(s). For a base U-clean instance I^c, the clean instance at parameter value x is equal to $\{t[A_1, \ldots, A_m] : t \in I^c \wedge x \in t[P]\}$.

Extracting the clean instance corresponding to a given parameter value x can be performed through a selection query with the predicate $Contains(P, x)$. For example, assume that we need to extract the clean instance of the U-clean relation $\texttt{Prices}^c$ in Figure 3.5(b) corresponding to the parameter setting $\tau_2 = 4.1$. This clean instance is computed using the following SQL query:

SELECT `Price`
FROM `Prices`^{c}
WHERE $Contains(P, 4.1)$

which results in the tuples $4k$, $6k$, and $8k$.

It is possible to speed up extraction of clean instances by indexing the c-tuples based on their parameter settings using an R-Tree index [49]. More specifically, we create a d-dimensional R-tree index over the space of possible settings of parameters $\tau_1, \ldots, \tau_d$. The parameter settings of a c-tuple t is generally a union of d-dimensional hyper-rectangles. For each c-tuple $t \in I^c$, we insert its hyper-rectangles into the R-tree, and label them with the identifier of t. To extract the repair at $\tau_1 = x_1, \ldots, \tau_d = x_d$, we search the R-tree for hyper-rectangles that contain the point $(x_1, \ldots, x_d)$ and report the associated c-tuples.

Obtaining the Most Probable Clean Instance

An intuitive query is to extract the clean instance with the highest probability. It is possible to answer this query with the help of two functions, namely $Contains$ and $MostProbParam$, through a selection SQL query. For example, assume that a user requests the most probable repair from Relation `Person`^{c} which is shown in Figure 3.3. This query can be answered using the following SQL query:

SELECT `ID, Name, ZIP, Income, BirthDate`
FROM `Person`^{c}
WHERE $Contains(P, MostProbParam(\texttt{Person}^c, U(0, 10)))$

Note that $MostProbParam$ is evaluated only once during the entire query and thus the cost incurred by this function is only paid once.

Finding α-certain c-tuples

We consider a query that finds c-tuples that exhibit a degree of membership certainty above a given threshold α. We call this type of queries an α-certain query. This query type can

be answered by issuing a selection query with the predicate $Prob(P, f_{\tau_1}, \ldots, f_{\tau_d}) \geq \alpha$. For example, consider a 0.5-certain query over the relation in Figure 3.5(c). This query is answered using the following SQL query:

SELECT `Income`, `Price`, $ConjC$(`PC`.C,`VC`.C) AS `C`, $ConjP$(`PC`.P,`VC`.P) AS `P`
FROM `Person`^{c} `PC` , `Vehicle`^{c} `VC`
WHERE `Income/10` >= `Price`
AND $Prob(P, U(0,10), U(0,10)) >= 0.5$

This SQL query reports only the first c-tuple in Figure 3.5(c), which has a membership probability of 0.5.

Note that α-certain queries can be considered a generalization of consistent query answers [9]. That is, setting α to 1 retrieves the c-tuples that appear in every possible clean instance.

Probability of Clustering Tuples Together

We show how to compute the probability that multiple tuples in an unclean instance I belong to the same cluster (i.e., declared as duplicates). For example, consider a query requesting the probability that two tuples `P1` and `P2` from the instance `Person` are clustered together according to the repairs encoded in U-clean relation `Person`^{c} (Figure 3.3). The probability of clustering a set of tuples is equal to the sum of probabilities of repairs in which this set of tuples is clustered together. To compute this probability, we first select all c-tuples whose attribute C contains all tuples in question (e.g., `P1` an `P2`). Values of attribute C of the selected c-tuples are overlapping since they all contain the query tuples. Consequently, the selected c-tuples are exclusive (i.e., cannot appear in the same repair) and the clustering probability can be obtained by summing probabilities of the selected c-tuples:

$$\Pr(\text{clustering } t_1, \ldots, t_k) = \sum_{t \in I^c : \{t_1, \ldots, t_k\} \subseteq t[C]} \Pr(t[P]) \tag{3.4}$$

For example, the probability of clustering tuples `P1` and `P2` is obtained using the following query:

SELECT $Sum(Prob(P, U(0, 10)))$
FROM `Person`^{c}
WHERE $ContainsBaseTuples(C,$ `'P1,P2'`$)$

which returns the probability 0.9.

It is worth mentioning that the way we obtain clustering probabilities is substantially different from other approaches that computes the matching probabilities of tuples pairs. For example, in [38], Fellegi and Sunter derive the probability that two tuples are duplicates (i.e., match each other) based on the similarity between their attributes. Unlike our approach, in [38], probabilities of matching (i.e., clustering) tuple pairs are computed in isolation of other pairs, which may lead to inconsistencies. For example, the pair (t_1, t_2) may have a matching probability of 0.9, and the pair (t_2, t_3) has a matching probability of 0.8, while the matching probability of the pair (t_1, t_3) is equal to 0. Our approach avoids such inconsistencies by deriving pair-wise clustering probabilities based on the uncertain output of a clustering algorithm, which by definition resolves such inconsistencies. Moreover, our approach can obtain the matching probability of more than two tuples.

3.5 Probabilistic Merging of Clusters

In this section, we show how to extend our model to allow capturing uncertainty in merging the clustered tuples.

In [6], possible outcomes of merging a cluster are assumed to be its member tuples. It is also possible to define multiple aggregate functions over the cluster members, each of which provide one possible merging output. For example, to merge numerical attributes, we might include the median and the mean values as possible outcomes.

For example, Figure 3.9 shows a set of repairs for an unclean relation. The corresponding parameter settings of the used clustering algorithm are shown above each repair. The

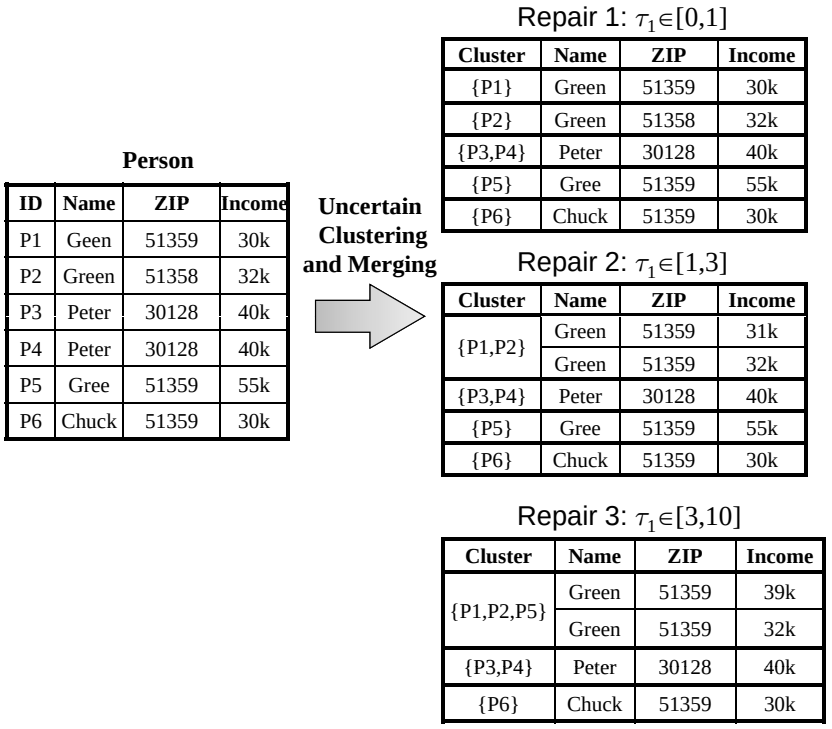

Figure 3.9: An example of possible repairs when both clustering and merging steps are uncertain

uncertain merging procedures are defined to report the longest name, the ZIP code of the majority (with arbitrary tie breaking), and both the mean and the median incomes.

We assume that the probability distribution of the outcomes of a merging procedure is given (e.g., using the method introduced in [6], or using user-specified confidence values associated with aggregation functions). Moreover, we assume that the merging outcome is independent from the parameters of the clustering algorithms, and that the merging outcomes of different clusters are independent. The outcome of merging each cluster C_i can be viewed as a random variable M_i. We call M_i the *merging random variable* of C_i. The possible outcomes of M_i are identifiers of the possible tuples resulting from the merging process.

We extend our model to allow encoding of possible merging outcomes as follows. We

Person^c

ID	Name	ZIP	Income	C	P
CP11	Green	51359	31k	{P1,P2}	τ_1:[1,3] ^ M_1:1
CP12	Green	51359	32k	{P1,P2}	τ_1:[1,3] ^ M_1:2
CP2	Peter	30128	40k	{P3,P4}	τ_1:[0,10]
CP3	Gree	51359	55k	{P5}	τ_1:[0,3]
CP4	Chuck	51359	30k	{P6}	τ_1:[0,10]
CP51	Green	51359	39k	{P1,P2,P5}	τ_1:[3,10] ^ M_2:1
CP52	Green	51359	32k	{P1,P2,P5}	τ_1:[3,10] ^ M_2:2
CP6	Green	51359	30k	{P1}	τ_1:[0,1]
CP7	Green	51358	32k	{P2}	τ_1:[0,1]

(a)

$\mathcal{T}_1 \sim$ U[0,10]

$Pr(M_1=1) = Pr(M_1=2) = 0.5$

$Pr(M_2=1) = Pr(M_2=2) = 0.5$

(b)

Figure 3.10: (a) An example U-Clean relation in the presence of uncertain clustering and merging. (b) the probability distributions of the used random variables

represent each possible merging outcome as a separate c-tuple, whose attribute C is equal to the set of merged tuples. The attribute P of each c-tuple is a conjunction of: (1) the parameter settings of the clustering algorithm leading to generating C, and (2) the outcome of the merging random variable associated with the cluster C that corresponds to this c-tuple. Note that in case of having a single outcome from merging a cluster, we do not need to introduce a new merging random variable, and thus, the attribute P of the corresponding c-tuple consists only of the parameter settings of the clustering algorithm.

In Figure 3.10(a), we show a U-clean relation `Person`^{c} that encodes repairs of the unclean relation `Person` that are shown in Figure 3.9. The used clustering algorithm has one parameter τ_1 that follows a uniform distribution over the interval $[0, 10]$. Two random variables M_1 and M_2 are introduced to encode different merging outcomes for the clusters $\{$`P1`, `P2`$\}$ and $\{$`P1`, `P2`, `P5`$\}$, respectively. Outcomes of M_1 and M_2 are assumed to be equiprobable. Note that the attribute C does not uniquely identifies a c-tuple in the presence of multiple merging outcomes. For example, the c-tuples `CP11` and `CP12` represent the same cluster $\{$`P1`, `P2`$\}$.

Membership probabilities of c-tuples can be derived using the attribute P and the

SELECT Income, DisjC(C), DisjP(P)
FROM Person^c
Group By Income

Income	C	P
31k	{P1,P2}	τ_1:[1,3] ^ M_1:1
32k	{P1,P2} ∨ {P1,P2,P5} ∨ {P2}	(τ_1:[1,3] ^ M_1:2) ∨ (τ_1:[3,10] ^ M_2:2) ∨ (τ_1:[0,1])
40k	{P3,P4}	τ_1:[0,10]
55k	{P5}	τ_1:[0,3]
30k	{P6} ∨ {P1}	τ_1:[0,10] ∨ τ_1:[0,1]
39k	{P1,P2,P5}	τ_1:[3,10] ^ M_2:1

Figure 3.11: An example of a query over `Person`^{c}

probability distributions of the used random variables (e.g., τ_1, M_1, and M_2 as shown in Figure 3.10(b)). For example, the membership probability of `CP51` is equal to $\Pr(\tau_1 \in [3, 10] \wedge M_2 = 1) = 0.7 \times 0.5 = 0.35$.

SPJ queries, α-certain query, repair extraction query, and querying probability of clustering tuples are performed in the same way as defined in Section 3.4. For example, Figure 3.11 depicts a query over `Person`^{c} (Figure 3.10) that performs projection on the attribute `Income` under set semantics.

Note that in repair extraction queries, if only the parameters of clustering algorithms are specified in query, all merging outcomes for the extracted repair will be reported. For example extracting the repair corresponding to the parameter setting $\tau_1 = 2.5$ from `Person`^{c} (Figure 3.10(a)) returns the c-tuples `CP11`, `CP12`, `CP2`, `CP3`, and `CP4`.

Unfortunately, the proposed algorithms for aggregation queries and for obtaining the most probable repair cannot efficiently be executed due to the possibility of having a large number of variables in attribute P corresponding to the merging random variables (recall that such algorithms have exponential complexity in the number of variables). Approximate query answering is to be investigated in our future work to handle such queries more efficiently.

3.6 Experimental Evaluation

In our experiments, we show that our probabilistic cleaning approach has negligible time and space overheads compared to the existing data cleaning approaches, which warrants adopting our approach in realistic settings. We also show that queries over U-clean relations can be answered efficiently using our algorithms.

3.6.1 Setup

All experiments were conducted on a SunFire X4100 server with Dual Core 2.2GHz processor, and 8GB of RAM. We implemented all functions in Section 3.4.1 as user defined functions (UDFs) in PostgreSQL DBMS [1]. We used the synthetic data generator that is provided in the Febrl project [26], which produces one relation, named `Person`, that contains persons data (e.g., given_name,surname, address, phone, age). Data sets generated using Febrl exhibit the content and statistical properties of real-world data sets [25], including distributions of the attributes values, error types, and error positions within attribute values. The parameters of the experiments are as follows.

- The number of tuples in the input unclean relation (the default is 100,000).
- The percentage of duplicate tuples in the input relation (the default is 10%).
- The width of the parameter range used in the duplicate detection algorithms (the default is 2, which is 10% of width of the broadest possible range according to the distribution of the pair-wise distance values). We assume that the parameters have uniform distributions. For deterministic duplicate detection, we use the mean value.

Our implementation of duplicate elimination algorithms is based on the single-linkage clustering (S.L.) [54], and the NN-based clustering algorithm using the function *max* for aggregating neighborhood growths [20]. Deduplication algorithms are executed in memory. All queries, except aggregate queries, are executed through SQL statements submitted to PostgreSQL. Aggregate queries are processed by an external procedure that implements

the algorithms described in Section 3.3.2. All queries are performed over a single U-clean relation, named `Person`^{c}, which is generated by uncertain deduplication of `Person`. Each query is executed five times and the average running time is recorded. We report the following metrics in our experiments:

- The running time of deterministic and uncertain clustering algorithms. The reported times do not include building the similarity graph, which is performed by Febrl [26].
- The sizes of the produced relations.
- The response times of an aggregate query using the *count* function, and the probabilistic queries in Section 3.4.2 against Relation `Person`^{c} constructed by the S.L. algorithm. The threshold α is set to 0.5 in α-certain queries. In clustering probability queries, we use two random tuples as query arguments. We omit queries for extracting clean instances as they have almost identical response times to obtaining the most probable clean instance.
- The relative overhead of maintaining attributes C and P in U-clean relations during selection, projection, and join queries as defined in Section 3.3.1 (we refer to such queries as uncertain queries). We compare the uncertain queries to regular SPJ queries that do not use, compute, or return attributes C and P (we call them base queries). The base selection query returns attribute `Age` of all tuples, while the uncertain selection query returns attributes `Age`, C and P. The base join query performs a self-join over `Person`^{c} to join c-tuples with the same `Surname` and different `ID`. The uncertain join query additionally computes attributes C and P of the results. The base projection query (with duplicate elimination) projects relation `Person`^{c} on attribute `surname` while ignoring attributes C and P. The uncertain projection computes attributes C and P of the results as described in Section 3.4.1.

3.6.2 Results

We first summarize the results of our experiments as follows. We observe that the overhead in execution time of the uncertain deduplication, compared to the deterministic dedupli-

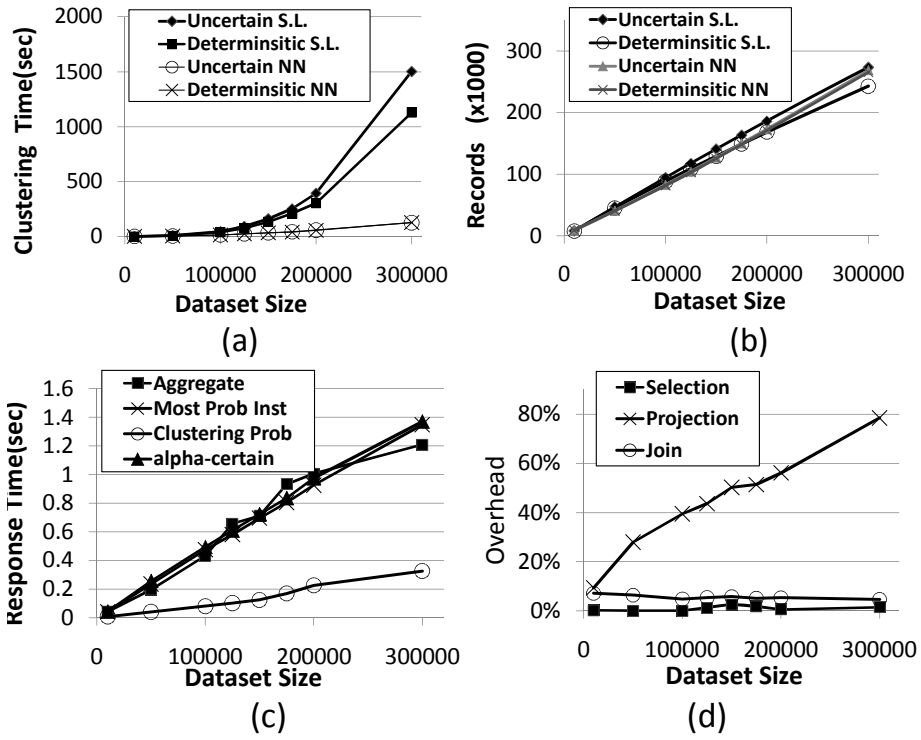

Figure 3.12: The effect of the data set size on (a) clustering running time (b) output size (c) uncertain queries running times (d) the running time overhead for SPJ queries

cation, is less than 30% in case of the S.L. algorithm, and less than 5% in case of the NN-based clustering algorithm for input data size of 300000 tuples. The average overhead in space requirements is equal to 8.35%, while the maximum overhead is equal to 33%. We also note that extracting a clean instance takes less than 1.5 seconds in all cases which indicates that our approach is more efficient than restarting the deduplication algorithm whenever a new parameter setting is requested.

In the following, we show more details about the effect of changing the experiments parameters.

The Effect of Data set Size (Figure 3.12): The average computational overhead of the uncertain S.L. algorithm is 20% compared to the deterministic version. The running times of both versions of the NN-based algorithm are almost identical. Output sizes (Fig-

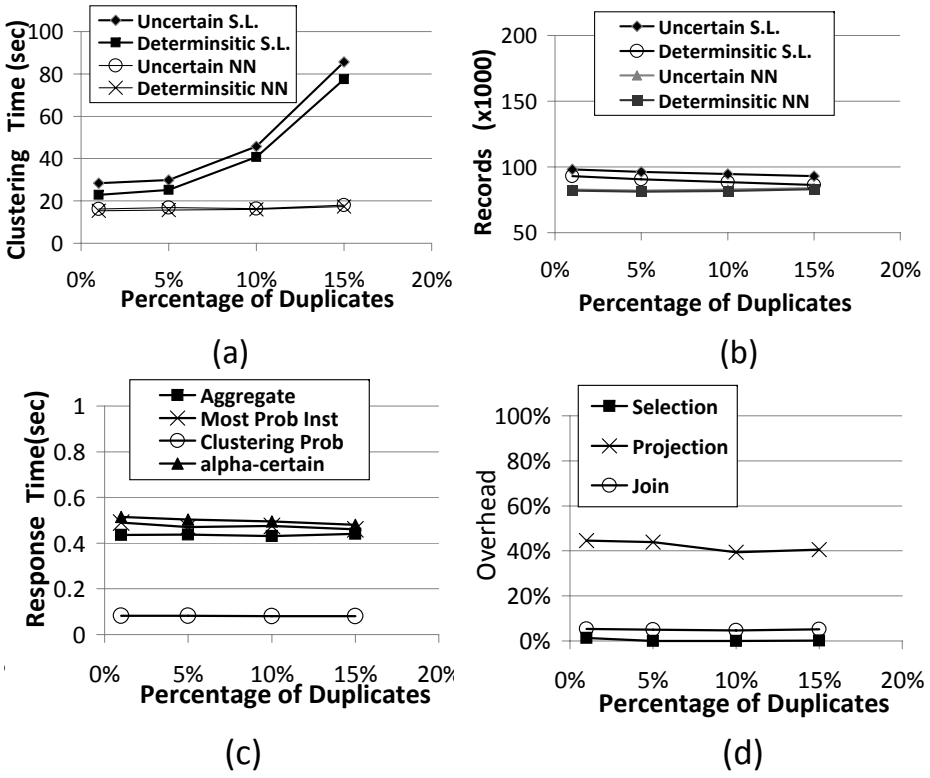

Figure 3.13: The effect of duplicate percentage on (a) clustering running time (b) output size (c) uncertain queries running times (d) the running time overhead for SPJ queries

ure 3.12(b)) are almost identical for uncertain and deterministic deduplication algorithms. The responses times of queries (Figure 3.12(c)) exhibit linear (or near-linear) increase with respect to the data set size.

The overhead in running time of SPJ queries varies among query types (Figure 3.12(d)). Selection queries have the lowest overhead (almost zero) because the only extra operation is converting data types of attributes C and P into string format in output. Join queries have almost fixed relative overhead (about 5% in all cases) due to the constant time consumed in computing attributes C and P per tuple in the join results. The projection query suffers from an overhead that increases linearly with the relation size due to evaluating the aggregate functions $DisjC$ and $DisjP$.

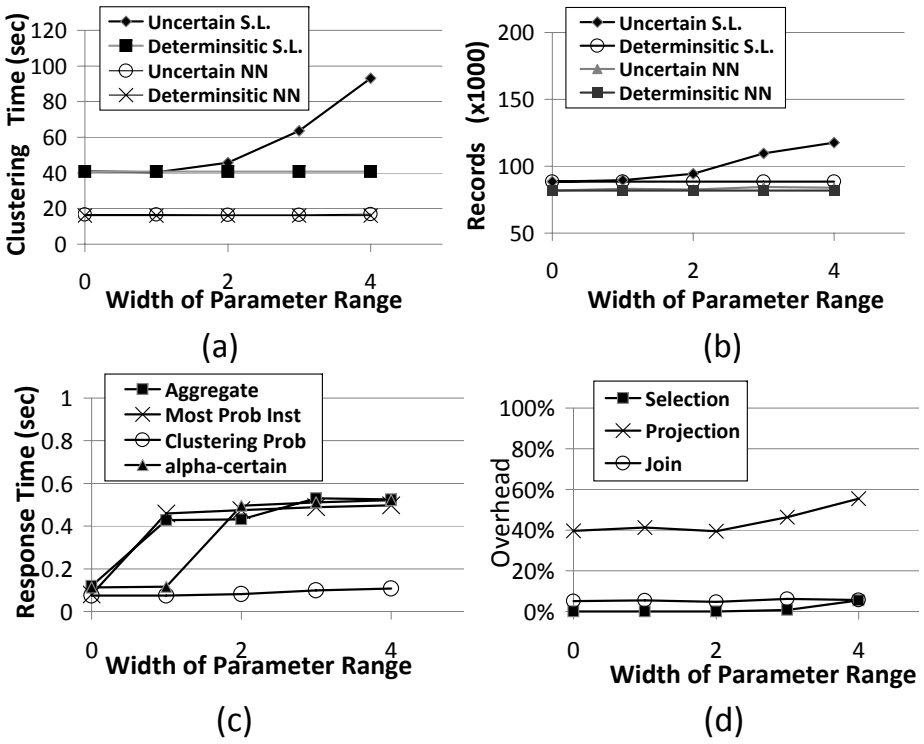

Figure 3.14: The effect of the parameter range on (a) clustering running time (b) output size (c) uncertain queries running times (d) the running time overhead for SPJ queries

The Effect of Percentage of Duplicates (Figure 3.13): The overhead of executing uncertain S.L. algorithm remains low (30% at most) as the percentage of duplicates rises (Figure 3.13(a)). The uncertain NN-based algorithm has almost no overhead regardless of the amount of duplicates. The output size slightly declines at higher percentages of duplicates due to the increasing number of merged tuples (Figure 3.13(b)). Produced clusters mainly consist of singletons. Hence, query response times are hardly affected by the increased percentage of duplicates (Figures 3.13(c) and 3.13(d)).

The Effect of the Width of Parameter Range (Figure 3.14): The clustering running times and the output sizes of the deterministic clustering algorithms do not change because the parameter value remains fixed at the mean parameter value. In contrast, the running times of the uncertain S.L. algorithm increase due to having greater upper bounds of

parameters, which results in testing and clustering additional tuples. We observe that the running time of the uncertain NN-based clustering algorithm does not increase significantly. This is due to the highly selective clustering criteria imposed by NN-based algorithm (i.e., compactness of the cluster).

The output size for the uncertain S.L. algorithm also grows as more candidate clusters are emitted to the output U-clean relation (Figure 3.14(b)). Consequently, queries imposed against the output of the S.L. clustering algorithm suffer from increased response times (Figures 3.14(c) and 3.14(d)).

Chapter 4

Sampling Repairs of FD Violations

In this chapter, we present our approach for probabilistic repairing of functional dependency violations [15]. In Section 4.1, we present previous definitions of possible repairs as well as a new definition. In Section 4.2, we introduce our approach to sample from the new space of possible repairs. We show how to improve the efficiency of the sampling algorithm through partitioning data into separately-repairable blocks in Section 4.3. In Section 4.4, we present an experimental study of our sampling approach. In Section 4.5, we provide a discussion about the difficulties in randomizing previous data cleaning algorithms.

4.1 Spaces of Possible Repairs

We first give a few notations that are used in this chapter. We denote by R a relation schema consisting of m attributes, denoted $(A_1, \ldots, A_m)$. We denote by I an instance of R consisting of n tuples, each of which has a unique identifier. We denote by $TIDs(I)$ the identifiers of tuples in I. We refer to an attribute $A \in R$ of a tuple $t \in I$ as a *cell*, denoted $t[A]$. We denote by $CIDs(I) = \{t[A] : t \in TIDs(I), A \in R\}$ the set of all cell identifiers in I. We denote by $I(t[A])$ the value of a cell $t[A]$ in an instance I. For an FD $X \rightarrow A$, where $X \subseteq R$ and $A \in R$, we refer to X as the left-hand-side (LHS) attributes, and we refer to A as the right-hard-side (RHS) attribute.

A repair of an inconsistent instance I with respect to a set of FDs Σ is another instance I' that satisfies Σ. As explained in Section 2.2.2, we only consider repairs obtained by modifying tuple attributes (i.e., cells) of I. An FD repair is formally defined as follows.

Definition 3. ***FD Repair***. *Given a set of FDs Σ defined over a relation R, and an instance I of R that does not necessarily satisfy Σ, a repair of I is another instance I' of R such that $I' \models \Sigma$ and $TIDs(I) = TIDs(I')$.*

That is, a repair I' of an inconsistent instance I is an instance that satisfies Σ and has the same set of tuple identifiers in I. The attribute values of tuples in I and I' can be different. The sets of cell identifiers in both I' and I are equal (i.e., $CIDs(I) = CIDs(I')$). We denote by $Repairs(I)$ the set of all possible repairs of an instance I. We denote by $\Delta(I, I')$ identifiers of the cells that have different values in I and I', that is, $\Delta(I, I') = \{C \in CIDs(I) : I(C) \neq I'(C)\}$. For example, in Figure 4.1, $\Delta(I, I_2) = \{t_2[B], t_3[B]\}$. Also, we denote by $\lambda(I, I')$ the set of changes made in I in order to obtain I', where each change is represented as a pair of a cell and the new value assigned to this cell in I'. Formally, $\lambda(I, I') = \{(C, v) : I(C) \neq I'(C) \wedge v = I'(C)\}$. For example, in Figure 4.1, $\lambda(I, I_4) = \{(t_1[A], 7), (t_1[B], 3)\}$.

It is useful to filter out repairs that are less likely to represent the actual clean database. A widely used criterion is the *minimality of changes* (e.g., [19, 23, 24, 47, 57]). The main hypothesis is that the largest part of the data is clean and thus we need only to change a small number of the database cells in order to bring the database instance into accordance with Σ. In the following, we describe two repair definitions that have different degrees of trust is such a hypothesis.

Definition 4. ***Cardinality-Minimal Repair*** *[19, 57]: A repair I' of I is cardinality-minimal iff there is no repair I'' of I such that $|\Delta(I, I'')| < |\Delta(I, I')|$.*

That is, a repair I' of I is cardinality-minimal iff the number of changed cells in I' is the minimum across all repairs of I. This definition has the strongest confidence in the described hypothesis.

Definition 5. ***Set-Minimal Repair*** *[9, 60]: A repair I' of I is set-minimal iff there is no repair I'' of I such that $\lambda(I, I'') \subset \lambda(I, I')$.*

That is, a repair I' of I is set-minimal iff no strict subset of the changed cells in I' can be reverted to their original values in I without violating Σ. This definition has the least confidence in the described hypothesis. Note that we use the symbol $\subset$ to indicate strict (proper) subset (also written as $\subsetneq$ in other publications).

Previous approaches that generate a single repair of an unclean relation instance typically find a nearly-optimal cardinality-minimal repair (finding a cardinality-minimal repair is NP-hard [19, 23, 57]). In contrast, prior work on consistent query answering considers set-minimal repairs [24, 47]. Repairs that are not set-minimal are believed to be unacceptable repairs since they involve unnecessary changes [9, 24, 60].

We introduce a novel space of repairs, called cardinality-set-minimal repairs. The goal of such a space is striking a balance between the "fewest changes" metric of cardinality-minimality and the "necessary changes" criterion of set-minimality.

Definition 6. ***Cardinality-Set-Minimal Repair*** *A repair I' of I is cardinality-set-minimal iff there is no repair I'' of I such that $\Delta(I, I'') \subset \Delta(I, I')$.*

That is, a repair I' of I is cardinality-set-minimal iff no subset $\mathcal{C}$ of the changed cells in I' can be reverted to their original values in I without violating Σ, even if we allow modifying the cells in $\Delta(I, I') \setminus \mathcal{C}$ to other values.

In Figure 4.1, we show the various types of repairs of an instance I, with the changed cells greyed out. Repairs I_1 is cardinality-minimal because no other repair has fewer changed cells. Repair I_1 is also cardinality-set-minimal and set-minimal. Repairs I_2 and I_3 are set-minimal because reverting any subset of the changed cells to the values in I will violate $A \rightarrow B$. On the other hand, I_3 is not cardinality-set-minimal (or cardinality-minimal) because reverting $t_2[B]$ and $t_3[B]$ back to 3 and changing $t_1[B]$ to 3 instead of 5 gives a repair of I, which is the same as I_1. Repair I_4 is not set-minimal because I_4 still satisfies $A \rightarrow B$ after reverting $t_1[A]$ to 1. The relationship among the various definitions of minimal repairs is depicted in Figure 4.2 and described in the following lemma.

Lemma 1. *The set of cardinality-minimal repairs is a subset of cardinality-set-minimal repairs. Moreover, the set of cardinality-set-minimal repairs is a subset of set-minimal repairs.*

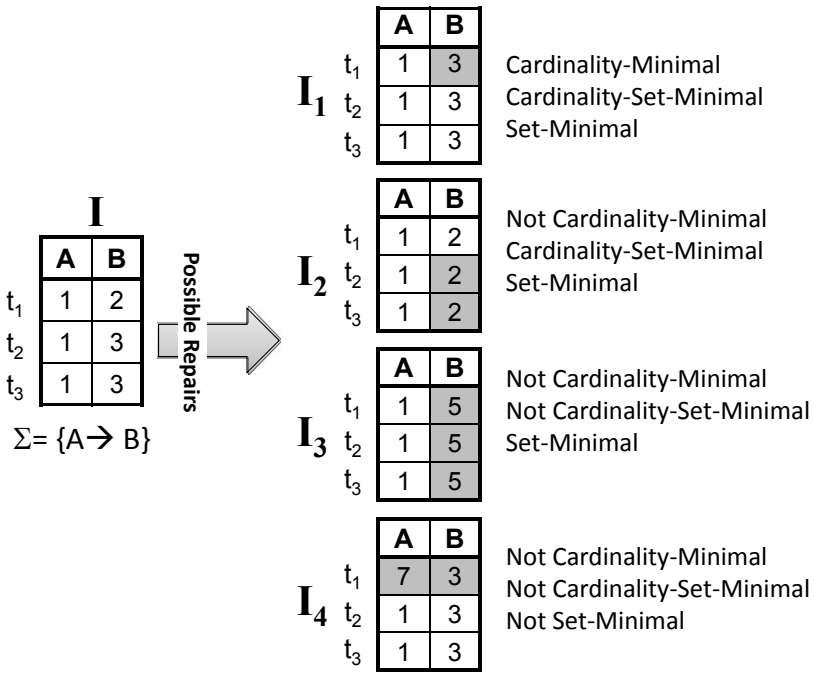

Figure 4.1: Examples of various types of repairs

Proof. For any two repairs I' and I'' of I,

$$\Delta(I, I'') \subset \Delta(I, I') \rightarrow |\Delta(I, I'')| < |\Delta(I, I')|$$

This implies that for any repair I' of I,

$$\nexists I'' \in Repairs(I)\ (|\Delta(I, I'')| < |\Delta(I, I')|)$$
$$\rightarrow \nexists I'' \in Repairs(I)\ (\Delta(I, I'') \subset \Delta(I, I'))$$

Therefore, if I' is a cardinality-minimal repair, I' is cardinality-set-minimal. Similarly, for any two repairs I' and I'' of I,

$$\lambda(I, I'') \subset \lambda(I, I') \leftrightarrow \Delta(I, I'') \subset \Delta(I, I') \wedge \forall C \in \Delta(I, I'')\ (I''(C) = I'(C))$$

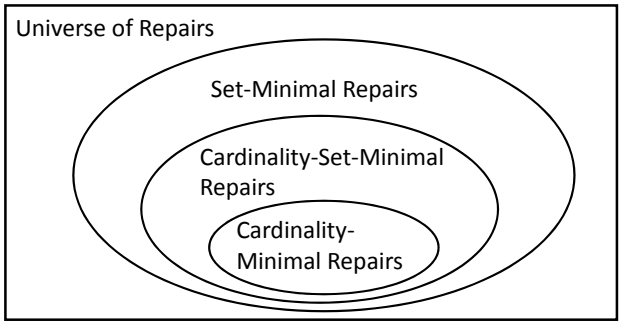

Figure 4.2: The relationship between spaces of possible repairs

and thus

$$\lambda(I, I'') \subset \lambda(I, I') \rightarrow \Delta(I, I'') \subset \Delta(I, I')$$

$$\nexists I'' \in Repairs(I)\ (\Delta(I, I'') \subset \Delta(I, I'))$$
$$\rightarrow \nexists I'' \in Repairs(I)\ (\lambda(I, I'') \subset \lambda(I, I'))$$

It follows that if I' is a cardinality-set-minimal repair, I' is set-minimal as well.

□

4.2 Sampling Possible Repairs

The main goal of our approach is to sample from a reasonable space of possible repairs. The sampling space should be neither too restrictive (and thus missing too many repairs), nor too large (and thus sampling repairs with very low probability of being correct). We argue that the cardinality-set-minimal space provides such balance, and we thus target sampling from this space.

Note that although existing heuristics for finding a single nearly-optimal repair may be modified to generate multiple random repairs, they do not give any guarantees on the space of generated repairs. For example, the algorithm in [19] can produce repairs that are not even set-minimal, while the algorithm from [57] may miss some cardinality-minimal repairs. We discuss these two cases in more details in Section 4.5.

The organization of this section is as follows. First, we introduce the concept of *clean cells* in Section 4.2.1 and we establish the link between this concept and the definition of cardinality-set-minimality. Then, we introduce a sampling algorithm in Section 4.2.2 that samples from the space of cardinality-set-minimal repairs. In Section 4.2.3, we describe how to enforce user-defined hard constraints that disallow modifying a given set of cells.

4.2.1 Sets of Clean Cells

Whenever attributes in R have unbounded domains (e.g., integers, and strings), there is an infinite number of FD repairs. For example, in Figure 4.1, assigning any value from the domain of A other than 1 to $t_1[A]$ results in a repair of I. If the domain of A is unbounded, then the number of such repairs is infinite. We use the notion of V-instances, which was introduced in [57], to concisely represent data instances. In V-instances, cells can be either set to constants, or to variables that can be instantiated in a specific way.

Definition 7. ***V-instance.*** *Given a set of variables $\{v_1^A, v_2^A, \dots\}$ for each attribute $A \in R$, a V-instance of R is an instance of R where each cell $t[A]$ in the instance can be assigned to either a constant in $Dom(A)$, or a variable from the set $\{v_1^A, v_2^A, \dots\}$.*

A V-instance I represents multiple ground (i.e., variable-free) instances of R that can be obtained by assigning each variable v_i^A in attribute A in I to any value from $Dom(A)$ that is not among the constants already occurring in attribute A in I, and such that no two distinct variables v_i^A and v_j^A have equal values. The main use of variables in the context of repairing FD violations is representing unknown values that emerge from modifying the left-hand-side attributes of a violated FD. In the remainder of the dissertation, we refer to a V-instance as simply an instance.

In the following, we define the concept of *clean cells*, and we establish the link between this concept and the cardinality-set-minimality.

We define a clean set of cells $\mathcal{C} \subseteq CIDs(I)$ with respect to a set of FDs Σ as follows.

Definition 8. ***Clean Cells.*** *A set of cells $\mathcal{C}$ in an instance I is clean iff there is at least one repair $I' \in Repairs(I)$ such that $\forall C \in \mathcal{C},\ I'(C) = I(C)$.*

That is, a set of cells in an instance I is clean if their values in I can remain unchanged while obtaining a repair of I. For example, in Figure 4.1, the sets $\{t_1[A], t_1[B], t_2[A]\}$ and $\{t_1[B], t_2[A], t_2[B]\}$ are clean, while the set $\{t_1[A], t_1[B], t_2[A], t_2[B]\}$ is not clean.

We say that a set of cells $\mathcal{C}$ is a *maximal* clean set iff $\mathcal{C}$ is clean and no strict superset of $\mathcal{C}$ is clean. For example, the sets $\{t_1[A], t_1[B], t_2[A], t_3[B]\}$ and $\{t_1[A], t_2[A], t_2[B], t_3[A], t_3[B]\}$ in Figure 4.1 are maximal clean sets. In the following theorem, we establish the link between the concept of clean cells and the cardinality-set-minimal repairs.

Theorem 1. *Given an input instance I and a set of FDs Σ, a repair I' of I with respect to Σ is cardinality-set-minimal iff the set of unchanged cells in I' (i.e., $CIDs(I') \setminus \Delta(I, I')$) is a maximal clean set of cells.*

Proof. First, we prove the "if" condition as follows. Let $\mathcal{C} = CIDs(I') \setminus \Delta(I, I')$ be a maximal clean set of cells. It follows that we cannot add any cell to $\mathcal{C}$ without making $\mathcal{C}$ unclean. Based on the definition of clean cells (Definition 8), there does not exist any other repair of I that have a set of unchanged cells $\mathcal{C}'$ that is a strict superset of $\mathcal{C}$ (i.e., $\nexists I'' \in repairs(I)(\Delta(I, I'') \subset \Delta(I, I'))$. Thus, I' is a cardinality-set-minimal repair.

Second, we prove the "only if" condition as follows. Let I' be a cardinality-set-minimal repair of I. The set $\mathcal{C} = CIDs(I') \setminus \Delta(I, I')$ is a clean set of cells because I' is a repair. Because I is cardinality-set-minimal, no cells in $\Delta(I, I')$ can be reverted back to their original values without violating Σ, even if we allow remodifying other changed cells (i.e., $\nexists I'' \in repairs(I)(\Delta(I, I'') \subset \Delta(I, I'))$). It follows that we cannot extend $\mathcal{C}$ by adding one or more cells without violating the cleanness property. It follows that $\mathcal{C}$ is a maximal clean set of cells.

□

Our sampling algorithm is based on Theorem 1. We randomly pick a maximal clean set of cells $\mathcal{C}$, and then we randomly change cells outside $\mathcal{C}$ in order to satisfy Σ.

In the following, we show how to determine whether a set of cells is clean or not. We observe that it is not enough to verify that the cells in $\mathcal{C}$ do not violate any FDs to determine cleanness of $\mathcal{C}$. For example, consider Figure 4.3, which shows a set of non-empty cells in

an instance. Assume that we need to determine if the shown cells are clean. Although the shown cells do not directly violate any FD in Σ (i.e., we cannot find a pair of tuples that violates Σ), no repair may contain the current values of those cells regardless of the values of the other cells. This is because $t_1[A] = t_2[A]$ implies $t_1[C] = t_2[C]$ (by $A \rightarrow C$) and $t_2[B] = t_3[B]$ implies that $t_2[C] = t_3[C]$ (by $B \rightarrow C$). Thus, $t_1[C]$,$t_2[C]$ and $t_3[C]$ have to be equal in any repair. However, $t_1[C] \neq t_3[C]$ in the shown instance.

To determine whether a set of cells $\mathcal{C}$ is clean or not, we capture all equality constraints over cells in I that are induced by values of cells in $\mathcal{C}$, and FDs in Σ. Then, we check for contradictions between the constraints and values of cells in $\mathcal{C}$ to determine whether $\mathcal{C}$ is clean or not. We model equality constraints as an equivalence relation over cells in I, denoted $\mathcal{E}$. We denote by $ec(\mathcal{E}, C_i)$ the equivalence class $E \in \mathcal{E}$ to which a cell C_i belongs. We denote by *merging* two equivalence classes in $\mathcal{E}$ replacing them by a new equivalence class that is equal to their union. Algorithm 4 builds the equivalence relation $\mathcal{E}$ given a set of cells $\mathcal{C}$ in an instance I.

Algorithm 4 `BuildEquivRel(`$\mathcal{C}$`,`I`,`Σ`)`

1: let $TIDs(\mathcal{C})$ be the set of tuple identifiers involved in $\mathcal{C}$: $\{t : t[A] \in \mathcal{C}\}$
2: let $Attrs(\mathcal{C})$ be the set of attributes involved in $\mathcal{C}$: $\{A : t[A] \in \mathcal{C}\}$
3: let $\mathcal{E}$ be an initial equivalence relation on the set $\{t[A] : t \in TIDs(\mathcal{C}), A \in Attrs(\mathcal{C})\}$ such that cells in $\mathcal{C}$ that belong to the same attribute and have equal values in I are in the same equivalence class, and all other cells outside $\mathcal{C}$ belong to separate (singleton) classes
4: **while** $\exists t_1, t_2 \in TIDs(\mathcal{C}), A \in Attrs(\mathcal{C}), X \subset Attrs(\mathcal{C})$ such that $X \rightarrow A \in \Sigma$, $\forall B \in X$ $(ec(\mathcal{E}, t_1[B]) = ec(\mathcal{E}, t_2[B]))$, and $ec(\mathcal{E}, t_1[A]) \neq ec(\mathcal{E}, t_2[A])$ **do**
5: merge the equivalence classes $ec(\mathcal{E}, t_1[A])$ and $ec(\mathcal{E}, t_2[A])$
6: **end while**
7: **return** $\mathcal{E}$

Algorithm 4 is similar in spirit to the *chase* algorithm that is frequently used in the context of data exchange and consistent query answering for dependency enforcement (e.g., [46, 34]).

Figure 4.3 shows an example of the initial and the final equivalence relations that are

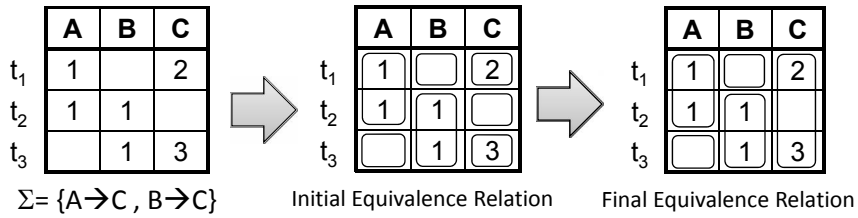

Figure 4.3: An example of checking whether the set of non-empty cells is clean or not

built by Algorithm 4. The equivalence class $\{t_1[C], t_2[C], t_3[C]\}$ in the final equivalence relation indicates that the three cells must be equal in any repair in which the non-empty cells are unchanged. This is clearly infeasible since $t_1[C]$ and $t_3[C]$ have different values in the shown example, which means that the non-empty cells in the figure are unclean.

In general, a set of cells $\mathcal{C}$ in I is clean with respect to Σ, denoted $isClean(\mathcal{C}, I, \Sigma)$, iff every two cells in $\mathcal{C}$ that belong to the same equivalence class in $\mathcal{E}$ have the same value in I. This result is formally described by the following theorem.

Theorem 2. *$isClean(\mathcal{C}, I, \Sigma)$ is True iff $\forall C_i, C_j \in \mathcal{C}$ such that $ec(\mathcal{E}, C_i) = ec(\mathcal{E}, C_j)$, $I(C_i) = I(C_j)$, where $\mathcal{E}$ is the outcome of the procedure* `BuildEquivRel(C,I,Σ)`.

Proof. We prove the "only if" direction as follows. Let $\mathcal{C}$ be a clean set of cells in I, and let $\mathcal{I}$ be the non-empty subset of repairs of I that have the cells in $\mathcal{C}$ unchanged (i.e., $\forall I' \in \mathcal{I}\ (\Delta(I, I') \cap \mathcal{C} = \phi)$). First, we prove that for all two cells that belong to the same equivalence class in $\mathcal{E}$, they must have equal values in every $I' \in \mathcal{I}$. Based on Algorithm 4, for every two cells $t_1[A]$ and $t_2[A]$ that belong to the same equivalence class, and for all $I' \in \mathcal{I}$, we have two possibilities:

- $t_1[A]$ and $t_2[A]$ belong to $\mathcal{C}$ and $I(t_1[A]) = I(t_2[A])$, and thus $I'(t_1[A]) = I'(t_2[A])$, or
- there exists an FD $X \rightarrow A \in \Sigma$ such that for all $B \in X$, $t_1[B]$ and $t_2[B]$ belong to the same equivalence class. Recursively, we can prove that for all $B \in X$, $I'(t_1[B]) = I'(t_2[B])$. The fact $I' \models X \rightarrow A$ implies that $I'(t_1[A]) = I'(t_2[A])$.

Because cells in $\mathcal{C}$ have identical values in I and I', we reach a similar conclusion for cells in $\mathcal{C}$ with respect to I: $\forall C_i, C_j \in \mathcal{C}\ (ec(\mathcal{E}, C_i) = ec(\mathcal{E}, C_j) \rightarrow I(C_i) = I(C_j))$.

We prove the "if" direction as follows. Consider the case where $\forall C_i, C_j \in \mathcal{C}$ $(ec(\mathcal{E}, C_i) = ec(\mathcal{E}, C_j) \rightarrow I(C_i) = I(C_j))$. We need to prove that the set $\mathcal{I}$ (i.e., the set of repairs of I that do not change cells in $\mathcal{C}$) is not empty. We construct one instance $I' \in \mathcal{I}$ as follows. We assign each cell in $\mathcal{C}$ in I' to the same value in I (i.e., $\forall C \in \mathcal{C}, I'(C) = I(C)$). We iterate over all other cells outside $\mathcal{C}$ in any random order. Each cell that belongs to a singleton equivalence class in $\mathcal{E}$ or belongs to a tuple that is not mentioned in $\mathcal{C}$ is set to a unique variable. For each cell C that belongs to a non-singleton equivalence class $E \in \mathcal{E}$ that includes at least one cell with an assigned value (call it x), we set $I'(C)$ to x. Finally, for each cell C that belongs to a non-singleton equivalence class $E \in \mathcal{E}$ whose cells are not assigned to any values yet, we assign C to a unique variable. This construction method ensures that cells that have equal values in I' are in the same equivalence class in $\mathcal{E}$, and vice versa.

Now, we show that the constructed instance I' is indeed a repair. For every two tuples $t_1, t_2 \in I'$ and for every FD $X \rightarrow A \in \Sigma$, $t_1[X] = t_2[X]$ implies that for all $B \in X$, $t_1[B]$ and $t_2[B]$ belong to the same equivalence class (based on our construction method). Therefore, $t_1[A]$ and $t_2[A]$ must belong to the same equivalence class as well (based on Algorithm 4), and thus $I'(t_1[A]) = I'(t_2[A])$. This proves that $I' \models \Sigma$ and thus $\mathcal{I}$ is not empty (i.e., cells in $\mathcal{C}$ are clean). □

Next, we show how to randomly pick a maximal clean set of cells, given I and Σ. We describe our procedure in Algorithm 5.

Algorithm 5 `MaxCleanSet(`I`,`Σ`)`

1: Define a set $CleanSet$ and initialize it to ϕ
2: **for** each cell $C \in CIDs(I)$ (based on a random iteration order) **do**
3: $\quad \mathcal{E} \leftarrow$ `BuildEquivRel`$(CleanSet \cup \{C\}, I, \Sigma)$
4: $\quad$ **if** $isClean(CleanSet \cup \{C\}, I, \Sigma) =$ `True`, based on $\mathcal{E}$ **then**
5: $\quad\quad CleanSet \leftarrow CleanSet \cup \{C\}$
6: $\quad$ **end if**
7: **end for**
8: **return** $CleanSet$

Algorithm 5 initially have an empty set of clean cells, and it randomly iterates over cells in I and attempts adding each cell to the clean set of cells, without violating the cleanness property. The algorithm terminates when all cells have been processed and returns the constructed clean set. In the following, we prove the correctness of the algorithm.

Lemma 2. *Sets of cells returned by Algorithm 5 are maximal clean sets.*

Proof. Given a set of clean cells returned by Algorithm 5, denoted $\mathcal{C}$, we need to prove that for any subset of $CIDs(I) \setminus \mathcal{C}$ (call it S), $\mathcal{C} \cup S$ is unclean.

First, we prove that if a set $\mathcal{C}$ is unclean, then any superset of $\mathcal{C}$ is unclean as well. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two sets of cells in an instance I such that $\mathcal{C}_1 \subset \mathcal{C}_2$. Let $\mathcal{E}_1$ (respectively, $\mathcal{E}_2$) be the outcome of `BuildEquivRel(`$\mathcal{C}_1$`,`I`,`Σ`)` (respectively, `BuildEquivRel(`$\mathcal{C}_2$`,`I`,`Σ`)`. By analyzing Algorithm 4, we reach that each equivalence class in $\mathcal{E}_1$ must be contained in another equivalence class in $\mathcal{E}_2$. Therefore, if there exist two cells in $\mathcal{C}_1$ that belong to the same equivalence class in $\mathcal{E}_1$ and have different values in I (i.e., $\mathcal{C}_1$ is unclean), the two cells must belong to the same equivalence class in $\mathcal{E}_2$, which means that $\mathcal{C}_2$ is unclean as well.

Assume, to the contrary, that $\exists S \subset CIDs(I) \setminus \mathcal{C}$ such that $\mathcal{C} \cup S$ is clean. Clearly, every cell C in S has been rejected at line 4 in Algorithm 5, which means that $\mathcal{C}_s \cup \{C\}$ is unclean, where $\mathcal{C}_s$ is the subset of $\mathcal{C}$ that is constructed up till the point of rejecting C. The set $\mathcal{C} \cup S$ is a superset of $\mathcal{C}_s \cup \{C\}$. Therefore, $\mathcal{C} \cup S$ is unclean, a contradiction.

□

Complexity Analysis

Let n be the number of tuples in the input instance I, and m be the number of attributes in I. In Algorithm 4, the maximum number of merges of equivalence classes is less than the number of tuples that appear in $\mathcal{C}$ multiplied by the number of attributes that appear in $\mathcal{C}$. Each merge operation can be done in a constant time (for all practical database sizes) using the find-union algorithm [74]. Therefore, the complexity of Algorithm 4 is in $O(n \cdot m)$. Evaluating *isClean* can be done in $O(n \cdot m)$ using a hash table structure. That is, all cells belonging to the same equivalence class are hashed to a unique bucket,

and we associate each bucket with the values of the inserted cells so far. Upon insertion of each cell, we only need to compare the cell value to the bucket value to determine the cleanness of the cells. A straightforward implementation of Algorithm 5 has a complexity of $O(n^2 \cdot m^2)$.

4.2.2 Sampling Cardinality-Set-Minimal Repairs

In this section, we present a randomized algorithm for generating cardinality-set-minimal repairs (Algorithm 6). This algorithm is a generalized version of the procedure we described in the proof of Theorem 2 to build a repair I'. The first step in the algorithm is constructing a maximal clean set of cells, denoted $MaxCleanCells$ (line 2). The algorithm iteratively cleans the cells outside $MaxCleanCells$ and adds them to a set called $Cleaned$. Initially, $Cleaned$ is equal to $MaxCleanCells$. In each iteration, the algorithm assigns a value to the current cell $t[A]$ such that $Cleaned \cup \{t[A]\}$ becomes clean. Specifically, if $t[A]$ belongs to a non-singleton equivalence class in $\mathcal{E}$ that contains other cells previously inserted in $Cleaned$, the only choice is to set $I'(t[A])$ to the same value as the other cells in the equivalence class (lines 5,6). Otherwise, we randomly choose one of the following three alternative values for $t[A]$: (1) a constant that is randomly selected from $Dom(A)$, (2) a variable that is randomly selected from the set of variables previously used in attribute A in I', or (3) a new variable v_j^A (line 8). For the first and second alternatives, we need to make sure that the selected constant or variable makes the set $Cleaned \cup \{t[A]\}$ clean. One simple approach is to keep picking a constant (similarly, a variable) at random until this condition is met. In the worst case, we can select up to n constants (similarly, n variables), where n is the number of tuples in the input instance. The third alternative, which is setting $I'(t[A])$ to a new variable, guarantees that the set $Cleaned \cup \{t[A]\}$ becomes clean. In fact, enforcing the third alternative at every iteration reduces Algorithm 6 to the repairing algorithm described in the proof of Theorem 2. The algorithm terminates when all cells have been added to $Cleaned$, and returns the resulting instance I'.

We show an example of executing Algorithm 6 in Figure 4.4. The algorithms obtains a maximal clean set, which is shown as the middle relation, and changes the two unclean cells $t_2[B]$ and $t_3[A]$. Because $t_2[B]$ exists in the same equivalence class as $t_1[B]$, the algorithm

Algorithm 6 `Gen-Repair(`I`,`Σ`)`

1: $I' \leftarrow I$
2: $MaxCleanCells \leftarrow$ `MaxCleanSet`(I, Σ)
3: $Cleaned \leftarrow MaxCleanCells$
4: $\mathcal{E} \leftarrow$ `BuildEquivRel`$(Cleaned, I, \Sigma)$
5: **for** each $t[A] \in CIDs(I) \setminus MaxCleanCells$ (based on a random iteration order) **do**
6: **if** $t[A]$ belongs to a non-singleton equivalence class in $\mathcal{E}$ that contains other cells in $Cleaned$ **then**
7: assign $I'(t[A])$ to the value (either a constant or a variable) of the other cells in $ec(\mathcal{E}, t[A]) \cap Cleaned$
8: **else**
9: randomly set $I'(t[A])$ to one of three alternatives: a randomly selected constant from $Dom(A)$, a randomly selected variable v_i^A that was previously used in I', or a fresh variable v_j^A such that $Cleaned \cup \{t[A]\}$ becomes clean
10: **end if**
11: $Cleaned \leftarrow Cleaned \cup \{t[A]\}$
12: $\mathcal{E} \leftarrow$ `BuildEquivRel`$(Cleaned, I', \Sigma)$
13: **end for**
14: **return** I'

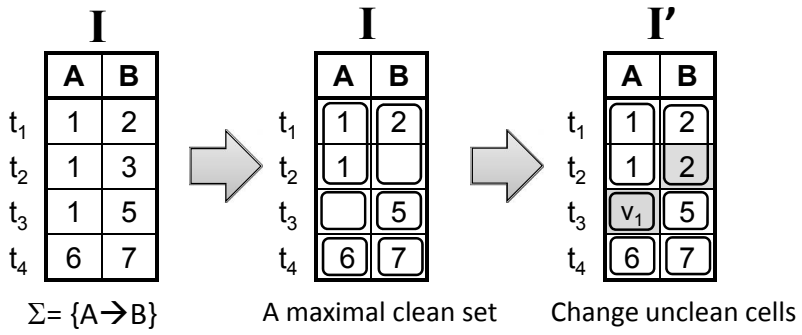

Figure 4.4: An example of executing Algorithm 6

assigns the value of $t_1[B]$ to $t_2[B]$. For the cell $t_3[A]$, the algorithm chooses to assign a fresh variable, v_1, to it.

In the following theorem, we prove the correctness on Algorithm 6.

Theorem 3. *Every instance that is generated by Algorithm 6 is a cardinality-set-minimal repair. Additionally, all cardinality-set-minimal repairs can be generated by Algorithm 6.*

Proof. First, we prove that every instance I' generated by Algorithm 6 is a repair of I with respect to Σ. In other words, we need to show that all cells in a generated repair I' represent a clean set. Initially, the set $Cleaned = MaxCleanCells$ is clean with respect to Σ (based on Theorems 1 and 2). In each iteration, the algorithm adds a cell to $Cleaned$ and changes this cell to ensure that the resulting version of $Cleaned$ is clean as well. Upon termination, all cells in I' are in $Cleaned$, which indicates that the resulting instance I' satisfies Σ.

Second, we prove that each generated repair is cardinality-set-minimal. The initial maximal clean set of cells, denoted $MaxCleanCells$, is not modified throughout the algorithm. Thus, the set of unchanged cells in any generated repair represents a maximal clean set of cells, which indicates that the generated repair is cardinality-set-minimal based on Theorem 1.

Third, we prove that every cardinality-set-minimal repair can be generated by Algorithm 6. Every cardinality-set-minimal repair I' corresponds to a maximal clean set of cells, denoted $\mathcal{C}$ (Theorem 1). Algorithm 5 produces set $\mathcal{C}$ when all cells in $\mathcal{C}$ are processed first.

Regardless of the iteration order in which Algorithm 6 processes the set $CIDs(I') \setminus \mathcal{C}$ (line 5), each cell C processed in lines 6-10 can be assigned to the value in I' to make $Cleaned \cup \{C\}$ clean. Assuming otherwise implies that there exists a subset of cells in I' that is not clean, which contradicts the fact that I' is a repair. It follows that any cardinality-set-minimal repair I' can be generated by Algorithm 6. □

Complexity Analysis

Obtaining a maximal clean set of cells costs $O(n^2 \cdot m^2)$, where n denotes the number of tuples in I and m denoted the number of attributes. In Algorithm 6, the number of cleaning iterations is at most equal to the number of cells in I' (i.e., $n \cdot m$). In each iteration, Algorithm 4 is invoked to build the equivalence classes of cells in $Cleaned$. Additionally, the condition $isClean$ can be evaluated for all possible constants and variables that appear in the attribute A in I' (in the worst case). Hence, the complexity of each iteration is $O(m \cdot n^2)$ and the overall complexity of Algorithm 6 is $O(m^2 \cdot n^3)$. Note that if we restrict changing cells in line 9 to the third alternative only (i.e., assigning new variables to the cells), the complexity is reduced to $O(n^2 \cdot m^2)$. Additionally, we reduce the runtime of the algorithm by implementing several optimizations to avoid recomputing the equivalence relation from scratch in every iteration.

4.2.3 User-defined Hard Constraints

In this section, we describe a simple modification to our approach to generate random repairs that satisfy user-defined hard constraints. We consider constraints that specify a set of immutable cells $\mathcal{T}$. Such cells are considered trustworthy (i.e., already clean cells), and thus the cleaning algorithm is required to keep their values unchanged.

Since the cleaning algorithm cannot change an immutable cell when generating a repair, we must first ensure that $\mathcal{T}$ itself is clean. To check the cleanness of $\mathcal{T}$, we build the equivalence relationship $\mathcal{E}_{\mathcal{T}}$ over $\mathcal{T}$ using Algorithm 4 and invoke Theorem 2. If $\mathcal{T}$ is found to be unclean, we return an empty answer. In the remainder of this section, we assume that $\mathcal{T}$ is clean.

In the following, we describe our modifications to the cleaning algorithm. When creating a maximal clean set of cells using Algorithm 5, we insert the cells in $\mathcal{T}$ first into the set $CleanSet$ (i.e., we initialize $CleanSet$ to $\mathcal{T}$ at line 1 in the algorithm). The remainder of the algorithm remains unchanged. Finding a maximal clean set that is a super set of $\mathcal{T}$ is possible as long as $\mathcal{T}$ is clean. This modification produces repairs in which none of the cells in $\mathcal{T}$ are changed since Algorithms 6 does not change the cells in the maximal clean set generated by Algorithm 5.

4.3 Block-wise Repairing

In this section, we improve the efficiency of generating repairs by partitioning the input instance I into disjoint blocks, each of which represents a subset of cells in I, such that blocks can be repaired independently. Such partitioning effectively splits a problem instance into a number of smaller instances, which results in a significant increase in performance. Also, partitioning I into disjoint blocks allows parallelization of the cleaning process (i.e., all blocks can be repaired in parallel). Furthermore, because sub-repairs of individual blocks are independent, we effectively generate an exponentially larger number of repairs, which represents all possible combinations of sub-repairs. That is, if instance I is partitioned into r blocks, and we generated k repairs for each partition, the sample size is effectively equal to k^r.

A simple strategy for partitioning I is to partition the attributes in R into multiple disjoint groups such that no FD in Σ spans more than one group of attributes (a.k.a., vertical partitioning). However, this strategy has a limited impact on the performance as it fails to reduce the number of tuples in each partition, which is the main complexity factor.

In order to allow more aggressive partitioning of the input instance, where each block represents a set of cells, we have to restrict the values that can be assigned to cells at line 9 in Algorithm 6 to new variables (i.e., the third alternative). Such restriction ensures that the modified cell $t[A]$ can never be equal to any other cell $t'[A]$ in other blocks. Thus, $t[A]$ cannot be a part of a violation of an FD that contains A in the left-hand-side attributes. We refer to the modified versions of Algorithm 6 as Algorithm `Block-Gen-Repair`. Note that the modified version might miss some cardinality-set-minimal repairs as a result of restricting the new values of the changed cells.

Modifying line 9 in Algorithm 6 allows deleting line 12 from the algorithm, which reconstructs the equivalence relation $\mathcal{E}$ after modifying each cell. The reason is that data changes performed in line 7 and the modified version of line 9 do not alter the equivalence relation $\mathcal{E}$. The only possible change to $\mathcal{E}$ in the original version of Algorithm 6 is caused by merging two equivalence classes due to changing a cell in line 7 to a constant or a variable that already exist in I' (splitting an equivalence class is not possible under any

Algorithm 7 `Partition`(I, Σ)

1: $\mathcal{E}_0 =$ `BuildEquivRel`$(CIDs(I), I, \Sigma)$
2: Initialize the set of blocks $\mathcal{P}$ such that each cell in I belongs to a separate block
3: **for** each $X \rightarrow A \in \Sigma$ **do**
4: **for** each pair of tuples $t_i, t_j \in I$ such that $\forall B \in X, ec(\mathcal{E}_0, t_i[B]) = ec(\mathcal{E}_0, t_j[B])$ **do**
5: merge the blocks of the cells $t_i[X] \cup t_i[A] \cup t_j[X] \cup t_j[A]$
6: **end for**
7: **end for**
8: **return** $\mathcal{P}$

circumstances). This case is not possible after modifying line 9 as described.

In the following, we describe our partitioning algorithm. Let $\mathcal{E}_0$ be the equivalence relation that is constructed over all cells in I (i.e., `BuildEquivRel`$(CIDs(I), I, \Sigma)$). $\mathcal{E}_0$ clusters cells into equivalence classes such that all pairs of cells that might have equal values throughout the execution of `Block-Gen-Repair`(I, Σ) belong to the same equivalence class (refer to the proof of Theorem 4). It follows that cells that belong to different equivalence classes can never have equal values. For example, Figure 4.5 shows an instance I and the corresponding equivalence relation $\mathcal{E}_0$. Cells $t_1[C], t_2[C]$ and $t_3[C]$ belong to the same equivalence class, which means that they may have equal values in some generated repairs. On the other hand, $t_1[B]$ and $t_2[B]$ belong to different equivalence classes, meaning that they can never have equal values.

We use the equivalence relation $\mathcal{E}_0$ to partition the input instance I such that any two tuples that belong to different blocks can never have equal values for the left-hand-side attributes X, for all $X \rightarrow A \in \Sigma$ (details are in Algorithm 7). Thus, any violation of FDs throughout the course of repairing I cannot span more than one block. In other words, repairing every block separately results in a repair for the entire instance I.

In Figure 4.5, we show an example of partitioning an instance. Initially, an equivalence relation $\mathcal{E}_0$ is constructed on the input instance by invoking `BuildEquivRel`$(CIDs(I), I, \Sigma)$. Each equivalence class is represented as a rectangle that surrounds the class members. We initially assign each cell to a separate block (i.e., cell $t_1[A]$ belongs to P_1, cell $t_2[A]$ belongs to P_2, and so on). For each FD $X \rightarrow A$, we locate

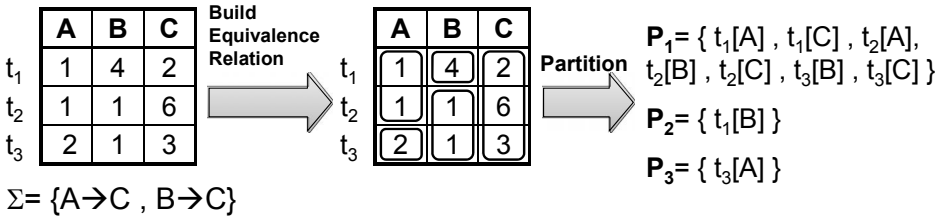

Figure 4.5: An example of partitioning an instance

tuples whose attributes X belong to the same equivalence classes and we merge the blocks of attributes XA of those tuples. For example, since the cells $t_1[A]$ and $t_2[A]$ belong to the same equivalence class and the FD $A \rightarrow C \in \Sigma$, we merge the blocks of $t_1[A]$, $t_2[A]$, $t_1[C]$, and $t_2[C]$. We continue the partitioning algorithm and we return the final partitioning that is shown in the figure.

We prove in Theorem 4 that the blocks generated by Algorithm 7 can be repaired separately using Algorithm `Block-Gen-Repair`.

Theorem 4. *Blocks of an instance I that are constructed by Algorithm* ***`Partition`*** *can be repaired separately using Algorithm* ***`Block-Gen-Repair`****.*

Proof. We first prove that, for any two cells $t_1[A]$ and $t_2[A]$, if there exists any possible repair I' generated by Algorithm `Block-Gen-Repair` in which $t_1[A]$ and $t_2[A]$ have the same value, $t_1[A]$ and $t_2[A]$ must be in the same equivalence class in $\mathcal{E}_0$. If $t_1[A]$ and $t_2[A]$ have equal values in I, they belong to the same equivalence class due to the initial step in creating $\mathcal{E}_0$ (line 3 in Algorithm `BuildEquivRel`). Otherwise, $t_1[A]$ and $t_2[A]$ have different values in I, and at least one of them has been modified by Algorithm `Block-Gen-Repair` to have equal values in I'. After modifying line 9 in Algorithm 6, we only assign new variables to cells in line 9 (i.e., they cannot be equal to any other cell). Therefore, the changed cells (i.e., $t_1[A]$, $t_2[A]$, or both) must have been changed in line 7 in (modified) Algorithm 6. Thus, both cells have to belong to the same equivalence class E in the equivalence relation $\mathcal{E}$ created by the repairing algorithm at line 4. Because the original values of $t_1[A]$ and $t_2[A]$ in I are different, E must have been created based on an FD $X \rightarrow A \in \Sigma$ (refer to Algorithm 4). That is, there exists $X \rightarrow A \in \Sigma$ such that for all $B \in X$, $t_1[B]$ and $t_2[B]$

belong to the same equivalence class in $\mathcal{E}$ that is maintained by the repairing algorithm. Because any repair must satisfy the constraints imposed by the relation $\mathcal{E}$, we deduce that for all $B \in X$, $I'(t_1[B]) = I'(t_2[B])$. We recursively prove that for all $B \in X$, $t_1[B]$ and $t_2[B]$ belong to the same equivalence class in $\mathcal{E}_0$. Based on Algorithm `BuildEquivRel` and FD $X \rightarrow A$, $t_1[A]$ and $t_2[A]$ must be in the same equivalence class in $\mathcal{E}_0$ as well.

A direct result is that the possible *constant* values of a cell $t[A]$, denoted $PV_I(t[A])$, in any repair I' of I that is randomly generated by the modified algorithm are values of the cells in the same equivalence class of $t[A]$ in $\mathcal{E}_0$ in I. That is, $PV_I(t[A]) = \{I(t'[A]) : ec(\mathcal{E}_0, t[A]) = ec(\mathcal{E}_0, t'[A])\}$. Therefore, for two cells $t_1[A]$ and $t_2[A]$ that belong to different equivalence classes in $\mathcal{E}_0$, $PV_I(t_1[A]) \cap PV_I(t_2[A]) = \phi$. Let $PV_P(t[A])$ be the possible constant values of $t[A]$ in a randomly generated repair P' of a block P that contains $t[A]$. $PV_P(t[A]) \subseteq PV_I(t[A])$ because each equivalence class in $\mathcal{E}'_0 = \texttt{BuildEquivRel}(CIDs(P), I, \Sigma)$ is contained in an equivalence class in $\mathcal{E}_0 = \texttt{BuildEquivRel}(CIDs(I), I, \Sigma)$ based on Algorithm 4. It follows that for two cells $t_1[A]$ and $t_2[A]$ that belong to different equivalence classes in $\mathcal{E}_0$ and to different blocks (P_1 and P_2, respectively), $PV_{P_1}(t_1[A]) \cap PV_{P_2}(t_2[A]) = \phi$.

Let $P_1, \ldots, P_r$ be the blocks of I generated by Algorithm 7, and let P'_i be a repair of P_i generated by Algorithm `Block-Gen-Repair`. Now, we prove that the instance I' that represents the union of all blocks $P'_1, \ldots, P'_r$ satisfies Σ. We approach the proof by contradiction. Assume that there exists a violation of $X \rightarrow A$ by two tuples t_1 and t_2 in I'. For all $B \in X$, $I'(t_1[B]) = I'(t_2[B])$. Because blocks have been repaired independently, the variables created in each block are disjoint. Thus, values of $t_i[B]$, for $i \in \{1, 2\}, B \in X$, must be constants (i.e., $I'(t_i[B]) \in PV_P(t_i[B])$, where P is the block containing $t_i[B]$). Cells of the violation have to span multiple blocks because sub-repairs of individual blocks cannot violate Σ. Hence, based on the partitioning algorithm (Algorithm 7), there must exist an attribute $B \in X$ such that $t_1[B]$ and $t_2[B]$ belong to different equivalence classes in $\mathcal{E}_0$. Based on our previous finding, $PV_{P_1}(t_1[B]) \cap PV_{P_2}(t_2[B]) = \phi$, where P_1 contains $t_1[B]$ and P_2 contains $t_2[B]$, which contradicts the assumption that $I'(t_1[B]) = I'(t_2[B])$.

□

Complexity Analysis

Algorithm 7 runs in $O(n \cdot m)$, where n is the number of tuples in I and m is the number of attributes. Building the equivalence relation $\mathcal{E}_0$ is performed in $O(n \cdot m)$. Furthermore, there is at most $O(n \cdot m)$ merges done in lines 3-7 in Algorithm 7, each of which can be done in a constant time (for all practical database sizes) [74]. It follows that the overall complexity is $O(n \cdot m)$.

4.4 Experimental Study

In this section, we preset an experimental evaluation of our approach. The goal of our experiments is twofold. First, we show that the proposed algorithms can efficiently generate random repairs. Second, we use our repair generator to study the correlation between the number of changes in a repair and the quality of the repair. To provide a reference point, we implemented two previous approaches that deterministically repair FD violations.

4.4.1 Setup

All experiments were conducted on a SunFire X4100 server with a Dual Core 2.2GHz processor, and 8GB of RAM. All computations are executed in memory. We use synthetic data that is generated by a modified version of the UIS Database generator [2]. This program produces a mailing list that has the following schema: `RecordID, SSN, FirstName, MiddleInit, LastName, StNumber, StAddr, Apt, City, State, ZIP`. The following FDs are defined:

- `SSN` $\rightarrow$ `FirstName, MiddleInit, LastName, StNumber, StAddr, Apt, City, State, ZIP`
- `FirstName, MiddleInit, LastName` $\rightarrow$ `SSN, StNumber, StAddr, Apt, City, State, ZIP`
- `ZIP` $\rightarrow$ `City, State`

The UIS data generator was originally created to construct mailing lists that have duplicate records. We modified it to generate two instances: a clean instance I_c and another instance I_d that is obtained by marking random perturbations to cells in I_c. These perturbations include modifying characters in attributes, swapping the first and last names, and replacing SSNs with all-zeros to indicate missing values. To control the amount of perturbation, we use a parameter P_{pert} that represents the probability of modifying one or more attributes of each tuple $t \in I_c$. We use four approaches to clean the instance I_d that are described as follows.

- `Holistic`: This approach implements Algorithm 6. To modify a cell in line 9, we randomly pick an alternative, and for the first two alternatives, we keep picking a constant or variable at random until set *Cleaned* is clean.
- `Block-wise`: This approach partitions the input instance using Algorithm 7 into disjoint blocks, and then uses Algorithm `Block-Gen-Repair` to separately repair each block (refer to Section 4.3).
- `Vertex-Cover` [57]: This approach is based on modeling FD violations as hyper-edges and using an approximate minimum vertex cover of the resulting hyper-graph to find a repair with a small number of changes.
- `Greedy-RHS` [19]: This approach repeatedly picks the violation with the minimum cost to repair and fixes it by changing one or more cells. Modifications are only performed to the right-hand-side attributes of the violated FDs.

4.4.2 Performance Analysis

In Figure 4.6(a), we show the running time for generating one repair for various data sizes. We report the average runtime for generating five repairs. For Algorithm `Block-wise`, the cost of the initial partitioning of the input instance is amortized across the generated repairs.

Algorithm `Greedy-RHS` provides the best scalability, however, at the cost of providing poor output quality as we describe in Section 4.4.3. Algorithms `Block-wise` is ranked

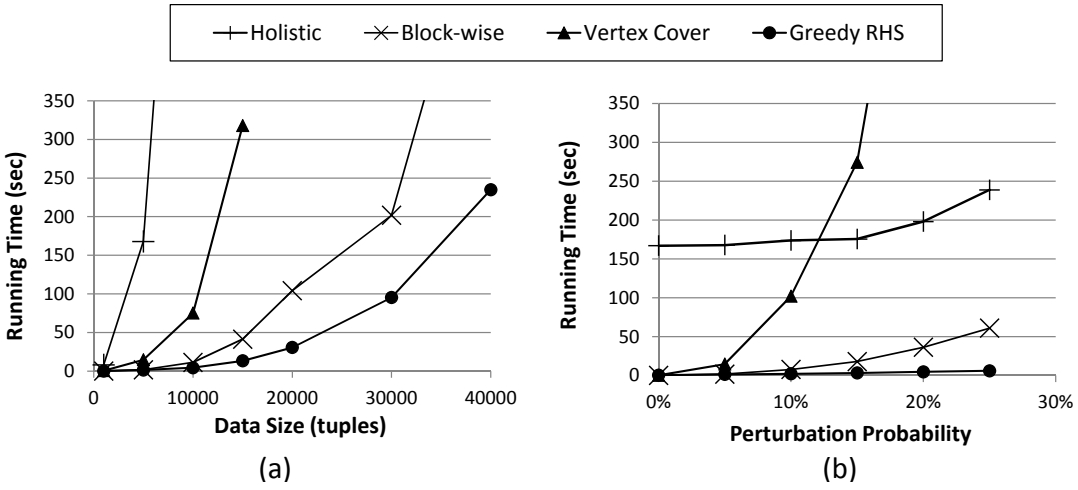

Figure 4.6: The running time for generating a repair

second and it outperforms the holistic version of the algorithm by orders of magnitude. For example, repairing 10000 tuples by Algorithm `Block-wise` took 11 seconds, while Algorithm `Holistic` took 1050 seconds. Algorithm `Vertex-Cover` is ranked third. We noticed that memory requirements of Algorithm `Vertex-Cover` grow quickly as the number of violations increases due to the large number of the hyper-edges in the initial hyper-graph (e.g., 2.2 million hyper-edges when the input instance contains 15000 tuple).

The running time of our sampling approached is almost linear in the number of generated repairs (i.e., the sample size) because the running time for generating a random repair has very low discrepancy.

Figure 4.6(b) depicts the running time of the four algorithms for various levels of errors in the input instance, which is captured by parameter P_{pert}. Note that Algorithm `Holistic` incurs a large overhead even when the input database is clean. This is because Algorithms 5 and 6 process all database cells one-by-one and check for the cleanness of the processed cells with respect to the previously inserted cells. On the other hand, Algorithm `Block-wise` eliminates such overhead by splitting the input instance into a large number of blocks that can be repaired more efficiently.

4.4.3 The Relation between the Number of Changes and Repair Quality

In this section, we use our repair sampling algorithm to study the correlation between the number of changes in a repair and the quality of the repair, given that the ground truth is available. This study allows for verifying the concept of minimality of changes. For completeness, we also show the characteristics of the repairs generated by other deterministic approaches.

We use the precision (i.e., the percentage of correct data changes) of the performed changes with respect to the given ground truth as a quality metric. Note that we do not report the recall (i.e., the percentage of errors that have been corrected), because several errors in the data set do not lead to violations of FDs (e.g., typos in the first name).

We use the clean instance I_c as the ground truth to assess the quality of a given repair I_r. First, we show how to count the number of correct changes in I_r. We denote by $CC(I_r)$ the set of cells that has been correctly fixed in I_r.

$$CC(I_r) = \{C \in \Delta(I_d, I_r) : I_r(C) = I_c(C) \wedge I_d(C) \neq I_c(C)\}$$

Replacing an incorrect value of a cell in I_d (with respect to I_c) with a variable can be considered as a *partially correct change*. We denote by $CVC(I_r)$ the set of cells that are partially corrected.

$$CVC(I_r) = \{C \in \Delta(I_d, I_r) : I_r(C) \textit{ is a variable } \wedge \ I_d(C) \neq I_c(C)\}$$

We define the number of correct changes as the sum of the cardinality of $CC(I_r)$, and a fraction (0.5 in our experiments) of the cardinality of $CVC(I_r)$. We compute the precision of a repair I_r as the ratio between the number of correct changes in I_r to the total number of changes in I_r.

We measured the precision of the repairs generated by all approaches: `Holistic`, `Block-wise`, `Vertex-Cover`, and `Greedy-RHS`. However, for clarity of presentation, we omitted the results of Algorithm `Holistic`. Figure 4.7 shows the quality results of the

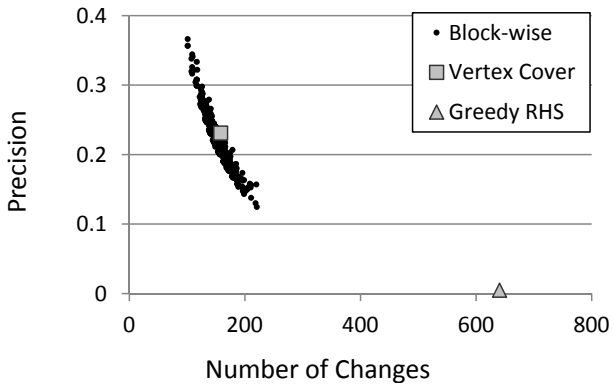

Figure 4.7: The precision of the generated repairs of a data set consisting of 5000 tuples with perturbation probability of 5%

other algorithms. The input instance consists of 5000 tuples and the parameter P_{pert} is set to 5%. Algorithms `Holistic` and `Block-wise` were executed 500 times (due to the randomness of the generating process) , while Algorithms `Vertex-Cover` and `Greedy-RHS` were executed once.

We found that the precision of the repairs generated by Algorithm `Holistic` is much lower than the repairs generated by Algorithm `Block-wise` (less than 0.1 in average). The reason is that the former algorithm can assign constant values to modified cells in line 9, which are very unlikely to be equal to the correct values. Algorithm `Block-wise` avoids such pitfall by always assigning a variable to represent a range of possible values.

Figures 4.7 shows the relationship between the number of changes and the precision of the resulting repair. The precision has a strong correlation with the number of changes (-0.95), which suggests that repairs with fewer changes have superior quality. This result also suggests the possibility of associating each generated repair with a confidence that is inversely proportional to the number of data changes. The exact formulation of the confidence of repairs will be targeted in our future work.

In Figure 4.7(c), we observe that Algorithm `Greedy-RHS` provides very low precision, compared to the other algorithms. The main reason is that this algorithm performs changes only to the right-hand-side attributes of FDs. Thus, errors in left-hand-side attributes of

FDs are always fixed in the wrong way. For example, missing SSNs are usually replaced by all-zeros. Algorithm `Greedy-RHS` changes all attributes of tuples with missing SSNs to the same value instead of replacing missing SSNs with variables.

Algorithm `Vertex-Cover` provides a relatively high precision compared to other approaches (Figure 4.7). However, a large number of repairs (around 50% of the repairs) generated by Algorithm `Block-wise` have better quality than those generated by Algorithm `Vertex-Cover`. The reason is that Algorithm `Vertex-Cover` uses an *approximate* minimum vertex cover to decide which cells should be changed (finding an exact minimum vertex cover is NP-hard). We also emphasize that even obtaining a single repair that has the fewest number of changes is not enough because there are several possible repairs that have the same number of changes.

Note that the precision is relatively low in all algorithms due to the high uncertainty about the right cells to modify. For example, given an FD $A \rightarrow B$, and two violating tuples t_1 and t_2, we have four cells that can be changed in order to repair the violation: $t_1[A]$, $t_1[B]$, $t_2[A]$, and $t_2[B]$. This uncertainty can be greatly reduced by considering additional information such as the user trust in various attributes and tuples (e.g., [19, 22, 57]. However, using this kind of information to improve data quality is beyond the scope of the paper.

4.5 Randomization of Previous Approaches

In this section, we give counterexamples to illustrate why previous approaches that produce a single repair (e.g., [19, 57]) are not suitable for generating a random sample of repairs.

First, we show that the algorithm introduced in [19] may generate repairs that are not set-minimal (i.e., contain unnecessary changes). The algorithm repairs an input instance by repeatedly searching for tuples that violate an FD $X \rightarrow A \in \Sigma$ and modifying attribute A of the violating tuples to have the same value. For example, in Figure 4.8, we show a possible repair I' of the input instance I that can be generated by the algorithm in [19]. Modified cells are shaded in the figure. The first step repairs a violation of $B \rightarrow C$ by associating the cells $t_2[C]$ and $t_3[C]$ to the same equivalence class and changing the cell

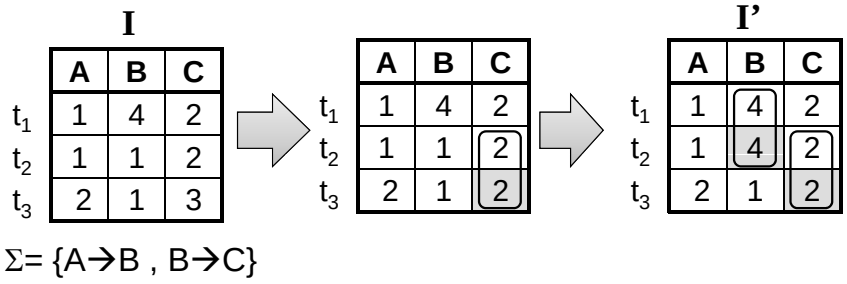

Figure 4.8: An example of a repair generated by the algorithm in [19] that is not set-minimal

$t_3[C]$ to 2. In the second step, a violation of $A \rightarrow B$ is fixed by changing $t_2[B]$ to 4. The resulting repair I' is not set-minimal because the cell $t_3[C]$ can be reverted to its value in I without violating any FD.

Repairs that are not set-minimal can be generated by the algorithm in [19] due to the fact that the generated repairs can involve contradicting assumptions. For example, the cell $t_2[B]$ is used in the first step for changing $t_3[C]$ to 2 (i.e., $t_2[B]$ is assumed to be a correct cell). In the second step, $t_2[B]$ is modified to 4, which implies that $t_2[B]$ is incorrect (i.e., unclean). We avoid such contradictions in our algorithm that is presented in Section 4.2. That is, once a cell C_i is used for modifying another cell C_j, C_i cannot be modified any further. We enforce this constraint by avoiding changing cells that are already in set *Cleaned* and only changing the cell currently being inserted (refer to Algorithm 6).

We show that some cardinality-minimal repairs cannot be generated by the approach presented in [57]. We illustrate this fact using the example in Figure 4.9. In Figure 4.9(a), we show the hyper-edges (also called double and triple conflicts) that exist in the initial conflict graph. The algorithm in [57] can only change a cell $t[B]$ if it appears in the initial conflict graph, or there exists an FD $X \rightarrow A \in \Sigma$ such that $B \in X$ and $t[A]$ appears in the initial conflict graph. It follows that the cell $t_2[E]$ in Figure 4.9 can never be changed by the algorithm. Therefore, the cardinality-minimal repair I' that is shown in Figure 4.9(b) cannot be generated by the algorithm in [57].

Some cardinality-minimal repairs cannot be generated by the algorithm in [57] because the algorithm is biased towards replacing the cells that belong to the left-hand-side at-

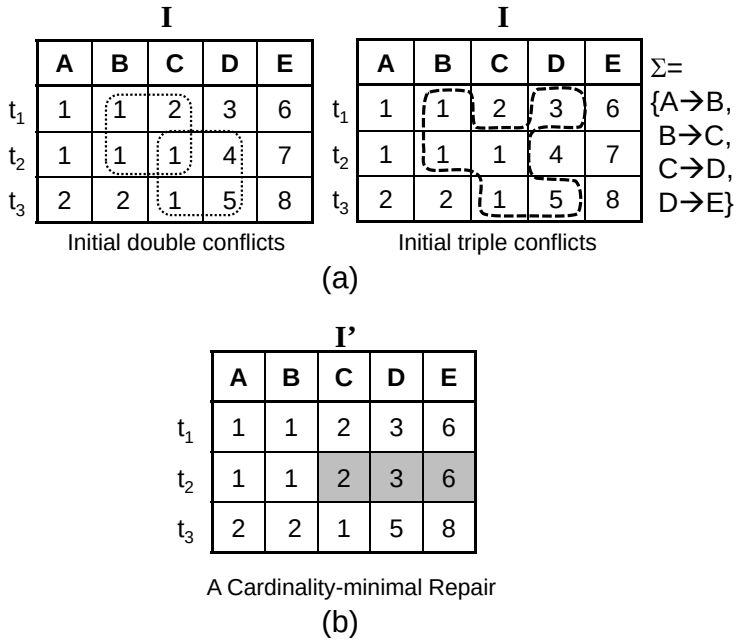

Figure 4.9: An Example of a cardinality-minimal repair that cannot be generated by the algorithm in [57]

tributes of FDs with new variables (step 2 of the algorithm presented in [57]). In our algorithms, we consider all possible values when changing a given cell that is involved in a violation as we describe in Section 4.2.

Chapter 5

Repairing Unclean Data and Unclean FDs

In this chapter, we discuss our approach for probabilistic data cleaning for the situation where both data and FDs are unclean. In Section 5.1, we introduce the notion of minimal repairs and we discuss the relative trust between data and FDs. We overview our cleaning approach in Section 5.2. In Section 5.3, we present our algorithm that obtains a repair of FDs, given a specific relative trust in data and FDs. In Section 5.4, we describe how to repair the data given a set of clean FDs. In Section 5.5, we show how to efficiently handle uncertainty in the relative trust. Finally, in section 5.6, we provide an experimental evaluation.

5.1 Spaces of Possible Repairs

In this section, we formally describe the problem of repairing data and FDs simultaneously, and we overview our approach. First, we define a space of minimal repairs of both data and FDs in Section 5.1.1. Then, we discuss the impact of having different trust in data and FDs in Section 5.1.2.

5.1.1 Minimal Repairs of Data and FDs

In Section 1.3, we advocated for two criteria to restrict the space of possible repairs of an unclean data instance I and a set of inaccurate FDs Σ. The criterion is to obtain repairs that involve the minimum amount of changes to data and FDs. The second criterion is to take into consideration the (relative) amount of errors in data and FDs, which reflects our relative trust in data versus FDs.

We focus on data repairs that change cells in I, rather than repairs that delete tuples from I. We denote by $\mathcal{S}(I)$ all possible repairs of I. All instances in $\mathcal{S}(I)$ have the same number of tuples as I. Because we aim at modifying a given set of FDs, rather than discovering a new set of FDs from scratch, we restrict the allowed FD modifications to those that relax, or weaken, the supplied FDs. We do not consider adding new constraints. That is, Σ' is a possible repair of Σ iff $I \models \Sigma$ implies $I \models \Sigma'$, for any data instance I. Given a set of FDs Σ, we denote by $\mathcal{S}(\Sigma)$ the set of all possible repairs of Σ resulting from relaxing the FDs in Σ in all possible ways. We define the universe of possible repairs as follows.

Definition 9. ***Universe of Data and FDs Repairs.*** *Given a data instance I and a set of FDs Σ, the universe of repairs of data and FDs, denoted $\mathbf{U}$, is the set of all possible pairs (Σ', I') such that $\Sigma' \in \mathcal{S}(\Sigma)$, $I' \in \mathcal{S}(I)$, and $I' \models \Sigma'$.*

In order to provide a practical solution, we focus on a subset of $\mathbf{U}$ that is large enough to cover a reasonable set of possible repairs, and can be generated efficiently. We achieve such a goal by focusing on repairs that are Pareto-optimal with respect to two distance functions: $dist_c(\Sigma, \Sigma')$ that measures the distance between two sets of FDs, and $dist_d(I, I')$ that measures the distance between two database instances. We refer to such repairs as *minimal repairs*.

For two vectors $V = (v_1, \ldots, v_k)$ and $W = (w_1, \ldots, w_k)$, we say that V dominates W, written $V \prec W$, iff $v_i \leq w_i$, for $i \in \{1, \ldots, k\}$, and at least one element v_j in V is strictly less than the corresponding element w_j in W. We define minimal repairs as follows.

Definition 10. ***Minimal Repair.*** *Given an instance I and a set of FDs Σ, a repair $(\Sigma', I') \in \mathbf{U}$ is minimal iff $\nexists (\Sigma'', I'') \in \mathbf{U}$ such that $(dist_c(\Sigma, \Sigma''), dist_d(I, I'')) \prec$*

$(dist_c(\Sigma, \Sigma'), dist_d(I, I'))$.

We deliberately avoid aggregating changes to data and changes to FDs into one metric in order to enable using various metrics for measuring both types of changes, which might be incomparable. For example, one metric for measuring changes in Σ is the number of modified FDs in Σ, while changes in I could be measured by the number of changed cells. Also, this approach provides a wide spectrum of Pareto-optimal repairs that ranges from completely trusting I (and only changing Σ) to completely trusting Σ (and only changing I).

For an instance repair I' of I, we denote by $\Delta_d(I, I')$ the cells that have different values in I and I'. We use the cardinality of $\Delta_d(I, I')$ to measure the distance between two instances, which has been widely used in previous data cleaning techniques (e.g., [19, 27, 57]). That is, $dist_d(I, I') = |\Delta_d(I, I')|$.

Recall that we restrict the modifications to Σ to those that relax the constraints in Σ. Thus, an FD F' is a possible repair of an FD F iff $I \models F \Rightarrow I \models F'$, for any instance I. We use a simple relaxation mechanism that satisfies this property: we only allow appending zero or more attributes to the left-hand-side (LHS) of an FD. Formally, an FD $X \rightarrow A \in \Sigma$ can be repaired by appending a set of attributes $Y \subseteq (R \setminus XA)$ to the LHS, resulting in an FD $XY \rightarrow A$. We disallow adding A to the LHS to prevent producing trivial FDs.

Note that different FDs in Σ might be modified to the same FD. For example, both $A \rightarrow B$ and $C \rightarrow B$ can be modified to $AC \rightarrow B$. Therefore, the number of FDs in any $\Sigma' \in \mathcal{S}(\Sigma)$ is less than or equal to the number of FDs in Σ. We maintain a mapping between each FD in Σ, and its corresponding repair in Σ'. Without loss of generality, we assume hereafter that $|\Sigma'| = |\Sigma|$ by allowing duplicate FDs in Σ'.

We define the distance between two sets of FDs as follows. For $\Sigma = \{X_1 \rightarrow A_1, \ldots, X_z \rightarrow A_z\}$ and $\Sigma' = \{Y_1X_1 \rightarrow A_1, \ldots, Y_zX_z \rightarrow A_z\}$, the term $\Delta_c(\Sigma, \Sigma')$ denotes vector $(Y_1, \ldots, Y_z)$, which consists of LHS extensions to FDs in Σ according to a repair Σ'. To measure the distance between Σ and Σ', we use the function $\sum_{Y \in \Delta_c(\Sigma, \Sigma'')} w(Y)$, where $w(Y)$ is a weighting function that determines the relative penalty of adding a set of attributes Y. The weighting function $w(.)$ is intuitively non-negative and monotone (i.e.,

for any two attribute sets X and Y, $X \subseteq Y$ implies that $w(X) \leq w(Y)$). A simple example of $w(Y)$ is the number of attributes in Y. However, this does not distinguish between attributes that have different characteristics. Other features of appended attributes can be used for obtaining other definitions of $w(.)$. For example, consider two attributes A and B that could be appended to the LHS of an FD, where A is a key (i.e., $A \rightarrow R$), while B is not. Intuitively, appending A should be more expensive that appending B because the FD resulting in the former case is trivially satisfied. In general, the more informative a set of attributes is, the more expensive it is when being appended to the LHS of an FD. The information captured by a set of attributes can be measured using various metrics, such as the number of distinct values of Y in I, and the entropy of Y. Another definition of $w(Y)$ could rely on the increase in the description length for modeling I using FDs due to appending Y [22, 58].

In general, $w(Y)$ depends on a given data instance to evaluate the weight of Y. Therefore, changing the cells in I during data cleaning might affect the weights of attributes. We make a simplifying assumption that $w(Y)$ depends only on the initial instance I. This assumption is based on an observation that the number of violations in I with respect to Σ is typically much smaller than the size of I, and thus repairing data does not significantly change the characteristics of attributes such as the entropy and the number of distinct values.

5.1.2 The Relative Trust in Data vs. FDs

We have defined a space of minimal repairs that covers a wide spectrum, ranging from repairs that only alter the data, while keeping the FDs unchanged, to repairs that only alter the FDs, while keeping the data unchanged. We propose a notion of relative trust between data and FDs to narrow down the space of desirable repairs and to steer the cleaning process towards a specific repair (or a range of repairs) in the described spectrum. The idea is to limit the maximum number of cell changes that can be performed while obtaining I' to a threshold τ, and to obtain a set of FDs Σ' that is the closest to Σ and is satisfied by I'. The obtained repair (Σ', I') is called a τ-constrained repair, formally defined as follows.

Definition 11. ***τ-constrained Repair*** *Given an instance I, a set of FDs Σ, and a threshold τ, a τ-constrained repair (Σ', I') is a repair in $\mathbf{U}$ such that $dist_d(I, I') \leq \tau$, and no other repair $(\Sigma'', I'') \in \mathbf{U}$ has $(dist_c(\Sigma, \Sigma''), dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), \tau)$.*

In other words, a τ-constrained repair is a repair in $\mathbf{U}$ whose distance to I is less than or equal to τ, and which has the minimum distance to Σ across all repairs in $\mathbf{U}$ that also have distance to I less than or equal to τ. We break ties using the distance to I (i.e., if two repairs have an equal distance to Σ and have distances to I less than or equal to τ, we choose the one closer to I).

Possible values of τ range from 0 to the minimum number of cells changes that must be applied to I in order to satisfy Σ, denoted $\delta_{opt}(\Sigma, I)$. We can also specify the threshold on the number of allowed cell changes as a percentage of $\delta_{opt}(\Sigma, I)$, denoted τ_r (i.e., $\tau_r = \tau / \delta_{opt}(\Sigma, I)$).

The parameter τ represents the relative trust in the data and the FDs. Setting τ to small values assumes that I is more trustworthy than Σ, and vice versa. Small values of τ enforce the cleaning algorithm to find a set of FDs Σ' that is almost satisfied by I. Thus, only a small number of changes suffice to find a data repair I' that satisfies Σ'. The opposite is true for large values of τ.

The mapping between minimal repairs and τ-constrained repairs is as follows. (1) Each τ-constrained repair is a minimal repair; (2) All minimal repairs can be found by varying threshold τ in the range $[0, \delta_{opt}(\Sigma, I)]$, and obtaining the corresponding τ-constrained repair. Specifically, each minimal repair (Σ', I') is equal to a τ-constrained repair, where τ is in the range defined as follows. Let (Σ'', I'') be the minimal repair with the smallest $dist_d(I, I'')$ that is strictly greater than $dist_d(I, I')$. If such a repair does not exist, let (Σ'', I'') be (ϕ, ϕ). The range of τ is defined as follows.

$$\tau \in \begin{cases} [dist_d(I, I'), dist_d(I, I'')) & \text{if } (\Sigma'', I'') \neq (\phi, \phi) \\ [dist_d(I, I'), \infty) & \text{if } (\Sigma'', I'') = (\phi, \phi) \end{cases} \tag{5.1}$$

If $(\Sigma'', I'') = (\phi, \phi)$, the range $[dist_d(I, I'), \infty)$ corresponds to a unique minimal repair where $dist_d(I, I')$ is equal to $\delta_{opt}(\Sigma, I)$. We prove these two points in the following theorem.

Theorem 5. *Each τ-constrained repair is a minimal repair. Each minimal repair (Σ', I') corresponds to a τ-constrained repair, where τ belongs to the range defined in Equation 5.1.*

Proof. In the following, we prove the first part of the theorem. Let (Σ', I') be a τ-constrained repair. It follows that no repair (Σ'', I'') has $(dist_c(\Sigma, \Sigma''), dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), \tau)$. Because $dist_d(I, I') \leq \tau$, there is no repair (Σ'', I'') satisfies $(dist_c(\Sigma, \Sigma''), dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), dist_d(I, I'))$. In other words, (Σ', I') is a minimal repair.

We prove the second part by contradiction. Assume that (Σ', I') is a minimal repair but it is not a τ-constrained repair for the values of τ described in Equation 5.1. Because $\tau \geq dist_d(I, I')$, and based on Definition 11, there must exist a repair (Σ_x, I_x) such that $(dist_c(\Sigma, \Sigma_x), dist_d(I, I_x)) \prec (dist_c(\Sigma, \Sigma'), \tau)$ (if multiple repairs satisfy this criteria, we select the repair with the minimum distance to I, and we break ties using the smaller distance to Σ). Repair (Σ_x, I_x) is a minimal repair because no other repair can dominate (Σ_x, I_x) with respect to distances to I and Σ. Because (Σ', I') is a minimal repair, then $dist_d(I, I_x) \geq dist_d(I, I')$ (otherwise, (Σ', I') would be dominated by (Σ_x, I_x)). However, existence of (Σ_x, I_x) contradicts the fact that no minimal repair exists with distance to I in the range $(dist_d(I, I'), \tau)$ (based on the value of τ obtained by Equation 5.1).

□

5.2 Holistic Cleaning of Data and FDs

There is a strong interplay between repairing data and repairing FDs. Obtaining a data instance that is closest to I, while satisfying a set of FDs Σ' highly depends on Σ'. Also, obtaining a set of FDs Σ' that is closest to Σ, such that Σ' holds in a given data instance I' highly depends on the instance I'. This interplay represents the main challenge for simultaneously repairing data and FDs in a way that achieves our three objectives: consistency of repair, minimality of changes, and adhering to the relative trust represented by threshold τ on the number of cell changes.

For example, consider a simple approach that alternates between editing the data and modifying the FDs until we reach consistency. This may not give a minimal repair (e.g., we might make a data change in one step that turns out to be redundant after we change one of the FDs in a subsequent step). Furthermore, such approach may have to make more than τ cell changes because it is difficult to predict the amount of necessary data changes while modifying the FDs.

We achieve the required objectives by dividing the cleaning process into two steps. In the first step, we *holistically* repair the entire set of FDs to obtain a modified FD set Σ' that is as close as possible to Σ, while guaranteeing that there exists a data repair I' satisfying Σ' with a distance to I less than or equal to τ. In the second step, we materialize the data instance I' by repairing I with respect to Σ' in a minimal way. We describe this approach in Algorithm 8.

Finding Σ' in the first step requires computing the minimum number of cell changes in I to satisfy Σ' (i.e., $\delta_{opt}(\Sigma', I)$) before the actual cleaning takes place. Note that computing $\delta_{opt}(\Sigma', I)$ does not require materialization of an optimum repair. Instead, we perform *speculative data cleaning* by collecting enough statistics about the violations in data to compute $\delta_{opt}(\Sigma', I)$. More details are provided in Section 5.4.

Algorithm 8 `Repair_Data_FDs(`Σ, I, τ`)`

1: obtain Σ' from $\mathcal{S}(\Sigma)$ such that $\delta_{opt}(\Sigma', I) \leq \tau$, and no other $\Sigma'' \in \mathcal{S}(\Sigma)$ with $\delta_{opt}(\Sigma'', I) \leq \tau$ has $dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma')$. (ties are broken using $\delta_{opt}(\Sigma', I)$)
2: **if** $\Sigma' \neq \phi$ **then**
3: obtain I' that satisfies Σ' while performing at most $\delta_{opt}(\Sigma', I)$ cell changes, and return (Σ', I').
4: **else**
5: Return (ϕ, ϕ)
6: **end if**

The following theorem establishes the link between the repairs generated by Algorithm 8 and Definition 11.

Theorem 6. *Repairs generated by Algorithm 8 are τ-constrained repairs.*

Proof. For a generated repair (Σ', I'), the condition $dist_d(I, I') \leq \tau$ holds due to the constraint $\delta_{opt}(\Sigma', I) \leq \tau$ in line 1. For any $\Sigma'' \in \mathcal{S}(\Sigma)$, $\delta_{opt}(\Sigma'', I) \leq dist_d(I, I'')$ for all $I'' \models \Sigma''$, and thus the condition $dist_d(I, I'') \leq \tau$ implies that $\delta_{opt}(\Sigma'', I) \leq \tau$. Therefore the condition $\nexists \Sigma'' \in \mathcal{S}(\Sigma)(\delta_{opt}(\Sigma'', I) \leq \tau \wedge dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma'))$ in line 1, along with the tie breaking mechanism, imply that $\nexists (\Sigma'', I'') \in \mathbf{U}$ $(dist_c(\Sigma, \Sigma''), dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), \tau)$. Thus, (Σ', I') is a τ-constrained repair. □

A key step in Algorithm 8 is computing $\delta_{opt}(\Sigma', I)$ (i.e., the minimum number of cells in I that have to be changed in order to satisfy Σ'). Unfortunately, computing the exact minimum number of cell changes when Σ' contains at least two FDs is NP-hard [57]. We will propose an approximate solution based on upper-bounding the minimum number of necessary cell changes. Assume that there exists a P-approximate upper bound on $\delta_{opt}(\Sigma', I)$, denoted $\delta_P(\Sigma', I)$ (details are in Section 5.4). That is, $\delta_{opt}(\Sigma', I) \leq \delta_P(\Sigma', I) \leq P \cdot \delta_{opt}(\Sigma', I)$, for some constant P. By using $\delta_P(\Sigma', I)$ in place of $\delta_{opt}(\Sigma', I)$ in Algorithm 8, we can satisfy the criteria in Definition 11 in a P-approximate way. Specifically, the repair generated by Algorithm 8 becomes a P-approximate τ-constrained repair, which is defined as follows (the proof is similar to Theorem 6).

Definition 12. ***P-approximate τ-constrained Repair*** *Given an instance I, a set of FDs Σ, and a threshold τ, a P-approximate τ-constrained repair (Σ', I') is a repair in $\mathbf{U}$ such that $dist_d(I, I') \leq \tau$, and no other repair $(\Sigma'', I'') \in \mathbf{U}$ has $(dist_c(\Sigma, \Sigma''), P \cdot dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), \tau)$.*

In the remainder of this paper, we present an implementation of line 1 (Section 5.3) and line 3 (Section 5.4) of Algorithm 8. Our implementation is P-approximate, as defined above, with $P = 2 \cdot \min\{|R| - 1, |\Sigma|\}$, where $|R|$ denotes the number of attributes in relation R, and $|\Sigma|$ denotes the number of FDs in Σ.

5.3 Holistic Repairing of FDs

In this section, we show how to obtain a modified set of FDs Σ' that is part of a P-approximate τ-constrained repair (line 1 of Algorithm 8). That is, we need to obtain

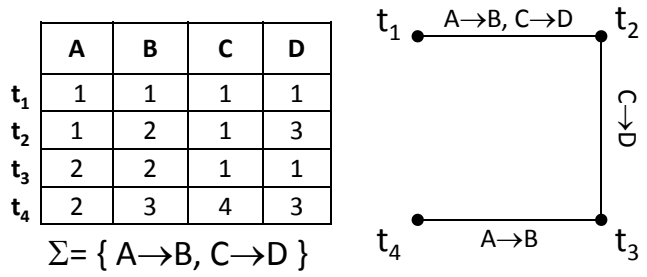

Figure 5.1: An example of a conflict graph

$\Sigma' \in \mathcal{S}(\Sigma)$ such that $\delta_P(\Sigma', I) \leq \tau$, and no other FD set $\Sigma'' \in \mathcal{S}(\Sigma)$ with $\delta_P(\Sigma'', I) \leq \tau$ has $dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma')$.

First, we need to introduce the notion of a conflict graph of I with respect to a set of FDs Σ, which has previously been used in [8]:

Definition 13. ***Conflict Graph.*** *A conflict graph of an instance I and a set of FDs Σ is an undirected graph whose set of vertices is the set of tuples in I, and whose set of edges consists of all edges (t_i, t_j) such that t_i and t_j violate at least one FD in Σ.*

Figure 5.1 shows an instance I, a set of FDs Σ, and the corresponding conflict graph. The label of each edge represents the FDs that are violated by the edge vertices.

In Section 5.4, we present an algorithm to obtain an instance repair I' that satisfies a set of FDs $\Sigma' \in \mathcal{S}(\Sigma)$. The number of cell changes performed by our algorithm is linked to the conflict graph of Σ' and I as follows. Let $C_{2opt}(\Sigma', I)$ be a 2-approximate minimum vertex cover of the conflict graph of Σ' and I, which we can obtain in PTIME using a greedy algorithm [44]. The number of cell changes performed by our algorithm is at most $\alpha \cdot |C_{2opt}(\Sigma', I)|$, where $\alpha = \min\{|R| - 1, |\Sigma|\}$. Moreover, we prove that the number of changed cells is 2α-approximately minimal. Therefore, we define $\delta_P(\Sigma', I)$ as $\alpha \cdot |C_{2opt}(\Sigma', I)|$, which represents a 2α-approximate upper bound of $\delta_{opt}(\Sigma', I)$ that can be computed in PTIME. Based on the definition of $\delta_P(\Sigma', I)$, our goal in this section can be rewritten as follows: obtain $\Sigma' \in \mathcal{S}(\Sigma)$ such that $C_{2opt}(\Sigma', I) \leq \frac{\tau}{\alpha}$, and no other FD set $\Sigma'' \in \mathcal{S}(\Sigma)$ with $C_{2opt}(\Sigma'', I) \leq \frac{\tau}{\alpha}$ has $dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma')$.

Figure 5.2 depicts several possible repairs of Σ from Figure 5.1, along with $dist_c(\Sigma, \Sigma')$

Σ'	$\text{dist}_c(\Sigma, \Sigma')$	Conflict Graph Edges	$C_{2opt}(\Sigma', I)$	$\delta_p(\Sigma', I)$
A→B, C→D	0	(t_1,t_2), (t_2,t_3), (t_3,t_4)	t_2, t_3	4
CA→B, C→D	1	(t_1,t_2), (t_2,t_3)	t_2	2
DA→B, C→D	1	(t_1,t_2), (t_2,t_3)	t_2	2
A→B, **AC**→D	1	(t_1,t_2), (t_3,t_4)	t_1, t_3	4
A→B, **BC**→D	1	(t_1,t_2), (t_2,t_3), (t_3,t_4)	t_2, t_3	4
CA→B, **AC**→D	2	(t_1,t_2)	t_1	2
...		...	...	...

Figure 5.2: An example of multiple FD repairs

(assuming that the weighting function $w(Y)$ is equal to $|Y|$), the corresponding conflict graph, $C_{2opt}(\Sigma', I)$, and $\delta_P(\Sigma', I)$. For $\tau = 2$, repairs of Σ that are part of P-approximate τ-constrained repairs are $\{CA \rightarrow B, C \rightarrow D\}$ and $\{DA \rightarrow B, C \rightarrow D\}$.

5.3.1 Searching the Space of FD Repairs

We model the possible FD repairs $\mathcal{S}(\Sigma)$ as a state space, where for each $\Sigma' \in \mathcal{S}(\Sigma)$, there exists a state representing $\Delta_c(\Sigma, \Sigma')$ (i.e., the vector of attribute sets appended to LHSs of FDs to obtain Σ'). Additionally, we call $\Delta_c(\Sigma, \Sigma')$ a *goal state* iff $\delta_P(\Sigma', I) \leq \tau$, for a given threshold value τ (or equivalently, $C_{2opt}(\Sigma', I) \leq \frac{\tau}{\alpha}$). The cost of a state $\Delta_c(\Sigma, \Sigma')$ is equal to $dist_c(\Sigma, \Sigma')$. We assume that the weighting function $w(.)$ is monotone and non-negative. Our goal is to locate the cheapest goal state for a given value of τ, which amounts to finding an FD set Σ' that is part of a P-approximate τ-constrained repair.

The monotonicity of the weighting function w (and hence the monotonicity of the overall cost function) allows for pruning a large part of the state space. We say that a state $(Y_1, \ldots, Y_z)$ *extends* another state $(Y'_1, \ldots, Y'_z)$, where $z = |\Sigma|$, iff for all $i \in \{1, \ldots, z\}$, $Y'_i \subseteq Y_i$. Clearly, if $(Y_1, \ldots, Y_z)$ is a goal state, we can prune all the FD sets that extend it because $w(.)$ is monotone.

In Figure 5.3(a), we show all the states for $R = \{A, B, C, D, E, F\}$ and $\Sigma = \{A \rightarrow F\}$. Each arrow in Figure 5.3(a) indicates that the destination state extends the source state

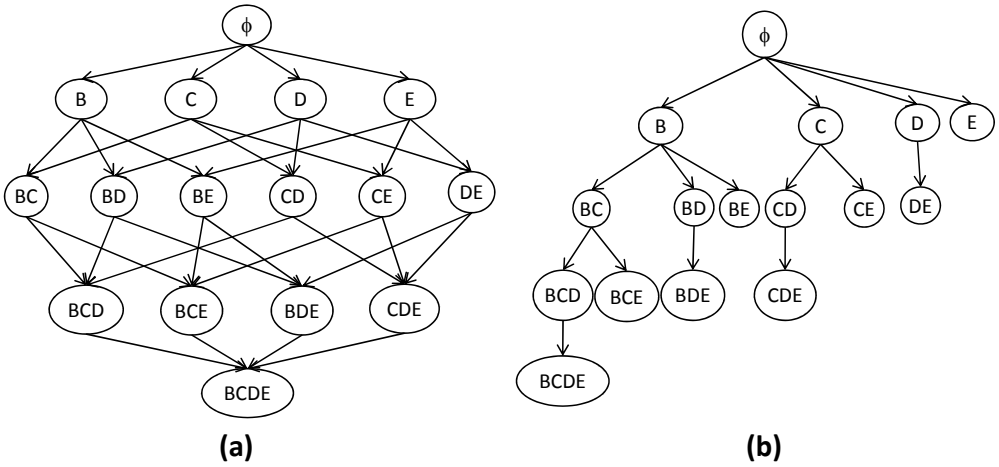

Figure 5.3: The state search space for $R = \{A, B, C, D, E, F\}$ and $\Sigma = \{A \rightarrow F\}$ (a) a graph search space (b) a tree search space

by adding exactly one attribute. We can find the cheapest goal state by traversing the graph in Figure 5.3(a). For example, we can use a level-wise breadth-first search strategy [66], which iterates over states with the same number of attributes, and, for each such set of states, we determine whether any state is a goal state. If one or more goal states are found at the current level, we return the cheapest goal state and terminate the search.

We can optimize the search by adopting best-first traversal of the states graph [66]. That is, we maintain a list of states to be visited next, called the *open list*, which initially contains the state $(\phi, \ldots, \phi)$, and a list of states that have been visited, called the *closed list*. In each iteration, we pick the cheapest state S from the open list, and test whether S is a goal state. If S is a goal state, we return it and terminate the search. Otherwise, we add S to the closed list, and we insert into the open list all the states that extend S by exactly one attribute and are not in the closed list.

We can avoid using a closed list that keeps track of visited states, and hence reduce the running time, by ensuring that each state can only be reached from the initial state $(\phi, \ldots, \phi)$ using a unique path. In other words, we need to reduce the graph in Figure 5.3(a) to a tree (e.g., Figure 5.3(b)). To achieve this goal, we assign each state, except $(\phi, \ldots, \phi)$, to a single parent. Assume that attributes in R are totally ordered (e.g., lexicographically).

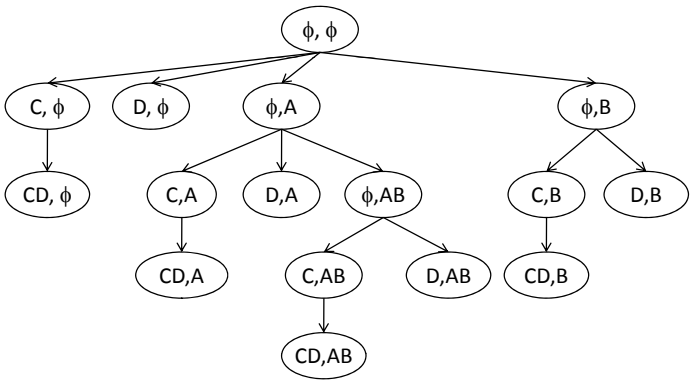

Figure 5.4: A tree search space for $R = \{A, B, C, D\}$ and $\Sigma = \{A \rightarrow B, C \rightarrow D\}$

For Σ with a single FD, the parent of a state Y is another state $Y \setminus \{A\}$ where A is the greatest attribute in Y. Figure 5.3(b) shows the search tree that is equivalent to the search graph in Figure 5.3(a). In general, when Σ contains multiple FDs, the parent of a state $(Y_1, \ldots, Y_z)$ is determined as follows. Let A be the greatest attribute in $\bigcup_{i=1}^{z} Y_i$, and j be the index of the last element in the vector $(Y_1, \ldots, Y_z)$ that contains A. The parent of the state $(Y_1, \ldots, Y_z)$ is another state $(Y_1, \ldots, Y_{j-1}, Y_j \setminus \{A\}, Y_{j+1}, \ldots, Y_z)$. Figure 5.4 depicts an example search space for the two FDs shown in Figure 5.1.

5.3.2 A*-based Search Algorithm

One problem with best-first tree traversal is that it might visit cheap states that only lead to expensive goal states or no goal states at all. A* search [66] avoids such pitfall by estimating the cost of the cheapest goal state reachable (i.e., descending) from each state S in the open list, denoted $\underline{gc}(S)$, and visiting the state with the smallest $\underline{gc}(S)$ first. In order to maintain soundness of the algorithm (i.e., returning the cheapest goal state), we must not overestimate the cost of the cheapest goal state reachable from a state S [66].

Algorithm 9 describes the search procedure. The goal of lines 1 and 12-16, along with the sub-procedure `getDescGoalStates`, is computing $\underline{gc}(S)$. The reminder of Algorithm 9 follows the A* search algorithm: it initializes an open list, which is implemented as a priority queue called PQ, by inserting the root state $(\phi, \ldots, \phi)$. In each iteration, the

Algorithm 9 `Repair_FDs`(Σ, I, τ)

construct the conflict graph G of Σ and I, and obtain the set of all difference sets in G, denoted $\mathcal{D}$
$PQ \leftarrow \{(\phi, \ldots, \phi)\}$
while PQ is not empty **do**
  pick the state S_h with the smallest value of $\underline{\text{gc}}(.)$ from PQ
  let Σ_h be the FD set corresponding to S_h
  Compute $C_{2opt}(\Sigma_h, I)$
  if $|C_{2opt}(\Sigma_h, I)| \cdot \min\{|R| - 1, |\Sigma|\} \leq \tau$ **then**
    return Σ_h
  end if
  remove S_h from PQ
  for each state S_i that is a child of S_h **do**
    let Σ_i be the FD set corresponding to S_i
    let $\mathcal{D}_s$ be the subset of difference sets in $\mathcal{D}$ that violate Σ_i
    let G_0 be an empty graph
    $minStates \leftarrow$ `getDescGoalStates`$(S_i, S_i, G_0, \mathcal{D}_s, \tau)$
    set $\underline{\text{gc}}(S_i)$ to the minimum cost across all states in $minStates$, or ∞ if $minStates$ is empty
    if $\underline{\text{gc}}(S_i)$ is not ∞ **then**
      insert S_i into PQ
    end if
  end for
end while
return ϕ

algorithm removes the state with the smallest value of $\underline{gc}(S)$ from PQ and checks whether it is a goal state. If so, the algorithm returns the corresponding FD set. Otherwise, the algorithm inserts the children of the removed state into PQ, after computing $\underline{gc}(.)$ for each inserted state.

The two technical challenges of computing $\underline{gc}(S)$ are the tightness of the bound $\underline{gc}(S)$ (i.e., being close to the actual cost of the cheapest goal state descending from S), and having a small computational cost. In the following, we describe how we address these challenges.

Given a conflict graph G of I and Σ, each edge represents two tuples in I that violate Σ. For any edge (t_i, t_j) in G, we refer to the attributes that have different values in t_i and t_j as the *difference set* of (t_i, t_j). Difference sets have been introduced in the context of FD discovery (e.g., [61, 81]). For example, the difference sets for (t_1, t_2), (t_2, t_3), and (t_3, t_4) in Figure 5.1 are BD, AD, and BCD, respectively. We denote by $\mathcal{D}$ the set of all difference sets for edges in G (line 1 in Algorithm 9). The key insight that allows efficient computation of $\underline{gc}(S)$ is that all edges (i.e., violations) in G with the same difference set can be completely resolved by adding one attribute from the difference set to the LHS of each violated FD in Σ. For example, edges corresponding to difference set BD in Figure 5.1 violate both $A \rightarrow B$ and $C \rightarrow D$, and to fix such violations, we need to add D to the LHS of the first FD, and B to the LHS of the second FD. Similarly, fixing violations corresponding to difference set BCD can be done by adding C or D to the first FD (second FD is not violated). Therefore, we partition the edges of the conflict graph G based on their difference sets. In order to compute $\underline{gc}(S)$, each group of edges corresponding to one difference set is considered atomically, rather than individually.

Let $\mathcal{D}_s$ be a *subset* of difference sets that are still violated at the current state S_i (line 13). Given a set of difference sets $\mathcal{D}_s$, the recursive procedure `getDescGoalStates`$(S, S_c, G_c, \mathcal{D}_c, \tau)$ (Algorithm 10) finds all minimal goal states descending from S that resolve $\mathcal{D}_c$, taking into consideration the maximum number of allowed cell changes τ. Therefore, $\underline{gc}(S)$ can be assigned to the cheapest state returned by the procedure `getDescGoalStates`. Note that we use a subset of difference sets that are still violated ($\mathcal{D}_s$), instead of using all violated difference sets, in order to efficiently compute $\underline{gc}(S)$. The computed value of $\underline{gc}(S)$ is clearly a lower bound on the cost of actual cheap-

est goal state descending from the current state S. To provide tight lower bounds, $\mathcal{D}_s$ is selected such that difference sets corresponding to large numbers of edges are favored. Additionally, we heuristically ensure that the difference sets in $\mathcal{D}_s$ have a small overlap.

We now describe Algorithm 10. It recursively selects a difference set d from the set of non-resolved difference sets $\mathcal{D}_c$. For each difference set d, we consider two alternatives: (1) excluding d from being resolved, if threshold τ permits, and (2) resolving d by extending the current state S_c. In the latter case, we consider all possible children of S_c to resolve d. Once S_c is extended to S'_c, we remove from $\mathcal{D}_c$ all the sets that are now resolved, resulting in $\mathcal{D}'_c$. Due to the monotonicity of the cost function, we can prune all the non-minimal states from the found set of states. That is, if state S_1 extends another state S_2 and both are goal states, we remove S_1.

In the following lemma, we prove that the computed value of $\underline{\text{gc}}(S)$ is a lower bound on the cost of the cheapest goal descending from state S.

Lemma 3. *For any state S, the computed value of $\underline{gc}(S)$ is less than or equal to the cost of the cheapest goal state that is a descendant of state S.*

Proof. Let Σ be the set of FDs corresponding to S. Assume that we are using the entire set of difference sets, denoted $\mathcal{D}_{all}$, that violate Σ rather than using a subset of difference sets (line 13 in Algorithm 9).

The cheapest goal state S_g that are a descendent of S will be among the states returned by the procedure `getDescGoalStates` because the procedure `getDescGoalStates` returns all minimal goal states (if any), and S_g is minimal (i.e., there exist no other state S' such that S_g extends S' and S' is a goal state).

Because we are using a subset of all difference sets $\mathcal{D}_{all}$, the cost of the reported cheapest goal state is less than or equal to the actual cost of the cheapest goal state.

□

Based on Lemma 3, and the correctness of the A* search algorithm [66], we conclude that the FD set generated by Algorithm 9 is part of a P-approximate τ-constrained repair.

Algorithm 10 `getDescGoalStates`$(S, S_c, G_c, \mathcal{D}_c, \tau)$

Require: S : the state for which we compute $\underline{gc}(.)$

Require: S_c : the current state to be extended (equals S at the first entry)

Require: G_c : the current conflict graph for non-resolved difference sets (is empty at the first entry)

Require: $\mathcal{D}_c$: the remaining difference sets to be resolved

1: **if** $\mathcal{D}_c$ is empty **then**
2: **return** $\{S_c\}$
3: **end if**
4: $States \leftarrow \phi$
5: select a difference set d from $\mathcal{D}_c$
6: let G'_c be the graph whose edges are the union of edges corresponding to d and edges of G_c
7: compute a 2-approximate minimum vertex cover of G'_c, denoted C_{2opt}
8: **if** $|C_{2opt}| \cdot \min\{|R| - 1, |\Sigma|\} < \tau$ **then**
9: $\mathcal{D}'_c \leftarrow \mathcal{D}_c \setminus \{d\}$
10: $States \leftarrow States \cup$ `getDescGoalStates`$(S, S_c, G'_c, \mathcal{D}'_c, \tau)$
11: **end if**
12: **for** each possible state S'_c that extends S_c, is descendant of S, and resolves violations corresponding to d **do**
13: let $\mathcal{D}'_c$ be all difference sets in $\mathcal{D}_c$ that are still violating Σ'_c that is corresponding to S'_c
14: $States \leftarrow States \cup$ `getDescGoalStates`$(S, S'_c, G_c, \mathcal{D}'_c, \tau)$
15: **end for**
16: remove any non-minimal states from $States$
17: **return** $States$

In the following, we investigate the complexity of Algorithms 9 and 10. Finding all difference sets in line 1 in Algorithms 9 is performed in $O(|\Sigma| \cdot n + |\Sigma| \cdot |E| + |R| \cdot |E|)$, where n denotes the number of tuples in I, and E denotes the number of edges in the conflict graph of I and Σ. Difference sets are obtained by building the conflict graph of I and Σ, which costs $O(|\Sigma| \cdot n + |\Sigma| \cdot |E|)$ (more details are in Section 5.4), and then computing the difference set for all edges, which costs $O(|R| \cdot |E|)$. In worst case, Algorithm 9, which is based on A* search, will visit a number of states that is exponential in the depth of the cheapest goal state [66], which is less than $|\Sigma| \cdot (|R| - 2)$. However, the number of states visited by an A* search algorithm is the minimum across all algorithms that traverse the same search tree and use the same heuristic for computing $\underline{gc}(S)$. Also, we show in our experiments that the actual number of visited states is much smaller than the best-first search algorithm (Section 5.6).

The worst-case complexity of Algorithm 10 that finds $\underline{gc}(S)$ is $O(|E| \cdot |R|^{|\Sigma| \cdot |\mathcal{D}_c|})$, where $|\mathcal{D}_c|$ is the number of difference sets passed to the algorithm. This is due to recursively inspecting each difference set in $\mathcal{D}_c$ and, if not already resolved by the current state S_c, appending one more attribute from the difference set to the LHS of each FD. At each step, approximate vertex graph cover might need to be computed, which can be performed in $O(|E|)$.

5.3.3 Improving the Efficiency of the State Search

We can reduce the computational cost of searching for an FD repair by modifying Algorithm 9 to obtain a near-optimal goal state, based on weighted A* algorithm [66],

Recall that each state S_i is associated with a non-overestimating cost of the cheapest goal state descending from S_i, denoted $\underline{gc}(S_i)$. The value of $\underline{gc}(S_i)$ can be decomposed into two components: the cost of the state S_i, denoted $c(S_i)$, and a non-overestimating cost to reach the cheapest goal state descending from S_i starting from S_i, denoted $h(S_i)$. The value of $\underline{gc}(S_i)$ is equal $c(S_i) + h(S_i)$, and thus we can obtain $h(S_i)$ by computing $\underline{gc}(S_i)$ and $c(S_i)$.

To speed up the search algorithm, we need to provide tighter cost bounds, represented by $\underline{gc}(S_i)$, which in turn reduces the number of visited states. This can be achieved by

multiplying the component $h(S_i)$ by a constant factor $Q > 1$. Note that we do not multiply $c(S_i)$ by Q because $c(S_i)$ represents the exact cost of S_i, which cannot be underestimated. Formally, we use the following estimate to determine which state to visit next: $\hat{gc}(S_i) = c(S_i) + Q(\underline{gc}(S_i) - c(S_i))$. Because we can only overestimate the minimum cost to reach a goal state by at most a factor equal to Q, the resulting FD set Σ' is guaranteed to be Q-approximate minimal with respect to the distance to Σ. That is, $\nexists \Sigma'' \in \mathcal{S}(\Sigma)$ such that $P \cdot \delta(\Sigma'', I) \leq \tau$ and $Q \cdot dist_c(\Sigma, \Sigma'') < dist_c(\Sigma, \Sigma')$. We call the resulting repair P-Q approximate τ-constrained repair, which is defined as follows.

Definition 14. ***P-Q-approximate τ-constrained Repair*** *Given an instance I, a set of FDs Σ, and a threshold τ, a P-Q-approximate τ-constrained repair (Σ', I') is a repair in $\mathbf{U}$ such that $dist_d(I, I') \leq \tau$, and no other repair $(\Sigma'', I'') \in \mathbf{U}$ has $(Q \cdot dist_c(\Sigma, \Sigma''), P \cdot dist_d(I, I'')) \prec (dist_c(\Sigma, \Sigma'), \tau)$.*

5.4 Near-Optimal Data Cleaning

In this section, we derive a P-approximation of $\delta_{opt}(\Sigma', I)$, denoted $\delta_P(\Sigma', I)$, in terms of the conflict graph of I and Σ', where P is equal to $2 \cdot \min\{|R| - 1, |\Sigma|\}$. Also, we provide a data cleaning algorithm that makes at most $\delta_P(\Sigma', I)$ cell changes.

There are several data cleaning algorithms that obtain a data repair, given that the set of FDs is fixed (i.e., completely trusted), such as [19, 27, 57]. Most approaches do not provide any bounds on the number of cells that are changed during the repairing process. In [57], the proposed algorithm provides an upper bound on the number of cell changes and it is proved to be near-minimum. The approximation factor depends on the set of FDs Σ, which is assumed to be fixed. Unfortunately, we need to deal with multiple FD sets, and the approximation factor described in [57] can grow arbitrarily while modifying the initial FD set. That is, the approximation factors for two possible repairs Σ', Σ'' in $\mathcal{S}(\Sigma)$ can be different. In this section, we provide a method to compute $\delta_P(\Sigma', I)$ such that the approximation factor is equal to $2 \cdot \min\{|R| - 1, |\Sigma|\}$, which depends only on the number of attributes in R and the number of FDs in Σ .

The output of our data cleaning algorithm is a V-instance, which have been first introduced in [57] to concisely represent multiple data instances (refer to Section 4.2.1 for more details). In the remainder of this chapter, we refer to a V-instance as simply an instance.

The algorithm we propose in this section is considered a variant of the data cleaning algorithms we proposed in Chapter 4. The main difference is that, in this section, we clean the data tuple-by-tuple instead of the cell-by-cell cleaning approach. That is, we first identify a set of clean tuples that satisfy Σ' such that the cardinality of the set is approximately maximum. We convert this problem to the problem of finding the minimum vertex cover, and we use a greedy algorithm with an approximation factor of 2. Then, we iteratively modify the unclean tuples as follows. For each unclean tuple t, we iterate over attributes of t in a random order, and we modify each attribute, if necessary, to ensure that the attributes processed so far are clean. in the remainder of this section, we provide a detailed discussion of the cleaning procedure.

Given a set of FDs Σ', the procedure `Repair_Data` in Algorithm 11 generates an instance I' that satisfies Σ'. Initially, the algorithm constructs the conflict graph of I and Σ'. Then, the algorithm obtains a 2-approximate minimum vertex cover of the obtained conflict graph, denoted $C_{2opt}(\Sigma', I)$, using a greedy approach described in [44] (for brevity, we refer to $C_{2opt}(\Sigma', I)$ as C_{2opt} in this section). The clean instance I' is initially set to I. The algorithm repeatedly removes a tuple t from C_{2opt}, and it changes attributes of t to ensure that, for every tuple $t' \in I' \setminus C_{2opt}$, t and t' do not violate Σ' (lines 5-15). This is achieved by repeatedly picking an attribute of t at random, and adding it to a set denoted $Fixed\_Attrs$ (line 9). After inserting an attribute A, we determine whether we can find an assignment to the attributes outside $Fixed\_Attrs$ such (t, t') are not violating Σ', for all $t' \in I' \setminus C_{2opt}$. We use Algorithm 12 to find a valid assignment, if any, or to indicate that no valid assignment exists. Note that when $Fixed\_Attrs$ contains only one attribute (line 6), it is guaranteed that a valid assignment exists (line 7). If a valid assignment is found, we keep $t[A]$ unchanged. Otherwise, we change $t[A]$ to the value of attribute A of the valid assignment found in the previous iteration (line 11). The algorithm proceeds until all tuples have been removed from C_{2opt}. We return I' upon termination.

Algorithm 12 searches for an assignment to attributes of a tuple t that are not in $Fixed\_Attrs$ such that every pair (t, t') satisfies Σ' for all $t' \in I' \setminus C_{2opt}$. An initial assign-

Algorithm 11 `Repair_Data`(Σ', I)

let G be the conflict graph of I and Σ'
obtain a 2-approximate minimum vertex cover of G, denoted C_{2opt}
$I' \leftarrow I$
while C_{2opt} is not empty **do**
  randomly pick a tuple t from C_{2opt}
  $Fixed\_Attrs \leftarrow \{A\}$, where A is a randomly picked attribute from R
  $t_c \leftarrow$ `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt})$
  while $|Fixed\_Attrs| < |R|$ **do**
    randomly pick an attribute A from $R \setminus Fixed\_Atts$ and insert it into $Fixed\_Attrs$
    if `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt}) = \phi$ **then**
      $t[A] \leftarrow t_c[A]$
    else
      $t_c \leftarrow$ `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt})$
    end if
  end while
  remove t from C_{2opt}
end while
return I'

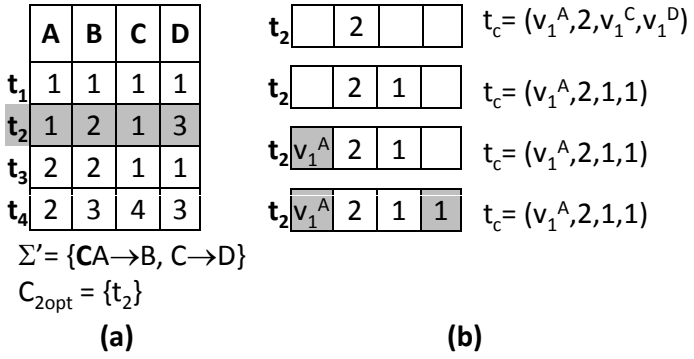

Figure 5.5: An example of repairing data: (a) initial value of I', Σ' and C_{2opt} (b) steps of fixing the tuple t_2

Algorithm 12 `Find_Assignment`$(t, Fixed\_Attrs, I', \Sigma', C_{2opt})$

construct a tuple t_c such that $t_c[A] = t[A]$ if $A \in Fixed\_Attrs$, and $t_c[A] = v_i^A$ if $A \notin Fixed\_Attrs$, where v_i^A is a new variable
while $\exists t' \in I' \setminus C_{2opt}$ such that for some FD $X \rightarrow A \in \Sigma'$, $t_c[X] = t'[X] \wedge t_c[A] \neq t'[A]$ **do**
  if $A \in Fixed\_Attrs$ **then**
    return ϕ
  else
    $t_c[A] \leftarrow t'[A]$
    add A to $Fixed\_Attrs$
  end if
end while
return t_c

ment t_c is created by setting attributes that are in $Fixed\_Attrs$ to be equal to t, and setting attributes that are not in $Fixed\_Attrs$ to new variables. The algorithm repeatedly selects a tuple $t' \in I' \setminus C_{2opt}$ such that (t, t') violates an FD $X \rightarrow A \in \Sigma'$. If attribute A belongs to $Fixed\_Attrs$, the algorithm returns ϕ, indicating that no valid assignment is available. Otherwise, the algorithm sets $t[A]$ to be equal to $t'[A]$, and adds A to $Fixed\_Attrs$. When no other violations could be found, the algorithm returns the assignment t_c.

In Figure 5.5, we show an example of generating a data repair for $\Sigma' = \{CA \rightarrow B, C \rightarrow D\}$, given the instance I shown in Figure 5.5(a). After adding the first attribute B to $Fixed\_Attrs$, the current valid assignment, denoted t_c, is equal to $(v_1^A, 2, v_1^C, v_1^D)$. When inserting C to $Fixed\_Attrs$, there is no need to change the value of C because we can find a valid assignment to the remaining attributes, which is $(v_1^A, 2, 1, 1)$. After inserting A to $Fixed\_Attrs$, no valid assignment is found, and thus we set $t[A]$ to the value of attribute A of the previous valid assignment t_c. Similarly, we set $t[D]$ to $t_c[D]$ after inserting D into $Fixed\_Attrs$. The resulting instance satisfies Σ'.

The following lemma proves the soundness and completeness of Algorithm 12.

Lemma 4. *Algorithm 12 is both sound (i.e., the obtained assignments are valid) and*

complete (it will return an assignment if a valid assignment exists).

Proof. We first prove the soundness of the algorithm. That is, we need to prove that if a tuple t_c is returned, $t_c[A] = t[A]$, for $A \in Fixed\_Attrs$, and for all $t' \in I' \setminus C_{2opt}$, (t_c, t') do not violate Σ.

From the algorithm description, it is clear that the condition $t_c[A] = t[A]$ holds, for $A \in Fixed\_Attrs$. Also, the condition in line 2 ensures that whenever a tuple t_c is returned, there does not exist $t' \in I' \setminus C_{2opt}$ such that (t_c, t') violate any FD in Σ.

We prove the completeness of the algorithm by contradiction. Assume that Algorithm 12 returns ϕ, while there exist a tuple t_g that satisfies the conditions $t_g[A] = t[A]$, for $A \in Fixed\_Attrs$, and (t_g, t') satisfy Σ, for all $t' \in I' \setminus C_{2opt}$.

We first show that, just before returning ϕ at line 4, all attributes in $Fixed\_Attrs$ have equal values in tuples t_c and t_g. This is clearly true for the initial value of $Fixed\_Attrs$. Let A be the attribute that is first inserted in $Fixed\_Attrs$ in line 7. Before setting $t_c[A]$ to $t'[A]$ in line 6, there exist a tuple $t' \in I' \setminus C_{2opt}$ such that (t', t_c) violate an FD $X \rightarrow A \in \Sigma$. Attributes in X belong to $Fixed\_Attrs$ because attributes outside $Fixed\_Attrs$ are assigned to new variables (line 1) and cannot be equal to attributes of any other tuples. It follows that attribute A in any valid solution must be equal to $t'[A]$ in order to satisfy Σ. Thus, $t_c[A] = t_g[A] = t'[A]$. The same argument is valid for attributes that are successively inserted into $Fixed\_Attrs$ before returning ϕ. When the algorithm returns ϕ (line 4), there exists a tuple $t' \in I' \setminus C_{2opt}$ such that (t', t_c) violate an FD $X \rightarrow A \in \Sigma$ and attribute A belongs to $Fixed\_Attrs$. Because $AX \subset Fixed\_Attrs$, and attributes in $Fixed\_Attrs$ have equal values in t_c and t_g, it follows that (t', t_g) violate $X \rightarrow A$ as well (i.e., t_g is not a valid answer), which contradicts our initial assumption.

□

The following theorem proves the P-optimality of Algorithm 11.

Theorem 7. *For a given instance I and a set of FDs $\Sigma' \in \mathcal{S}(\Sigma)$, Algorithm* `Repair_Data`$(\Sigma', I)$ *obtains an instance $I' \models \Sigma'$ such that the number of changed cells in I' is at most $|C_{2opt}(\Sigma', I)| \cdot \min\{|R| - 1, |\Sigma|\}$, and it is $2 \cdot \min\{|R| - 1, |\Sigma|\}$-approximate minimum.*

Proof. We first prove that the returned I' satisfies Σ'. Let G be the conflict graph of I with respect to Σ' and let C_{2opt} be a 2-approximate minimum vertex cover of G that is obtained at line 2 in Algorithm 11. The tuple set $I \setminus C_{2opt}$ satisfies Σ', and thus the corresponding tuples in I' satisfy Σ' as well. For each tuple t that is randomly picked from C_{2opt} in line 5 in Algorithm 11, modifying t as described in lines 6-15 makes the set $I' \setminus C_{2opt} \cup \{t\}$ satisfies Σ', as we show in the following. We observe that for a $Fixed\_Attrs$ containing a single attribute A, there exists an assignment to the attributes $R \setminus \{A\}$ in t such that $I' \setminus C_{2opt} \cup \{t\}$ satisfies Σ' (i.e., t_c cannot be ϕ at line 7). We describe one possible assignment as follows. If the value of $t[A]$ does not appear in attribute A of any tuple in $I' \setminus C_{2opt}$, then setting attributes $R \setminus \{A\}$ to new variables is a valid assignment. Otherwise, let t_r be a tuple in $I' \setminus C_{2opt}$ such that $t[A] = t_r[A]$. Setting attributes $R \setminus \{A\}$ in t to the values of corresponding attributes in t_r is a valid assignment. Thus, t_c cannot be ϕ in line 7 due to completeness of Algorithm 12, which is proved in Lemma 4.

After each iteration of the while loop in line 8, Algorithm 11 maintains a tuple t_c such that current attributes in $Fixed\_Attrs$ have equal values in t_c and the current version of t, and other attributes outside $Fixed\_Attrs$ in t_c are assigned to values that make $I' \setminus C_{2opt} \cup \{t_c\}$ satisfies Σ' (due to soundness of Algorithm 12 as proved in Lemma 4). After inserting all attributes in $Fixed\_Attrs$, t is equal to t_c and thus $I' \setminus C_{2opt} \cup \{t\}$ satisfies Σ'. After processing, and removing, all tuples from C_{2opt}, the resulting instance I' satisfies Σ'.

We prove the approximate optimality of the algorithm as follows. Let C_{opt} be a minimum vertex cover of G. The minimum number of cell changes $\delta_{opt}(\Sigma', I)$ must be greater than or equal to $|C_{opt}|$. This can be proved by contradiction as follows. Assume that there exists an instance $I' \models \Sigma'$ such that the number of changed cells in I' is less than $|C_{opt}|$. Let T be the set of changed tuples in I'. T represents a vertex cover of G and $|T| < |C_{opt}|$, which contradicts minimality of C_{opt}.

In the following, we prove that the number of changed cells is $|C_{2opt}| \cdot \min\{|R| - 1, |\Sigma|\}$, which is $2 \cdot \min\{|R| - 1, |\Sigma|\}$-approximate minimum, based on the fact that $\delta_{opt}(\Sigma', I) \geq |C_{opt}|$. The algorithm changes only attributes of tuples in C_{2opt}. Furthermore, we prove that the number of changed cells in each tuple in C_{2opt} is at most $\min\{|R| - 1, |\Sigma|\}$. It is clear that the maximum number of changed cells in each tuple is $|R| - 1$ because the first

attribute inserted into $Fixed\_Attrs$ cannot be changed (line 6 in Algorithm 11).

We show that after changing $|\Sigma'|$ attributes in t, the set $I' \setminus C_{2opt} \cup \{t\}$ satisfies Σ' and thus no other attributes in t need to be changed. In general, we prove that after the k-th change to t, $I' \setminus C_{2opt} \cup \{t\}$ can violate at most $|\Sigma'| - k$ FDs in Σ'. Let B be a changed attribute in t. If B was changed to a variable, there must exist an FD $X \rightarrow A \in \Sigma'$ such that $B \in X$. The reason is that if B does not appear in any FD, it cannot be changed by Algorithm 11, and if B only appears as a right-hand-side attribute in FDs in Σ', it can only remain unchanged or be changed to a constant. It follows that (t, t') cannot violate $X \rightarrow A$, for all $t' \in I' \setminus C_{2opt}$ after changing $t[B]$ to a variable and adding B to $Fixed\_Attrs$. If B was changed to a constant, there must exist an FD $X \rightarrow B \in \Sigma'$ and another FD $Y \rightarrow X$ implied by Σ' such that $Y \subset Fixed\_Attrs$ and values of attributes in Y are constants (refer to lines 2-9 in Algorithm 12). In successive iterations, attributes in X cannot be assigned to values other than the current constants in t_c, otherwise $Y \rightarrow X$ would be violated. It follows that (t, t') cannot violate $X \rightarrow B$, for $t' \in I' \setminus C_{2opt}$ in successive iterations. After changing $|\Sigma'|$ attributes in t, we do not need to perform further changes. Because $|\Sigma'| \leq |\Sigma|$, it follows that the maximum number of attributes changed for each tuple in C_{2opt} is $|\Sigma|$, which completes the proof.

□

In the following, we describe the worst-case complexity of Algorithms 11 and 12. Algorithm 12 has a complexity of $O(|R| + |\Sigma'|)$ because constructing t_c in line 1 costs $O(|R|)$, and the loop in lines 2-9 iterates at most $|\Sigma'|$ times. The reason is that, for each FD $X \rightarrow A \in \Sigma'$, there is at most one tuple in $I' \setminus C_{2opt}$ satisfying the condition in line 2 (otherwise, tuples in $I' \setminus C_{2opt}$ would be violating $X \rightarrow A$).

Constructing the conflict graph in line 1 in Algorithm 11 is performed in $O(|\Sigma'| \cdot n + |\Sigma'| \cdot |E|)$, where $|\Sigma'|$ is the number of FDs in Σ', n is the number of tuples in I and E is the set of edges in the resulting conflict graph. This step is performed by partitioning tuples in I based on LHS attributes of each FD in Σ' using a hashing function, and constructing sub-partitions within each partition based on right-hand-side attributes of each FD. Edges of the conflict graph are generated by emitting pairs of tuples that belong to the same partition and different sub-partitions. The approximate vertex cover is computed in $O(|E|)$

[44]. The loop in lines 4-17 iterates a number of times equal to the size of the vertex cover, which is $O(n)$. Each iteration costs $O(|R| \cdot (|R| + |\Sigma'|))$. To sum up, the complexity of finding a clean instance I' is $O(|\Sigma'| \cdot |E| + |R|^2 \cdot n + |R| \cdot |\Sigma'| \cdot n)$. Assuming that $|R|$ and $|\Sigma'|$ are much smaller than n, the complexity can be reduced to $O(|E| + n)$.

5.5 Uncertainty in the Relative Trust in Data vs. FDs

In practice, we may not be able to estimate the number of errors in the data. Thus, it can be difficult to determine a single value for threshold τ and it might be easier to provide a range of possible values of τ, corresponding to multiple repairs of Σ and I.

One way to obtain a small sample of possible repairs is to execute Algorithm 8 multiple times while randomly varying the value of τ within the specified range. This approach can be easily parallelized. However, this approach is inefficient when used for obtaining all possible repairs for two reasons. First, multiple values of τ could result in the same repair, and some executions of the algorithm would be redundant. Second, different invocations of Algorithm 9 are expected to visit the same states, which represents a waste of computational resources. To overcome these drawbacks, we develop an algorithm (Algorithm 13) that generates all repairs corresponding to a range of τ. We can use Algorithm 11 to find the corresponding clean data instance for each obtained FD set.

Algorithm 13 generates all repairs corresponding to the threshold range $\tau \in [\tau_l, \tau_u]$. Initially, threshold τ is set to τ_u. The search algorithm proceeds by visiting states in order of $\underline{\text{gc}}(.)$, and expanding PQ by inserting new states. Once a goal state is found, the corresponding FD repair Σ_h is added to the set of possible repairs. The set Σ_h corresponds to the parameter range $[\delta_P(\Sigma_h, I), \tau]$. Therefore, we set the new value of τ to $\delta_P(\Sigma_h, I) - 1$ in order to discover a new repair. Because the value of $\underline{\text{gc}}(.)$ depends on the value of τ, we enforce recomputation of $\underline{\text{gc}}(.)$ for all states in PQ. Note that states that have been previously removed from PQ because they were not goal states (line 13) cannot be goal states with respect to the new value of τ. The reason is that if a state is not a goal state for $\tau = x$, it cannot be a goal state for $\tau < x$ (refer to line 8). The algorithm terminates when PQ is empty, or when $\tau < \tau_l$.

Algorithm 13 `Range_Repair_FDs`$(\Sigma, I, \tau_l, \tau_u)$

1: $PQ \leftarrow \{(\phi, \ldots, \phi)\}$
2: $\tau \leftarrow \tau_u$
3: $FD\_Repairs \leftarrow \phi$
4: **while** PQ is not empty and $\tau \geq \tau_l$ **do**
5: Pick the state S_h with the smallest value of $\underline{gc}(.)$ from PQ
6: Let Σ_h be the FD set corresponding to S_h
7: Compute $C_{2opt}(\Sigma_h, I)$
8: **if** $|C_{2opt}(\Sigma_h, I)| \cdot \min\{|R| - 1, |\Sigma|\} \leq \tau$ **then**
9: Add Σ_h to $FD\_Repairs$
10: $\tau \leftarrow |C_{2opt}(\Sigma_h, I)| \cdot \min\{|R| - 1, |\Sigma|\} - 1$
11: For each state $S_i \in PQ$, recompute $\underline{gc}(S_i)$ using the new value of τ
12: **end if**
13: Remove S_h from PQ
14: **for** each state S_i that is a child of S_h **do**
15: Compute $\underline{gc}(S_i)$ (similar to Algorithm 9)
16: Insert S_i into PQ
17: **end for**
18: **end while**
19: **return** $FD\_Repairs$

5.6 Experimental Evaluation

In this section, we study the relationship between the quality of repairs and the relative trust determined by τ. Also, we show the efficiency of our cleaning algorithms.

5.6.1 Setup

All experiments were conducted on a SunFire X4100 server with a Quad-Core 2.2GHz processor, and 8GB of RAM. All computations are executed in memory. Repairing algorithms are executed as single-threaded processes, and we limit memory usage to 1.5GB. We use a

real data set, namely the Census-Income data set[1], which is part of the UC Irvine Machine Learning Repository. Census-Income consists of 300k tuples and 40 attributes (we only use 34 attributes in our experiments). To perform experiments on smaller data sizes, we randomly pick a sample of tuples.

We test two variants of Algorithm `Repair_Data_FDs`. The first version, called A^*`-Repair`, uses the A*-based search algorithm described in Section 5.3.2. The second variant, called `Best-First-Repair` uses best-first search to obtain FD repairs, as we described in Section 5.3. Both variants use Algorithm 11 to obtain the corresponding data repair. We use the number of distinct values to measure the weights of sets of attributes appended to LHS's of FDs (i.e., $w(Y)$). In our experiments, we adjust the relative threshold τ_r, rather than the absolute threshold τ (recall Section 5.1.2).

In order to assess the quality of the generated repairs, we first use an FD discovery algorithm to find all the minimal FDs with a relatively small number of attributes in the LHS (less than 6). In each experiment, we randomly select a number of FDs from the discovered list of FDs. We denote by I_c and Σ_c the clean database instance and the FDs, respectively. The data instance I_c is perturbed by changing the value of some cells such that each cell change results in a violation of an FD. Specifically, we inject two types of violations as follows.

- Right-hand-side violation: We first search for two tuples t_i, t_j that agree on XA for some FD $X \rightarrow A \in \Sigma$. Then, we modify $t_i[A]$ to be different from $t_j[A]$.
- Left-hand-side violation: We search for two tuples t_i, t_j such that for some FD $X \rightarrow A$, $t_i[X \setminus \{B\}] = t_j[X \setminus \{B\}]$, $t_i[B] \neq t_j[B]$ and $t_i[A] \neq t_j[A]$, where $B \in X$. We introduce a violation by modifying $t_i[B]$ to be equal to $t_j[B]$.

We refer to the resulting instance as I_d. In our approach, we concentrate on one method of fixing FDs, which is appending one or more attributes to LHS's of FDs. Therefore, we perform FDs perturbation by randomly removing a number of attributes from their LHS's. The perturbed set of FDs is denoted Σ_d. The cleaning algorithm is applied to (Σ_d, I_d), and

[1]http://archive.ics.uci.edu/ml/datasets/Census-Income+(KDD)

the resulting repair is denoted (Σ_r, I_r). The parameters that control the perturbation of data and FDs are (1) Data Error Rate, which is the fraction of cells that are modified, and (2) FD Error Rate, which is the fraction of LHS attributes that were removed. We use the following metrics to measure the quality of the modified data and FDs.

- Data precision: the ratio of the number of correctly modified cells to the total number of cells modified by the cleaning algorithm. A modification of a cell $t[A]$ is considered correct if the values of $t[A]$ in I_c and I_d are different, and either $t[A]$ in I_r is equal to $t[A]$ in I_c, or $t[A]$ is a variable in I_r.
- Data recall: the ratio of the number of correctly modified cells to the total number of erroneous cells (i.e., cells with different values in I_d and I_c).
- FD precision: the ratio of the number of attributes correctly appended to LHS's of FDs in Σ_d to the total number of appended attributes.
- FD recall: the ratio of the number of attributes correctly appended to LHS's of FDs in Σ_d to the total number of attributes removed from Σ_c while constructing Σ_d.

In order to measure the overall quality of a repair (Σ_r, I_r), we compute the harmonic averages of precision and recall for both data and FDs (also called F-scores). Then, we compute the average F-score for data and FDs, which we refer to as the combined F-score.

5.6.2 The Impact of Relative Trust on the Quality

In this experiment, we measure the combined F-score at various error rates. We use 5000 tuples from the Census-Income data set to represent the clean instance I_c, and we use an FD with 6 LHS attributes to represent Σ_c. Figure 5.6 shows the combined F-score of repairs at various error rates, for multiple values of τ_r. When only FDs perturbation is performed, we notice that the peak quality occurs at $\tau_r = 0\%$ (i.e., when no changes to data are allowed). At 5% data error rate and 30% FD error rate, the peak quality occurs at $\tau_r = 20\%$. At a higher FD error rate of 50%, we notice that the peak quality occurs at higher value (29%). Finally, when only data perturbation is performed, the peak quality

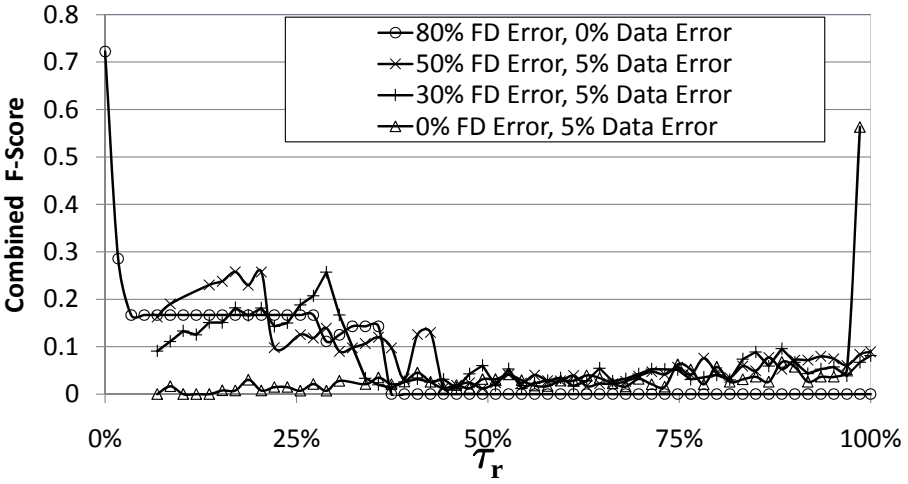

Figure 5.6: Repair quality at multiple error rates

occurs at $\tau = 100\%$ (i.e., the algorithm can freely change the data, while obtaining the cheapest FD repair, which is the original FD).

Note that the precision and recall for data repairs is relatively low due to the high uncertainty about the right cells to modify. For example, given an FD $A \rightarrow B$, and two violating tuples t_1 and t_2, we have four cells that can be changed in order to repair the violation: $t_1[A]$, $t_1[B]$, $t_2[A]$, and $t_2[B]$. Such uncertainty can be reduced by considering additional information such as the user trust in various attributes and tuples (e.g., [19, 22, 57]. Using such information to improve the data quality is not considered in our work.

5.6.3 Performance Results

In this section, we study the efficiency of our approach.

Scalability with the Number of Tuples

In this experiment, we show the scalability of our algorithms with respect to the number of tuples. We use two FDs, and we set τ_r to 1%. Figures 5.7(a) and 5.7(b) show the running time, and the number of visited states, respectively, against the number of tuples.

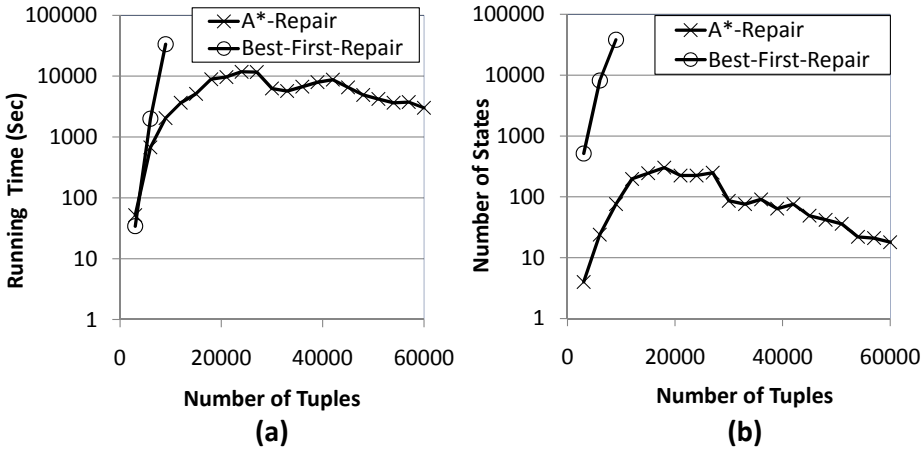

Figure 5.7: Performance at various instance sizes

When increasing the number of tuples in the range $[0, 20000]$, we notice that the number of unique difference sets increases, while the average frequency of difference sets remains relatively small, compared to τ. It follows that the computed lower bounds $\underline{gc}(S)$ are very loose because most difference sets considered by Algorithm 10 can be left unresolved (i.e., the condition in line 8 is true). It follows that the search algorithm needs to visit more states, as we show in Figure 5.7(b), which affects the overall running time.

When the number of tuples increases beyond 20000, we notice in Figure 5.7 that the running time, as well as the number of visited states, decreases. The reason is that the largest percentage of the running time is consumed by the state searching algorithm (Algorithm 9), which becomes more efficient after reaching a large number of tuples. The reason is that after reaching a certain number of tuples, the number of distinct difference sets stabilizes, and the frequencies of individual difference sets start increasing. It follows that most difference sets can no longer remain unresolved, and tighter lower bounds $\underline{gc}(S)$ are reported, which leads to decreasing the number of visited states (Figure 5.7(b)).

Algorithm `Best-First-Repair` does not depend on cost estimation, and thus, the execution time rapidly grows with the number of tuples in the entire range $[0, 60000]$.

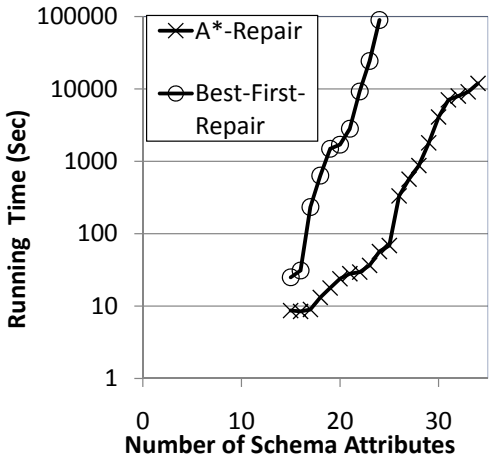

Figure 5.8: Salability with number of attributes

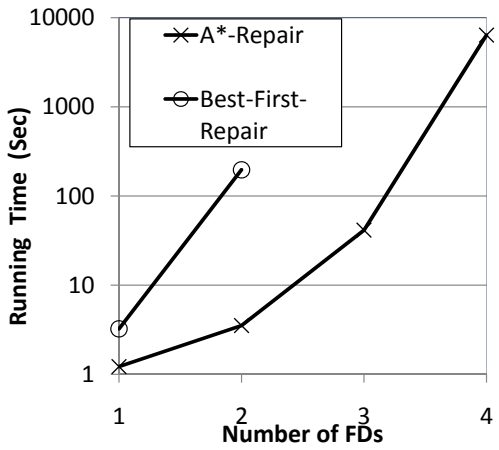

Figure 5.9: Salability with number of FDs

Scalability with the Number of Attributes

Figure 5.8 depicts the scalability of our approach with respect to the number of attributes. In this experiment, we used two FDs and 24000 tuples, and we set τ_r to 1%. We changed the number of attributes by excluding some number of attributes from the input relation. The running time increases with the number of attributes mainly because the size of state space increases exponentially with the number of attributes. Therefore, the state search algorithm has to visit more states before reaching a goal state.

Scalability with the Number of FDs

Figure 5.9 depicts the scalability of our approach with respect to the number of FDs. In this experiment, we used 10000 tuples, and we set τ_r to 1%. We use a single FD, and we replicate this FD multiple times to simulate larger sizes of Σ. The size of state space grows exponentially with the number of FDs. Thus, the searching algorithm visits more states, which increases the overall running time for both approaches: A^*`-Repair` and `Best-First-Repair`. Note that the algorithm `Best-First-Repair` did not terminate in 24 hours when the number of FDs is greater than 2.

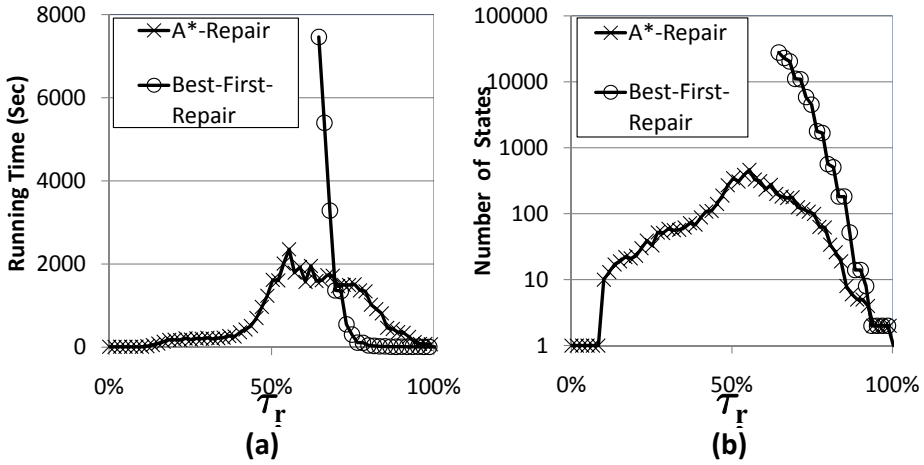

Figure 5.10: Effect of τ on (a) running time (b) visited states

Effect of Parameter τ

Figures 5.10(a) and 5.10(b) show the running time and the number of visited states, respectively, for various values of τ_r. In this experiment, we fix the number of tuples to be 5000, and we use Σ_d with one FD. The number of appended attributes ranges from 9 at $\tau_r = 10\%$ to 1 at $\tau_r = 99\%$. No repair could be found for τ_r less than 10%. We notice that at small values of τ, Algorithm A^*-`Repair` is orders of magnitude faster than Algorithm `Best-First-Repair`. This is due to the effectiveness (i.e., tightness) of the cost estimation implemented in Algorithm A^*-`Repair`. The lack of such estimation causes Algorithm `Best-First-Repair` to visit many more states.

As the value of τ_r increases up to 55%, we observe that Algorithm A^*-`Repair` becomes slower. The reason is that larger values of τ_r decreases the tightness of computed bounds $\underline{gc}(S)$. As τ_r increases beyond 55%, we notice an improvement in the running time as we only need to add very small number of attributes to reach a goal state.

Approximation Factors P and Q

Our approach provides approximate minimal repairs with approximation factors P and Q (refer to Definition 14 in Section 5.3.3). The upper bound on the actual approximation

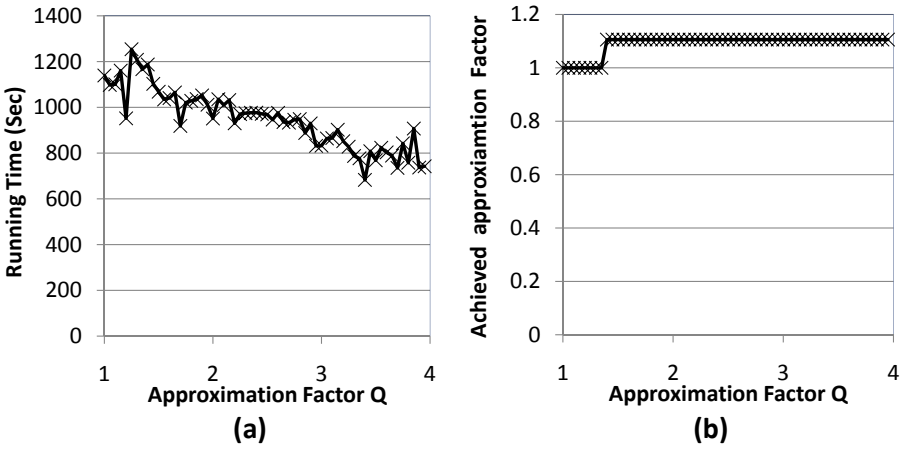

Figure 5.11: The effect of approximation factor Q

factor P that is achieved in our experiments is equal to $2 \cdot dist_d(I_d, I_r)/C_{2opt}(\Sigma_r, I_d)$, which is found to be less than 1.1 in all experiments we performed.

In Figures 5.11(a) and 5.11(b), we show the effect of the approximation factor Q on the running time of the state search algorithm, and the actual approximation factor that has been achieved, respectively. These figures suggest that using large values of Q can reduce the execution time, without having a significant effect on the optimality of the algorithm. For example, setting Q to 4 reduces the running time by 35%, while returning a goal state that is 1.1-approximate.

Uncertainty in Relative Trust

In this experiment, we assess the efficiency of two approaches that generate possible repairs for a given range of τ_r. In the first approach, denoted `Range-Repair`, we execute Algorithm 13, and we invoke the data cleaning algorithm (Algorithm 11) for each obtained FD repair. In the second approach, denoted `Sampling-Repair`, we invoke the algorithm A^*`-Repair` at a sample of possible values of τ_r. In this experiment, we used 5000 tuples, and one FD. We set the minimum value of τ_r to 0, and we varied the upper bound of τ in the range $[10\%, 30\%]$, which is represented by the X-axis in Figure 5.12. For the sampling approach, we started by $\tau_r = 0\%$, and we increased τ_r in steps of 1.7% (which is equal

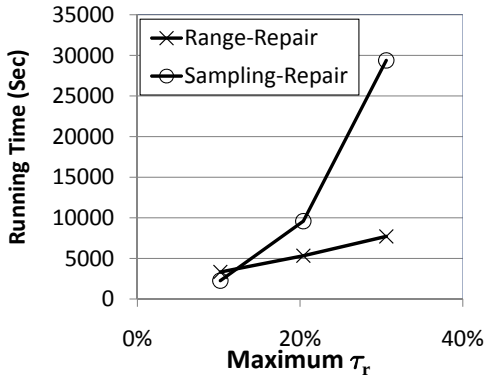

Figure 5.12: Performance under uncertain relative trust

to 13 in this experiment) until we reach the maximum value of τ_r. Figure 5.12 shows the running time for both approaches. We observe that `Range-Repair` outperforms the sampling approach, especially at wide ranges of τ_r. For example, for the range $[0, 30\%]$, `Range-Repair` is 3.8 times faster than `Sampling-Repair`.

Chapter 6

Conclusion and Future Work

In this chapter, we conclude the dissertation, and we provide directions for future work.

6.1 Conclusion

In this dissertation, we proposed a data cleaning approach that leverages the uncertainty in the data cleaning process in order to provide probabilistic data cleaning. We applied this approach to multiple data cleaning problems, namely duplicate elimination and repairing functional dependency violations. We also addressed the problem of repairing violations of functional dependencies (FDs) when the FDs are not trusted as well.

In the context of duplicate elimination, we defined a space of possible repairs that is generated by varying the parameter of any given parameterized deduplication algorithm. For a broad class of deduplication algorithm that are based on hierarchical clustering, we developed a technique that enables efficient generation of all possible repairs using a single execution of the cleaning algorithm. Also, we described how to compactly store the generated possible repairs, and use all stored repairs for probabilistic answering of user queries. Additionally, we proposed new types of queries that are uncertainty-aware, such as obtaining the most probable repair, and obtaining the probability that two given tuples are duplicates. Our experiments show that the overhead of our data cleaning approach is

negligible, compared to deterministic data cleaning, and that user queries can be answered efficiently using our data model.

To enable probabilistic repairing of FD violations, we developed a randomized algorithm that samples from a space of possible repairs. The sampling space contains all repairs that have minimal set of changed cells. We significantly improved the performance of the sampling algorithm by partitioning the input instance into multiple disjoin blocks that can be repaired independently. Furthermore, we showed how to modify our sampling algorithm to satisfy a set of hard constraints that prevent changing specific cells that are trusted to be clean. We performed a number of experiments to show the efficiency of our approach.

Finally, we studied the problem of repairing of FD violations when both the data and the given FDs are not clean. We leveraged the user relative trust in data and FDs in order to steer the cleaning process into performing the right changes. We proposed an algorithm that obtains a repair of data and FDs such that the amount of data changes is below a certain threshold, and the amount of FD changes is approximately minimum. The approximation factor only depends on the number of FDs, and the number of attributes. Also, we described how to extend our algorithm when the threshold on data changes is uncertain (i.e., defined as an interval). We performed several experiments to show the efficiency of our algorithm, and to show the effect of the relative trust in data versus FDs on the quality of the repairs.

6.2 Future Work

The main direction of our future work is enriching the data cleaning systems to be aware of the underlying uncertainty, and thus avoid destroying valuable information that could affect the reliability of query answering. Along this line of research, we envision the following points that we plan to pursue in our future work.

6.2.1 Simultaneously Repairing Multiple Types of Errors

In practice, multiple types of errors coexist in the same data instance. Different types of errors are expected to be dependent. For example, existence of near-duplicate tuples increases the chance of violating functional dependencies. In our dissertation, as well as most data cleaning systems, we focused on solving individual types of errors separately. Unfortunately, serial execution of data cleaning algorithm to solve different error types might lead to sub-optimal data quality due to their dependencies. Having a system that is aware of all existing error types and their interplay is expected to increase the overall quality of generated clean instances. Similar observations have been highlighted by Wenfei Fan et al. in [37]. The approach proposed in [37] aims at simultaneously solving two data quality problem: record matching, and violations of conditional functional dependencies. Also, in [21], Chaudhuri et al. proposed an approach that use aggregate constraints to improve the quality of duplicate elimination.

Our objective is to understand and formulate the dependencies among a large class of error types such as FD violations, missing values, duplicate records, and heterogenous formats. We believe that probabilistic data cleaning perfectly fits our goal. For example, consider the possible repairs obtained by a probabilistic duplicate elimination process. If a set of FDs are defined over data, we can compute a posterior probability distribution over the possible repairs that is conditioned on satisfying the FDs.

6.2.2 Modeling Patterns of Errors in Data

Errors in data are induced by different causes (e.g., noisy data sensors, data integration, and human errors). It follows that patterns of errors in different data instances are expected to be different as well. This fact is also visible in synthetic data generators that mimic the error patterns frequently seen in practice (e.g.,frequent spelling mistakes, swapping first and last names, and using default values) [25].

Unfortunately, existing data cleaning systems focus on the actual cleaning of data without first analyzing and investigating the patterns of errors in data and the underlying generative process. Jumping directly to rectifying errors in data before completely under-

standing the causes of errors leads to a number of problems. For example, current systems that repair FD violations use only a fixed type of data modifications to repair the input instance (e.g., either deleting tuples or modifying tuple attributes), regardless of the causes of violations. Knowing, for instance, that each tuple could be either completely correct or completely erroneous would favor using tuple deletion as a cleaning method. In general, we argue that collecting information about error patterns is crucial to successful, high-quality data cleaning systems.

Our objective is to capture common error patterns in data, and to construct a probabilistic generative process to replicate such errors. This process is considered a reverse engineering of data perturbations that led to the errors. Supervised or semi-supervised learning approaches could be useful tools for modeling error patterns in data. A second goal is to allow using the obtained error model in steering the data cleaning process.

6.2.3 Learning Probabilities of Parameter Values

In our work, we provided a method to induce a probability distribution on the possible repairs by assuming a probability distribution over the parameter values of the cleaning algorithms (refer to Chapter 3). A open question is how to obtain (or estimate) the probability distribution of the possible parameter values, denoted f_τ. We envision two possible directions that could be used for obtaining f_τ. The first direction is relying on an expert user to manually obtain an estimate of f_τ based on analyzing the quality of the repairs generated by the clustering algorithm for various data sets that are close to the actual data.

The second direction is using supervised machine learning techniques to learn the distribution f_τ. In the following, we briefly outline a possible implementation of this direction. Assume that we have multiple training data sets $S_1, \ldots, S_k$ that are representative of the input database instance. Each data set S_i is associated with its correct clustering X_i. For each data set S_i, we obtain the parameter value that results in the closest clustering to X_i (based on some distance function such as Rand Index [54]). Finally, we use a non-parametric probability estimation method such as histograms or kernel density estimation

to obtain a probability distribution of τ given the parameter values of the training data sets.

For example, given five data sets $S_1, \ldots, S_5$, with corresponding correct clustering $X_1, \ldots, X_5$, respectively, assume that the parameter values leading to the highest quality clusterings are 0.1, 0.6, 0.7, 0.2, 0.1, respectively. Constructing a histogram over the parameter range $[0, 1]$ with a fixed bin-width of 0.5 results in the following histogram: $[0, 0.5) : 3/5; [0.5, 1] : 2/5$.

In our future work, we will concentrate on the supervised learning approach, and examine multiple implementations for obtaining training data sets $S_1, \ldots, S_k$, and using various probability distribution estimation methods. Also, we are planning to test the effect of the obtained probability distribution of the algorithm parameter on the quality of the user queries.

6.2.4 Uncertainty-aware Accuracy Evaluation of Query Answers

Evaluating the accuracy of query answers is an important task for assessing the end-to-end reliability of the data cleaning process with respect to multiple query types. A key issue is quantifying the similarity between the obtained query answers and the correct query answers. Multiple metrics have been proposed to compare deterministic query answers to the correct answers. Some of the widely used metrics are summarized as follows.

- For queries that return unordered sets of tuples, the accuracy is measured using the precision, which is the number of the correct tuples returned over the total number of returned tuples, and the recall, which is equal to the number of correct tuples returned over the total number of correct answers [77]. The harmonic average of the precision and the recall, called the F-measure, is commonly used for combining the two metrics.

- For top-k queries, the accuracy metric should consider not only having the correct answers in the query results, but also having the correct rank for each answer. Possible metrics that measure the correlation between two rankings of a given set of

objects include Kendall tau coefficient and Spearman's rank correlation coefficient [52]. In the information retrieval literature, two commonly used metrics to measure the accuracy of the returned top-k are the precision evaluated at a given cut-off rank (written $P@n$), and the average precision [77].

- For aggregate queries, the accuracy metric should measure the distance between the expected (correct) aggregate value and the returned value for each group of tuples returned. The overall accuracy could be computed as the average accuracy across all groups.

In presence of uncertainty in the database generated by our probabilistic cleaning approach, it is necessary to develop new metrics to measure the accuracy of query results while taking into consideration uncertainty in query results. Ideally, the probabilities of the possible repairs should be strongly correlated to their quality, based on some deterministic quality metric. Thus, it is possible to use the following two metrics to evaluate the quality of the probabilistic cleaning.

- The correlation between the probabilities of possible repairs and their quality (e.g., the precision and recall). One correlation measure is the Pearson correlation coefficient which measures the linear dependence between two variables [70].
- The range of qualities that is spanned by the possible repairs.

Query answering is based on the possible worlds semantic (Section 2.3.2). That is, the possible query answers are semantically equivalent to the union of the deterministic query answers corresponding to each possible repair. Therefore, it is possible to adopt the same evaluation scheme as follows. For each possible repair, we obtain the corresponding result set, the quality of this result set, and probability of the result set (which is equal to the probability of the repair). Then, we compute the two described measures: the correlation between the probabilities of the result sets and their qualities, and the range of quality covered by all result sets. The exact quality metric that should be used depends on the query type. For example, for top-k queries, we could use the Kendall tau coefficient to measure the quality of each possible result set.

One challenge in using the described evaluation method is the need for extracting the individual result sets corresponding to the possible repairs, and computing their probabilities. In case that the number of the possible repairs is very large, we need to provide more efficient techniques to obtain the described measures without extracting individual result sets. An alternative solution is extracting the k most probable result sets, for a reasonable value of k. Then, we perform the quality evaluation based on these k repairs only.

In our future work, we plan to define other possible metrics to evaluate the quality of query results in presence of uncertainty, and develop efficient algorithms to compute such metrics.

References

[1] PostgreSQL database system, http://www.postgresql.org.

[2] UIS data generator, http://www.cs.utexas.edu/users/ml/riddle/data.html.

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

[4] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):158–187, 1991.

[5] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[6] Periklis Andritsos, Ariel Fuxman, and Renée J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.

[7] Arvind Arasu, Christopher Ré, and Dan Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, pages 952–963, 2009.

[8] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Scalar aggregation in fd-inconsistent databases. In *ICDT*, pages 39–53.

[9] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

[10] Egon Balas and Manfred W. Padberg. Set partitioning: A survey. *SIAM Review*, 1976.

[11] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD*, 2003.

[12] Rudolf Bayer. The universal B-Tree for multidimensional indexing: general concepts. In *WWCA*, 1997.

[13] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

[14] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.

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

[16] George Beskales, Mohamed A. Soliman, Ihab F. Ilyas, and Shai Ben-David. Modeling and querying possible repairs in duplicate detection. *PVLDB*, 2(1):598–609, 2009.

[17] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007.

[18] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

[19] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.

[20] Surajit Chaudhuri, Venkatesh Ganti, and Rajeev Motwani. Robust identification of fuzzy duplicates. In *ICDE*, pages 865–876, 2005.

[21] Surajit Chaudhuri, Anish Das Sarma, Venkatesh Ganti, and Raghav Kaushik. Leveraging aggregate constraints for deduplication. In *SIGMOD Conference*, pages 437–448, 2007.

[22] Fei Chiang and Renée J. Miller. A unified model for data and constraint repair. In *ICDE*, pages 446–457, 2011.

[23] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1/2):90–121, 2005.

[24] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Computing consistent query answers using conflict hypergraphs. In *CIKM*, pages 417–426, 2004.

[25] Peter Christen. Probabilistic data generation for deduplication and data linkage. In *IDEAL*, pages 109–116, 2005.

[26] Peter Christen and Tim Churches. Febrl. freely extensible biomedical record linkage, http://datamining.anu.edu.au/projects.

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

[28] Carlo Curino, Hyun Jin Moon, Letizia Tanca, and Carlo Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In *ICEIS (1)*, pages 323–332, 2008.

[29] Yang C.Yuan. Multiple imputation for missing data: Concepts and new development. In *the 25th Annual SAS Users Group International Conference*, 2002.

[30] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.

[31] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.

[32] Wayne W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. *TDWI Report Serie*, 2002.

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

[34] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[35] Ronald Fagin and Moshe Y. Vardi. The theory of data dependencies - an overview. In *ICALP*, pages 1–22, 1984.

[36] Wenfei Fan. Dependencies revisited for improving data quality. In *PODS '08*, pages 159–170, New York, NY, USA, 2008. ACM.

[37] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Interaction between record matching and data repairing. In *SIGMOD Conference*, pages 469–480, 2011.

[38] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328), 1969.

[39] Gary William Flake, Robert Endre Tarjan, and Kostas Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4), 2003.

[40] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Consistent query answers on numerical databases under aggregate constraints. In *DBPL*, pages 279–294, 2005.

[41] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Preferred database repairs under aggregate constraints. In *SUM*, pages 215–229, 2007.

[42] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. *Journal of Computer and System Sciences*, 73(4):610–635, 2007.

[43] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, pages 371–380, 2001.

[44] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[45] Parke Godfrey, John Grant, Jarek Gryz, and Jack Minker. Integrity constraints: Semantics and applications. In *Logics for Databases and Information Systems*, pages 265–306, 1998.

[46] Georg Gottlob. Computing cores for data exchange: new algorithms and practical solutions. In *PODS*, pages 148–159, 2005.

[47] Sergio Greco and Cristian Molinaro. Approximate probabilistic query answering over inconsistent databases. In *ER*, pages 311–325, 2008.

[48] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: An efficient clustering algorithm for large databases. In *SIGMOD Conference*, pages 73–84, 1998.

[49] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[50] Oktie Hassanzadeh, Fei Chiang, Renée J. Miller, and Hyun Chul Lee. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.

[51] Y Huhtala, Juha Karkkainen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, pages 100–111, 1999.

[52] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[53] Tomasz Imieliński and Jr. Witold Lipski. Incomplete Information in Relational Databases. *J. ACM*, 31(4):761–791, 1984.

[54] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall College Div, 1988.

[55] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher M. Jermaine, and Peter J. Haas. MCDB: a monte carlo approach to managing uncertain data. In *SIGMOD Conference*, pages 687–700, 2008.

[56] Richard M. Karp and Michael Luby. Monte-carlo algorithms for enumeration and reliability problems. In *FOCS*, pages 56–64, 1983.

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

[58] Stefan Kramer and Bernhard Pfahringer. Efficient search for strong partial determinations. In *KDD*, pages 371–378, 1996.

[59] Mong-Li Lee, Tok Wang Ling, and Wai Lup Low. IntelliClean: a knowledge-based intelligent data cleaner. In *KDD*, pages 290–294, 2000.

[60] Andrei Lopatenko and Leopoldo E. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *ICDT*, pages 179–193, 2007.

[61] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT*, pages 350–364, 2000.

[62] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[63] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.

[64] H. Müller and j. Problems, Methods and Challenges in Comprehensive Data Cleansing. Technical Report HUB-IB-164, Humboldt-Universität zu Berlin, Institut für Informatik, 2003.

[65] Mary H. Mulry, Susanne L. Bean, D. Mark Bauder, Deborah Wagner, Thomas Mule, and Rita J. Petroni. Evaluation of estimates of census duplication using administrative records information. *Jour. of Official Statistics*, 2006.

[66] Judea Pearl. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley, 1984.

[67] Reinhard Pichler and Sebastian Skritek. The complexity of evaluating tuple generating dependencies. In *ICDT*, pages 244–255, 2011.

[68] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[69] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[70] Joseph L. Rodgers and Alan W. Nicewander. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician*, 42(1):59–66, 1988.

[71] Anish Das Sarma, Omar Benjelloun, Alon Y. Halevy, and Jennifer Widom. Working models for uncertain data. In *ICDE*, 2006.

[72] Prithviraj Sen and Amol Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, pages 596–605, 2007.

[73] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[74] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[75] Benjamin J. Tepping. A model for optimum linkage of records. *Journal of the American Statistical Association*, 1968.

[76] Maurice van Keulen, Ander de Keijzer, and Wouter Alink. A probabilistic XML approach to data integration. In *ICDE*, pages 459–470, 2005.

[77] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

[78] Vincent Ventrone. Semantic heterogeneity as a result of domain evolution. *SIGMOD Rec.*, 20:16–20, December 1991.

[79] Jef Wijsen. Condensed representation of database repairs for consistent query answering. In *ICDT*, pages 375–390, 2003.

[80] Jef Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768, 2005.

[81] 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.